PROGRAMMING LANGUAGE ABSTRACTIONS FOR EXTENSIBLE SOFTWARE COMPONENTS

THÈSE N^{O} 2930 (2004)

PRÉSENTÉE À LA FACULTÉ INFORMATIQUE ET COMMUNICATIONS

Institut d'informatique fondamentale

SECTION D'INFORMATIQUE

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Matthias ZENGER

Diplom-Informatiker, Universität Karlsruhe, Allemagne
et de nationalité allemande

acceptée sur proposition du jury:

Prof. M. Odersky , directeur de thèse
Prof. R. Guerraoui, rapporteur
Prof. M. Mezini, rapporteur
Prof. O. Nierstrasz, rapporteur

Lausanne, EPFL
2004

Matthias Zenger

Programming Language Abstractions for Extensible Software Components

Doctoral Thesis
EPFL, Switzerland

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Laboratoire des Méthodes de Programmation
Institut d'Informatique fondamentale
Faculté Informatique et Communications
École Polytechnique Fédérale de Lausanne

Programming Methods Laboratory
Institute of Core Computing Science
School of Computer and Communication Sciences
Swiss Federal Institute of Technology, Lausanne, Switzerland

This document was set in ITC Charter using the LaTeX typesetting system on MacOS X. The graphical content was produced with OmniGraffle Professional.

Acknowledgments

First and foremost I would like to express my sincere gratitude and appreciation to my supervisor, Prof. Martin Odersky, for his encouragement, guidance, and support during the last six years. His expertise and understanding and the countless inspiring discussions have been of great importance and influence to me, both academically and personally. From 1997 on, I enjoyed working in his group.

I would like to thank my friend and former colleague Dr. Christoph Zenger for his interest in my work, his advice and help, as well as the many loud and passionate discussions about work and life in general. I appreciated the collaboration with him in Karlsruhe, Adelaide, and Lausanne. Many thanks also go to Prof. Martin Sulzmann for his support and the numerous interesting discussions in Adelaide and Melbourne.

I would like to thank the members of the thesis jury, Prof. Karl Aberer, Prof. Rachid Guerraoui, Prof. Mira Mezini, and Prof. Oscar Nierstrasz, for the efforts and the time they have spent on the examination of this dissertation.

Furthermore, I am thankful to my colleagues at the Programming Methods Laboratory of EPFL for their support and their collaboration. In particular, I thank Dr. Erik Stenman for proof-reading parts of this thesis. Thanks also to all other people I was working with in the last years. I had the pleasure to publish papers together with Dr. Jim Buckley, Vincent Cremet, Matthias Jacob, Prof. Günter Kniesel, Dr. Tom Mens, Prof. Michael Philippsen, Dr. Christine Röckl, and Dr. Awais Rashid.

Last but not least, I would like to thank my parents for the help and support they provided me throughout my entire life. I am especially grateful to my wife, Marie-Louise, without whose love, encouragement, and understanding, I would not have finished this thesis.

Nyon, Switzerland, October 2003

Abstract

With the growing demand for software systems that can cope with an increasing range of information processing tasks, the reuse of code from existing systems is essential to reduce the production costs of systems as well as the time to manufacture new software applications. For this reason, component-based software development techniques gain increasing attention in industry and research. Component technology is driven by the promise of building software by composing off-the-shelf components provided by a software component industry. Therefore, component technology emphasizes the independent development and deployment of components. Even though components look like perfect reusable assets, they embody general software solutions that need to be adapted to deployment-specific needs and therefore cannot be deployed "as is" in general. Furthermore, as architectural building blocks, components are subject to continuous change. For these reasons, it is essential that components can easily be extended by both the component manufacturer to create new versions of components and by third-parties that have to adapt components for use in specific software systems. Since in both cases concrete changes cannot be foreseen in general, mechanisms to integrate unanticipated extensions into components and component systems are required.

While today many modern programming techniques, methodologies, and languages provide means that are well suited for creating static black-box components, the design and implementation of extensible components and extensible software systems often remains a challenge. In practice, extensibility is mostly achieved through ad-hoc techniques, like the disciplined use of design patterns and component frameworks, often in conjunction with meta-programming. The use of design patterns and component frameworks requires a rigorous coding discipline and often forces programmers to write tedious "boilerplate" code by hand, which makes this approach fragile and error-prone. Meta-programming techniques on the other hand are rather code-centric and mostly source code-based. Therefore, they are often not very suitable for today's component technology practice that stresses the binary reuse of black-box components.

In this thesis I argue that technical difficulties in the development of extensible software components are due to the lack of appropriate programming language abstractions. To overcome the problems, concrete programming language mechanisms are proposed to facilitate the creation of extensible software. The proposed language features are strongly typed to help the programmer extend systems safely and consistently.

The first part of the thesis illustrates the vision of truly extensible software components by proposing a simple theoretical model of first-class components built on top of a conventional class-based object-oriented language. This typed model includes a small set of primitives to dynamically build, compose, and extend software components safely, while supporting features like explicit context dependencies, late composition, unanticipated component extensibility, and strong encapsulation.

The second part takes some ideas from the theoretical model and applies them in the design of the programming language KERIS. KERIS extends JAVA with an expressive module system featuring extensible modules. The main contributions are:

- A module system that combines the benefits of classical module systems for imperative languages with the advantages of modern component-oriented formalisms. In particular, modules are reusable, generic software components that can be linked with different cooperating modules without the need for resolving context dependencies by hand.
- A module composition scheme based on aggregation that makes the static architecture of a system explicit, and
- A type-safe mechanism for extending atomic modules as well as fully linked systems statically by replacing selected subsystems with compatible versions without needing to re-link the full system. The extensibility mechanism is non-invasive; i.e. it preserves the original version and does not require access to source code.

The overall design of the language was guided by the aim to develop a pragmatic, implementable, and conservative extension of JAVA which supports software development according to the *open/closed principle:* Systems written in KERIS are closed in the sense that they can be executed, but they are open for unanticipated extensions that add, refine, or replace modules or whole subsystems.

The last part of the thesis finally presents a case study which compares an extensible JAVA compiler implemented using mainstream object-oriented language features with one that was written in KERIS. It shows how in practice, extensible modules can be used to develop extensible systems safely and efficiently.

Zusammenfassung

Bei Software-Anwendungen, die mit ständig neuen Informationsverarbeitungsanforderungen konfrontiert sind, trägt die Wiederverwendung von Code wesentlich dazu bei, sowohl die Produktionskosten, als auch die Zeit für die Entwicklung neuer, verwandter Systeme zu reduzieren. Aus diesem Grund kommt komponentenbasierten Softwaretechnologien verstärkt Aufmerksamkeit im Entwicklungs- und Forschungsbereich zu. Komponententechnologie basiert auf der Vision, Software primär durch eine Kombination von vorgefertigten Komponenten zu erstellen, welche von einer globalen Softwarekomponentenindustrie angeboten werden. Obwohl Komponenten eigentlich als ideal wiederverwendbare Bausteine erscheinen, stellen sie dennoch allgemeine Softwarelösungen dar, die stets an anwendungsspezifische Bedürfnisse angepasst werden müssen und daher auch selten in ihrer allgemeinen Form eingesetzt werden können. Außerdem unterliegen Softwarekomponenten, als grundlegende architekturelle Bausteine, natürlicherweise ständigen Veränderungen. Es ist daher unverzichtbar, dass Komponenten auf einfache Art und Weise, sowohl vom Hersteller, als auch von Klienten, erweitert werden können. Da in beiden Fällen die zukünftigen, konkreten Änderungen selten im Voraus absehbar sind, muss es möglich sein, auch unvorhergesehene Erweiterungen an Komponenten und Komponentensystemen vornehmen zu können.

Während viele moderne Programmiertechniken und Programmiersprachen durchaus gut für die Entwicklung statischer *Black-Box-Komponenten* geeignet sind, bleibt die Entwicklung von *erweiterbaren* Komponenten und komponentenbasierten Softwaresystemen oft eine Herausforderung. In der Praxis wird Erweiterbarkeit hauptsächlich mit Hilfe von Ad-hoc-Techniken erzielt; z.B. durch eine disziplinierte Verwendung von Entwurfsmustern und Rahmenwerken, oft in Verbindung mit Meta-Programmiertechniken. Eine konsequente Verwendung von Entwurfsmustern und Rahmenwerken zwingt den Programmierer oft dazu, langweiligen und dadurch auch fehleranfälligen Anpassungscode von Hand zu schreiben. Meta-Programmierung, auf der anderen Seite, basiert meist auf Quellcode und ist damit nicht sonderlich geeignet für den Einsatz in einer auf binären Komponenten beruhenden Technologie.

In dieser Dissertation wird argumentiert, dass die technischen Schwierigkeiten bei der Entwicklung von erweiterbaren Softwarekomponenten auf einen Mangel an geeigneten Abstraktionen auf der Programmiersprachenebene zurückzuführen sind. Zur Lösung des Problems werden konkrete Programmiersprachen-Mechanismen vorgeschlagen, die die Entwicklung von

erweiterbarer, komponentenbasierter Software vereinfachen und damit eine sichere und konsistente Evolution von Systemen ermöglichen sollen.

Im ersten Teil der Dissertation wird die Vision von flexibel erweiterbaren Softwarekomponenten an Hand eines einfachen theoretischen Modells veranschaulicht, welches Komponenten im Kontext einer konventionellen, klassenbasierten, objekt-orientierten Sprache einführt. Das typisierte Modell definiert eine kleine Menge von Primitiven mit deren Hilfe auf typsichere Art und Weise dynamisch Komponenten erzeugt, kombiniert und erweitert werden können.

Im zweiten Teil werden Ideen des theoretischen Modells aufgegriffen und bei dem Entwurf der Programmiersprache KERIS eingesetzt. KERIS erweitert die Programmiersprache JAVA um ein ausdrucksstarkes Modulsystem. Diese Arbeit liefert folgende Beiträge:

- Ein Modulsystem, welches die Vorzüge klassischer, imperativer Modulsysteme mit den Vorteilen moderner komponentenorientierter Formalismen verbindet. Module repräsentieren wiederverwendbare, generische Softwarekomponenten, die mit anderen Modulen kombiniert werden können ohne dass Kontextabhängigkeiten von Hand aufgelöst werden müssen.
- Ein Prinzip zur Modulkomposition, welches auf Aggregation beruht und welches die statische Architektur eines Systems explizit macht.
- Ein typsicherer Mechanismus, um sowohl atomare Module, als auch voll verlinkte Systeme statisch zu erweitern, indem ausgewählte Subsysteme durch kompatible Versionen ersetzt werden, ohne dass es notwendig wird, das gesamte System erneut zu linken. Der Erweiterungsmechanismus ist nicht destruktiv; er erzeugt eine neue Version einer Softwarekomponente ohne die alte Version zu verändern und ohne auf den Quellcode der alten Version zuzugreifen.

Ein Ziel des Sprachdesigns war es, eine pragmatische und implementierbare Erweiterung von JAVA zu konzipieren, welche es erlaubt, komponentenbasierte Software nach dem *Open/Closed Prinzip* zu entwickeln: Systeme die in KERIS geschrieben sind, sind geschlossen, in dem Sinne, dass sie ausführbar sind; sie sind aber auch offen für zukünftige Erweiterungen, die Module oder ganze Subsysteme neu hinzufügen oder ersetzen.

Im letzten Teil der Dissertation wird eine Fallstudie präsentiert, welche einen erweiterbaren JAVA Übersetzer, der mit gängigen, objekt-orientierten Sprachmitteln geschrieben wurde, mit einem in KERIS geschriebenen Übersetzer vergleicht. Mit Hilfe dieser Fallstudie wird demonstriert, welche Möglichkeiten KERIS in der Praxis bietet, erweiterbare Systeme sicher und effizient zu implementieren.

Contents

Chapter 1

Extensible Component-Based Software

1.1 Introduction

1.1.1 Reusability

With the growing demand for software systems that can cope with an increasing range of information processing tasks, the reuse of code from existing systems becomes more and more important. *Software reusability* refers to the ability of software elements to serve for the construction of many different software products [134]. Software reuse is motivated by the observation that software systems often share common elements. By reusing existing software components for the construction of new software systems, one can expect lower costs due to a reduced development time, decreased maintenance requirements, as well as increased reliability and consistency [140, 107, 134, 166]. Furthermore, reusing software means that less software has to be written and consequently that more time and effort may be spent on improving other factors, such as correctness and robustness.

Mainly for these reasons, *component-based software* development techniques gain increasing attention in industry and research. Component technology is driven by the promise of building software by composing off-the-shelf components provided by a software component industry [195]. This is also why component-oriented software engineering emphasizes the independent development and deployment of software components.

Even though components look like perfectly reusable assets, it is, unfortunately, often quite difficult to reuse software components off-the-shelf. Even though software development is a highly repetitive activity which involves frequent use of common patterns, there is a considerable variation in how these patterns can be used and combined. Without mechanisms supporting the adaptation and extension of software components, programmers are forced into, what Meyer [134] calls, the *reuse-redo dilemma:* Either the component is reused exactly as it is, or the job has to be redone completely.

1.1.2 Extensibility

Software is *extensible* if it can be adapted to possibly unanticipated changes in the specification. Extensibility is an important property for software because of the following reasons:

- At the basis of all software lies some human phenomenon and hence fickleness, yielding ongoing changes in the specification and the implementation of systems [134]. This is why Nierstrasz encourages to see software as a living and evolving entity which is developed and maintained by people [147]. Software can be changed more easily, if it is designed to be extensible [7].
- Component technology is based on the notion of components being independently developed and deployed by unrelated parties [195]. Since it is quite unlikely that components from external vendors fit into a specific deployment scenario off-the-shelf, it is necessary that software components are adaptable, not only by the manufacturer, but also by third party users.
- Many software systems share a common architecture or even large parts of the implementation. Such *families of software systems* [158, 27] are much easier to derive from a base system if the base system is extensible. Similarly, software *product-lines* [99, 205] rely heavily on a mechanism for creating variants of a system which all share a common structure and some common functionality, but which are equipped with possibly different components.

As these points show, extensibility boosts significantly software reuse. Since reusability is one of the main aims of component technology, we also consider extensibility to be a major factor in the development of component-based software systems.

Given the fundamental importance of extensibility, it is ironic that systems are often not explicitly engineered with this property in mind. In practice, programmers avoid extensible designs and implementations often for the following reasons [7]:

- Extensibility is a technical challenge that increases the complexity of software, making it more difficult to develop, test, and deploy.
- Extensible designs and implementations are more time consuming and are therefore more cost-intensive initially.
- Successful extensible designs and implementations require some knowledge about the way a system is going to be extended at a later time. The less is known about possible evolution scenarios, the more difficult it is to keep a system open for future extensions.
- Extensibility often decreases the performance.

While there is indeed a trade-off between extensibility and performance, it turns out in practice that the parts of the system for which extensibility is most beneficial, are often not the most performance-critical ones.

Programmers develop extensible software mainly for reducing the cost of implementing new or similar functionality in a system. If it is clear from the beginning that a system will be modified in the future, or will possibly be reused by third parties, it pays off to spent extra time and money on extensibility considerations. As Allen points out in [7], the *extreme programming* literature [20, 21] compares the addition of extensibility to the purchase of a stock option: You purchase the option early so that you can easily extend the system at a later date. If you eventually exercise this option, you can greatly benefit from it, even compensating for the critical items in the previous list.

1.2 Characteristics of Extensibility Mechanisms

Change is pervasive in software development. It involves changes of the requirements, of the design, of the implementation, of data representations, etc. This thesis focuses mainly on implementation-related issues, in particular on implementation techniques and formalisms (i.e. programming languages) that support the development of extensible software. This section will review important factors that characterize extensibility mechanisms on an abstract level [40, 132]. The terminology will be used in the next section to setup a classification of techniques for extending software.

Object of change. Different extensibility mechanisms differ in what software artifacts they change and at what time this happens. Mechanisms that introduce extensions directly into the source code do this typically at or before *compile-time*, whereas mechanisms that extend binaries or intermediate code representations like bytecode files typically operate at *link-* or *load-time*. Extensibility mechanisms that are applied before *runtime* are said to evolve a system *statically*, while all other mechanisms provide some form of *dynamic* software evolution. This distinction is sometimes unclear if link-time or load-time coincides with runtime (e.g. like in Java).

Anticipation. We have to distinguish between mechanisms where changes or variations of a software product have to be *anticipated* and others which support *unanticipated* requirement changes. Parameterization is, for instance, a form of anticipation. It allows one to vary a certain predefined set of features. Inheritance and overriding in combination with late binding, on the other hand, make it possible to extend software without anticipating all possible directions in which a system may evolve in future.

Invasiveness. Extensibility mechanisms that introduce or modify features destructively are called *invasive*. Invasive changes that are applied *in place* (i.e. that are not applied to a copy of the original software artifact) have a *global impact* and possibly influence all other depending components. A mechanism that supports non-invasive changes has to provide a way to formulate extensions *modularly*.

Versioning. To avoid that incompatible changes invalidate other software components and therefore endanger the consistency of the whole system, extensibility mechanisms often support some form of *versioning* [183, 195]. While in unversioned environments changes are typically invasive in the sense that new versions overwrite old ones, versioned environments provide means to distinguish old from new versions. In systems that support versioning *statically*, new and old versions can physically coexist at compile-time, but they are identified at runtime and therefore cannot be used simultaneously in the same context [40]. In contrast to this, fully versioned systems do not only distinguish versions statically, they also distinguish versions dynamically at runtime, allowing two different versions of one component being deployed simultaneously side by side. This is particularly relevant for the dynamic evolution of systems (e.g. systems that allow components to be *hot swapped*, i.e. dynamically replaced with compatible versions). Here, safe updates of existing components often require that new clients of the component use the services provided by the new version whereas existing clients of the old component continue to access the services of the old one.

Independent extensibility. Software changes may be carried out *sequentially* or in *parallel*. With sequential software evolution, changes are always applied to the last, most recent version of a component. For the case of parallel evolution it may happen that a component gets extended independently by different parties at the same time.

If software is changed in parallel, we have to distinguish between *convergent changes* and *divergent changes*. With convergent changes, parallel versions can be merged or integrated together into a new combined version [131]. For divergent changes, different versions of a component coexist indefinitely without having the possibility to combine the changes and thus use the two extensions jointly. If, for instance, inheritance is used to extend a class in different ways, object-oriented languages with single inheritance do not permit different extensions to be combined into a single class — changes diverge in this case. Languages with multiple inheritance, on the other hand, allow one to consolidate independent class extensions into a single subclass.

Extensibility mechanisms which allow programmers to evolve components in parallel and which make it possible to integrate several, independently developed extensions into a combined system support the notion of *independent extensibility* [194].

Safety. We distinguish between extensibility mechanisms that provide *static* and *dynamic safety*. A mechanism features static safety regarding certain erroneous program behaviors if it ensures at compile-time that the extended system will never be subject to these erroneous behaviors at runtime. An extensibility mechanism provides dynamic safety if there are built-in provisions for preventing or restricting undesired behavior of software extensions at runtime.

There are many different notions of safety. One of them is *security*, which aims at protecting the software from unauthorized access to sensitive parts of a system or to certain resources. Another is *behavioral safety*, which covers crashes and unpredictable or meaningless behavior at runtime. Yet another notion is *backward compatibility*. Backward compatibility guarantees that former versions of a software component can safely be replaced by newer versions without the need for global coherence checks during or after load-time.

Obviously, the kind and degree of safety that is required has a direct influence on the extensibility mechanisms. For example, a certain degree of static safety can be achieved by a programming language's type system at compile-time, while dynamic type tests have to be used for those cases that are not covered by the static type system. Similarly, systems that support dynamic loading need coherence checks at load-time to ensure that extended components "fit" the rest of the system. Such checks are even necessary for systems that guarantee safety statically but support some form of separate compilation. Here, components that are compiled together might not necessarily also be deployed together.

1.3 Classification of Extensibility Mechanisms

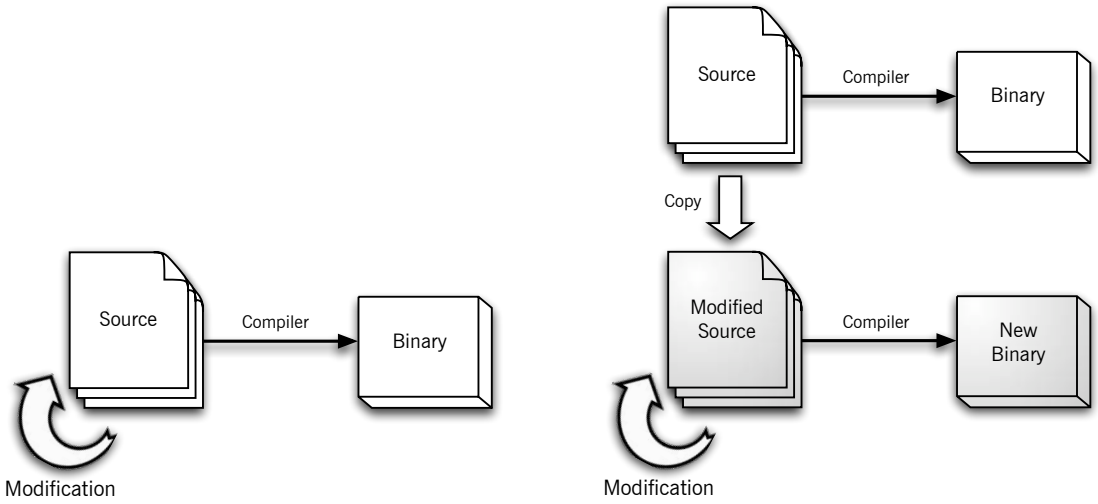
We will now propose a classification of extensibility mechanisms based on what artifacts are changed and in what way they are changed. In general, we distinguish between three different forms of extensibility: white-box extensibility, gray-box extensibility, and black-box extensibility.

1.3.1 White-Box Extensibility

White-box extensibility refers to the ways in which a software system can be extended by modifying or adding to the source code. This is the least restrictive and most flexible form of extensibility. Depending on the way changes are applied, we have to distinguish further between *open-box extensibility* and *glass-box extensibility* [7].

1.3.1.1 Open-Box Extensibility

In *open-box extensible* systems, changes are performed in an invasive fashion; i.e. they are directly hacked into the original source code. Here are a few implications of this approach:



(a) Invasive change (b) Souce code duplication

Figure 1.1: Extensibility based on source code reuse.

- It requires that the source code is available and the source code license permits modifications.
- Changing source code (especially if it was written by someone else) is an error-prone and tedious activity, especially if the source code and the way a system works is not well understood.
- Since arbitrary changes can be performed, it is easy to break clients which were written for the original system with extensions that do not preserve backward-compatibility. Thus, open-box extensibility is inherently unsafe in its general, unrestricted form.

An illustration of open-box extensibility can be found in Figure 1.1a. Open-box extensibility is most relevant to a development team when fixing bugs, refactoring internal code, or producing the next version of their own software product.

In some cases, open-box extensibility may also be of interest to developers attempting to take advantage of existing *open-source software*. For deriving a variation of an already existing software product, the source code has to be copied and the changes have to be hacked into the copied sources. This principle is illustrated in Figure 1.1b.

While this approach makes it extremely easy to develop a new member of a software system family from scratch, it complicates maintenance significantly. The problem is that extension code and the original application code are mixed. When changes, e.g. bug fixes, get introduced into the original system, these also have to be somehow integrated into the extended system. This can be an extraordinary difficult and time-consuming task. And even if the system is modular enough that the actual code that was changed does not conflict, it is difficult to ensure that no assumed invariants of the new code are broken by the own changes.

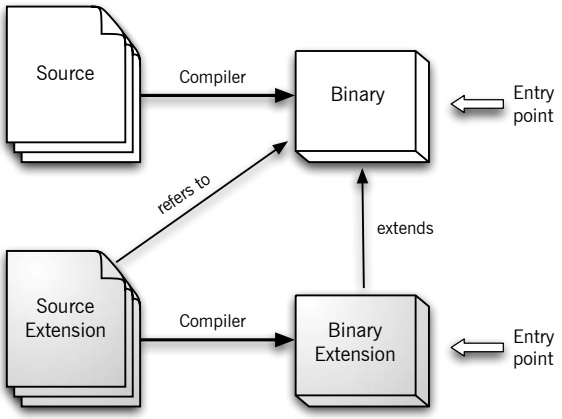


Figure 1.2: Extensibility based on binary reuse.

Therefore, it is not surprising to see that in practice, changes introduced by an open-box approach of that kind diverge most often.

1.3.1.2 Glass-Box Extensibility

Glass-box extensibility refers to the ways in which a software system may be extended when the source code is available, but may not be modified. Programmers that want to extend the system can view the code, but they have to separate their extensions from the original system in a way that does not affect the original system. Glass-box extensibility has several advantages over open-box extensibility:

- Since extensions and the original system are cleanly separated, it gets easier to understand and maintain extensions, as well as the original system. It is, in particular, more easy to combine new versions of the original system with extensions that were developed for the old one.
- Since glass-box extensibility is not directly based on source code modifications, it is less likely that the extension process introduces bugs in the original system or invalidates invariants established in the original system.

Object-oriented application frameworks provide an example for glass-box extensibility. These frameworks typically rely on features like inheritance and dynamic binding in order to achieve extensibility. Existing functionality is reused and extended by inheriting from framework base classes and overriding predefined hook methods. Such white-box frameworks are also called *architecture-driven frameworks*.

Figure 1.2 illustrates how glass-box extensibility can be used to derive an extended variant of a system. Both the extension's source code and the corresponding binary refer to the original binary, but they neither physically modify the original source nor the original binary. The binary of the extension provides a second entry-point into the system — which is, in fact, the entry-point into the extended

system. The old entry-point is still available and can be used to deploy the original system without the extension.

1.3.2 Gray-Boy Extensibility

Figure 1.2 shows that the glass-box approach does not necessarily rely on the availability of the original source code. Technically, only the original binary is required for developing extensions (assuming that the binary contains all relevant meta-data and the development platform supports late binding). Consequently, it is possible to extend systems in a glass-box fashion even if source code is not available. In this case, programmers of extensions could be given an alternative, more abstract documentation of the system's *specialization interface*. This interface lists all abstractions that are available for refinement and specifies how extensions interact with the original system. The rules for correctly extending a system can be described in form of *reuse contracts* [188].

This form of extensibility is called *gray-box extensibility*. It represents a compromise between a pure white-box and a pure black-box approach in the sense that it does not rely on the full exposure of the source code, but still provides internal details about the implementation and extension of a system.

1.3.3 Black-Box Extensibility

Black-box extensibility refers to the ways in which a software system may be extended when no internal details about a system's architecture and implementation are available. Black-box extensible systems are deployed and extended only by using their interface specification. This approach allows system manufacturers to fully encapsulate their systems and hide all implementation details.

As opposed to white-box extensibility, the extensibility mechanism is explicitly built into the system and part of the system's design. For designing such an extensibility mechanism, it is often necessary to anticipate all possible extension scenarios. For this reason, black-box approaches are often much more limited than white-box approaches. On the other hand, black-box extensible systems are generally easier to use and to extend since they require less knowledge about internal details of a system.

Typically, black-box extensions are made through system configuration applications or through the use of application-specific scripting languages. In the context of object-oriented application frameworks, black-box extensibility is mostly achieved by defining interfaces for components that can be plugged into the framework via object composition. Existing functionality is reused by defining components that conform to a particular interface and by integrating these components into the framework using design patterns like the *Strategy* pattern [74]. Figure 1.3 illustrates such a *plug-in* mechanism.

Black-box extensibility is most applicable to proprietary components and

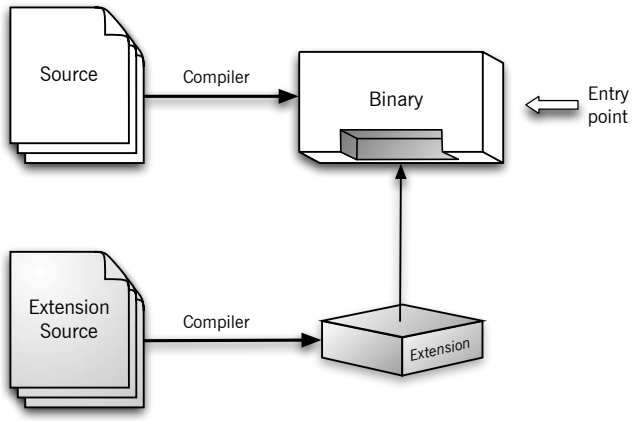


Figure 1.3: Extensibility based on plug-ins.

frameworks in which the business model of the original development team requires that the source code must not be published, but where external developers should still be given some degree of flexibility in customizing and extending the functionality of the software. Black-box extensible frameworks are often also called *data-driven frameworks*.

1.4 Extensibility Requirements

Software component technology emphasizes the construction of software from off-the-shelf components that are provided by a global software component industry consisting of independent component developers [195]. To facilitate the construction, deployment, and evolution of components in such a context, programming techniques and programming platforms have to satisfy the following requirements:

- Components are *generic* units of software that are implemented in a *modular* way with *explicit context dependencies* and with support for *separate compilation* [195].
- Facilities for composing independently developed components have to be *flexible* but also *safe* and should not require access to implementation details (*black-box component deployment*).
- Component composition and component evolution mechanisms have to *scale* well, since component-oriented programming is targeted towards *programming in the large* [56].
- The reuse of components in different contexts should imply the least possible need for *explicit adaptation code*.
- In support for a smooth software evolution process, components have to be *extensible* without the need to *anticipate* all possible future extensions.

- The extensibility and deployment mechanism has to be *safe* in the sense that it rules out erroneous ways to reuse a system [132].
- Component systems have to be extensible on the system level as well, allowing programmers to plug in alternative or additional components without the need for *rewiring* the whole system.
- Extensibility has to be *non-invasive*, avoiding that local changes have a global impact, and making it possible to derive different, independent extensions from a single component.
- Different extensions of a component have to be able to coexist, requiring an appropriate *versioning mechanism*.
- The extension of components must not require the availability of the full source code since this would violate the principle of *binary component reuse* (*gray-box component extension*).

In general, these requirements pose high demands on the implementation platform. The following list enumerates some features that programming languages have to support in order to meet these requirements [193].

- High-level abstractions for components and composition mechanisms,
- Modular encapsulation (as a high-level information hiding mechanism),
- Parametric polymorphism (to support genericity on the component level),
- Subtype polymorphism (to enable substitutability and variability of software components),
- Late binding and loading (as a basis for the independent deployment of components by third-parties),
- Static type safety (to make large-scale component engineering practical and safe), and
- Covariant refinement and specialization of abstractions (including types, to support extensibility and reuse).

Mainstream class-based object-oriented programming languages, which are today predominantly used for programming software components, do not live up to many of the requirements. In particular, they do not provide suitable abstractions for components which would allow programmers to make the architecture of a system and the relationships of a system's components explicit. Furthermore, type abstraction facilities are often quite restricted. Even though inheritance allows one to specialize classes covariantly, there is often no way to specialize composite structures, not even types. Furthermore, object-oriented programs are often full of hard links (static variables and methods, specific references to classes, etc.) making it difficult to evolve systems.

Only recently, this lack of support for component-oriented programming gave rise to research about how to best support component technology on the programming language level.

1.5 Programming Language Support

The principal means for writing software are programming languages. But not all programming languages are suitable for all programming tasks. It is important to choose a language that supports application specific requirements well.

Since component-oriented programming predominantly deals with the manufacturing of components and their deployment, a programming language supports this paradigm well only if it makes it easy to specify and compose components effectively and safely. Apart from such *abstraction* capabilities, programming languages offer formalisms for *composition*, *reuse*, and *verification*.

Abstraction. Component-oriented programming languages have explicit support for component abstractions; i.e. they provide *linguistic facilities* for defining software components on the programming language level. As a unit of composition with contractually specified interfaces and explicit context dependencies, components correspond, on the level of programming languages, very closely to *modules*, as introduced by imperative languages like Modula-2 [206] and Ada [196]. Nevertheless, modules in those classical module systems do not fully qualify as suitable abstractions for generic software components, since they hard-wire all dependencies by referring to specific cooperating modules. In more recent language designs — such as MzScheme [69], ComponentJ [182], ACOEL [187], and ArchJava [5] — language level component abstractions may abstract over their dependencies, making it possible to develop and deploy components independently of each other.

Composition and reuse. The independent reuse of software components in different contexts with different cooperating components is the most common form of reuse in the context of component-oriented software development. This form of reuse is enabled by software composition mechanisms like aggregation (object composition) or mixin-based inheritance (class composition), which allow programmers to compile ever-new composites of components and component instances.

This thesis focuses on extensibility, a specific form of reuse. On the language level, extensibility can be supported with mechanisms that allow one to create new versions of components or to specialize components for more specific tasks.

Verification. Static analysis tools approximate the runtime behavior of a program before it is executed. Compilers for programming languages that are

equipped with a static *type system* [43] perform exactly such an analysis at compile-time. They automatically prove the absence of certain program behaviors by classifying phrases according to the kinds of values they compute. Furthermore, type systems are also used to enforce modularity properties and to protect the integrity of user-defined abstractions.

As static analysis tools, type systems have to be conservative. Even though they can be used to categorically prove the absence of bad program behaviors or illegal interactions, they are unsuitable for proving their presence. Consequently, type systems sometimes reject programs that actually behave well at runtime.

Nevertheless, type systems are extremely helpful for developing correct and reliable software [42]. They are significant for the following reasons [164]:

- *Error Detection:*
  Static type checking allows detection of some programming errors already at compile-time. In addition to the *early detection of errors*, it is often possible to *pinpoint errors more accurately* during type checking than at runtime when their effects may not become visible until some time after the actual moment where things begin to go wrong. Type checkers can also be used as *maintenance tools*, for example, when abstractions are changed and all clients have to be updated accordingly.

- *Abstraction:*
  Type systems support the software development process by enforcing disciplined programming. In the context of large-scale software composition, type systems form the backbone of module systems which are used to assemble the components of large systems. Module systems often enforce a separation of component interface specifications from concrete implementations. While types are typically used to specify the signature of module members in interface descriptions, module interfaces themselves can be regarded as types of modules. Structuring large systems in terms of modules with clear interfaces leads to a more abstract style of design where interfaces are designed and discussed independently from their eventual implementations. More abstract thinking about interfaces, in turn, generally leads to a better design.

- *Language Safety:*
  Type systems do not only make it possible to setup abstractions, they can also enforce their integrity and their correct usage. In particular, they can guarantee that abstractions and established invariants are not broken by clients. Without static type systems, the *safety* of a programming language has to be provided solely by dynamic checks.

- *Documentation:*
  Type declarations constitute a form of documentation giving useful hints

about the functionality of an abstraction and its usage. In contrast to documentation in a natural or semi-formal language, this form of documentation can not become outdated as the program evolves, since with every run of the compiler, the type checker will verify its conformance with the implementation. The role of types as machine-checkable documentation is particularly important at the level of module interface descriptions, but can also be useful when reading programs.

- *Efficiency:*
  Better understanding about what kind of values are handled by a program term helps compilers to produce more specialized code with better runtime performance. In particular, safe languages allow elimination of many dynamic checks, which would otherwise be needed to guarantee safety at least at runtime.

Since each of these points is extraordinarily important for the development of software components, static typing is indispensable for component-oriented programming. For this reason, the following chapters have a strong focus on statically typed programming languages in which the manufacturing, the composition, and the evolution of software components is subject to a static type system.

1.6 Component Engineering Approaches

Today, component-oriented programming is mostly carried out in the context of mainstream object-oriented programming languages. Component programming practice relies on component models that are based on *component frameworks* and *meta-programming* technology.

1.6.1 Frameworks

Component frameworks are software entities that establish an environment in which components that conform to a certain standard can be instantiated and in which those instances can be plugged into the system. A component framework defines rules and interfaces that govern the interaction of component instances that are plugged into the framework. It also enforces architectural principles by forcing component instances to perform certain tasks via mechanisms that are under the control of the framework itself. Some component frameworks are higher-order; they apply this concept hierarchically such that environments established by a component framework are themselves component instances that can be plugged into other component frameworks.

The various industrial component models like the Corba Component Model [83], COM [172], JavaBeans [190], and Enterprise Java Beans [191], all have similar design goals and provide, to some degree, similar functionality,

but they follow quite different design philosophies. What they have in common is that the implementation of these models is based on technology that goes beyond the infrastructure provided by mainstream object-oriented languages. Such models typically provide a class framework for modeling components and component interactions together with an informally specified set of implementation rules. Component instances are often composed dynamically using meta-programming technology like *introspection* and *reflection*. Some component models, like EJB, even require that contracts are specified completely outside of the programming language, typically using an XML-based approach.

These implementation issues all defeat an effective usage of a programming language as a means to guarantee statically the integrity of component systems and their conformance to the implementation standards required by the framework. In particular, there is often not even a way to statically check if the various software artifacts specified in different formalisms are consistent and hence "fit" to each other.

1.6.2 Extensibility

In practice, extensibility is often either achieved by relying on *design patterns* or by applying *meta-programming*.

For design pattern-based approaches it is necessary to plan extensibility ahead by rigorously deploying suitable patterns that are typically derived from the *AbstractFactory* pattern [74]. Frameworks in which extensibility is mainly achieved by design patterns that are based on object composition and forwarding are called *data-driven frameworks*, frameworks where extensions are derived by inheritance and overriding are called *architecture-driven frameworks*.

As opposed to design patterns, meta-programming technology provides ways to extend systems without necessarily planning extensibility ahead. Meta-programming has many different variations. As an overview, we will only mention a few approaches here.

Generative programming [55] aims at transformational approaches to the construction of software. Approaches based on generative programming work best in two areas: They help to produce a larger number of similar components and they can be used to enhance composed systems. In a world of deployable components, both cases must be handled with care to ensure that component boundaries are respected and possible interferences with cooperating components are avoided. No matter if generative programming techniques are applied to source code or some intermediate bytecodes, they most often do not come with any safety guarantees.

Aspect-oriented programming techniques [103] make it possible to modularize crosscutting aspects of a system, facilitating the separation of different concerns. A system consisting of various "aspect slices" is assembled by an *aspect weaver* which merges the fragments into a whole. In the context of aspect-

oriented programming, a system is typically extended by adding new aspects. But with the current aspect-oriented programming technology it is relatively difficult to set up strong component boundaries which enforce encapsulation and which specify context dependencies explicitly. As a consequence, aspects often do not qualify as generic abstractions which can be deployed in arbitrary contexts, even though they are modular units of software [104].

Systems that have to be available constantly and cannot be terminated for introducing extensions are typically evolved using *dynamic meta-programming* technology like *reflection*. Often such approaches are based on special *middleware* which explicitly supports the dynamic evolution of systems.

Extensibility mechanisms based on meta-programming allow systems to be extended in a flexible and non-invasive manner, but they rarely give safety or consistency guarantees at compile-time. Since many static meta-programming approaches are mostly source code-based, they are not very suitable for today's component technology practice which stresses the binary deployment of software components. Nevertheless, with a stronger focus on static safety and a tighter language integration, meta-programming technologies like aspect-oriented programming or *multi-stage programming* [197] are likely to provide helpful tools for developing extensible software in the future.

1.7 Overview

1.7.1 Scope

This thesis investigates how the development of extensible, component-based software can be facilitated with programming languages that support extensible component abstractions explicitly. The work was driven by the belief that only on the programing language level it is possible to find mechanisms that meet all the requirements listed in Section 1.4. The work puts special emphasis on safety and consistency, promoting solutions in strongly typed programming languages.

1.7.2 Contributions and Outline

Chapter 2. The main part of the dissertation is split up into five chapters. The role of Chapter 2 is two-fold:

- It explains the notion of software components and component composition in a formal way in the context of a class-based object-oriented language. The theoretical model captures typical principles like explicit context dependencies, late composition, and strong encapsulation. It motivates how types help to manufacture and compose components safely.

- It illustrates the vision of truly extensible software components by introducing a small set of composition primitives for dynamically building, composing, and extending software components in a concise and safe manner.

Chapter 3. Chapter 3 applies the essential ideas of the theoretical model in the design of the programming language KERIS. KERIS extends the programming language JAVA with an expressive system of extensible modules. The main contributions of this work are:

- A practical module system that combines the benefits of classical module systems for imperative languages with the advantages of modern component-oriented formalisms. In particular, modules are reusable, generic software components that can be linked with different cooperating modules without the need for resolving context dependencies by hand (*wiring inference*).
- A module composition scheme based on aggregation that makes the static architecture of a system explicit, and
- A type-safe mechanism for extending atomic modules as well as fully linked systems statically by replacing selected subsystems with compatible versions without the need to re-link the full system. The extensibility mechanism is non-invasive; i.e. it preserves the original version and does not require access to source code.

The overall design of the language was guided by the aim to develop a pragmatic, implementable, and conservative extension of JAVA which supports software development according to the *open/closed principle:* Systems written in KERIS are closed in the sense that they can be executed, but they are open for unanticipated extensions that add, refine or replace modules or whole subsystems without planning extensibility ahead.

Chapter 3 explains the design of KERIS, presents application scenarios, shows how the language is implemented, and investigates how efficient this implementation is. Furthermore, it discusses the module system of KERIS in relation to other module systems. This discussion is based on a general module system taxonomy.

Chapter 4. Chapter 4 presents a case study for evaluating the module system of KERIS. It focuses on the development of extensible compilers. Regarding extensibility, compilers are interesting for various reasons:

- Programming languages are often extended by different people for different, often domain-specific purposes. Implementing compilers for such extended languages from scratch is a tedious and quite redundant process. If compilers would be extensible systems, the implementation of specialized programming languages would be easier and more effective.

- Compilers are language processors; extending such systems requires that language constructs as well as operations upon language constructs are extensible. It is a well-known technical problem to facilitate extensions in both dimensions.
- Compilers are relatively complex systems with many recursively dependent datatypes and components. It is a challenge to extend such tightly interconnected systems consistently.

Chapter 4 compares an extensible Java compiler implemented using mainstream object-oriented language features with one that was written in Keris. It shows how extensible modules can be used in practice to develop extensible systems safely and efficiently.

Chapter 5. The last chapter concludes the thesis with a presentation of related work, a summary of the problems and solutions discussed in the dissertation, as well as some notes about possible future work.

Chapter 2

A Formal Model for Extensible Software Components

This chapter introduces basic concepts of software component technology and component-oriented programming by discussing details of a simple but expressive theoretical component model. This component model is designed to support the implementation and evolution of lightweight, extensible components in object-oriented programming languages. The model is expressed as a small component-oriented programming language featuring dynamic component manufacturing, composition, and extension in a type-safe way through a minimal set of component specialization primitives. Furthermore, there is support for principles like explicit context dependencies, late composition, unanticipated extensibility, and strong encapsulation of component services. In contrast to other approaches, the services which components provide and require do not have to be linked explicitly. Instead, components are composed using high-level composition operators which implicitly link services by inferring the wiring. Finally, a formalization of the component model is presented which extends *Featherweight Java* [95, 164], a typed core calculus modeling the essential features of JAVA.

The remainder of this chapter is organized as follows. Section 2.1 discusses the implications of component-based software development, emphasizing, in particular, the importance of software adaptability, extensibility, and software evolution in general. Section 2.2 introduces the component model step by step, presenting the various component specialization primitives and motivating their application. A formalization of the model is described in Section 2.3 in form of a core component calculus. We present a type system and prove that this system is sound with respect to the given operational semantics.

2.1 Motivation

Before presenting details we motivate specific design principles of the component model. The main features of the model include:

- Components are first-class core language abstractions,
- Composition operators enable coarse-grained component composition,
- Components can be manufactured and composed dynamically,
- Components are extensible, promoting advanced component reuse, adaptability, and evolution.

Furthermore, the model expresses many principles common among component-oriented formalisms, like explicit context dependencies (external linking), cyclic component linking, and strong encapsulation of component services. Component manufacturing, composition, and specialization are type-safe. The type system supports subtype polymorphism for components and component instances.

2.1.1 Language Integration

Currently, component-based programming relies on mainstream object-oriented programming languages. Object-oriented languages seem to promote component-based programming well: They support encapsulation of state and behavior, inheritance and overriding enable extensibility, and subtype polymorphism and late binding make it possible to flexibly reuse objects and classes.

Unfortunately, object-oriented techniques alone are not powerful enough to provide flexible and type-safe solutions for component composition and evolution. Therefore, industrial component models like CORBA [83], COM [172], and JavaBeans [190] rely on additional concepts, namely component frameworks and meta-programming. They provide a class framework for modeling components and component interactions together with an informal set of implementation rules. Components are composed using meta-programming technology, e.g. reflection. This ad-hoc approach yields a dynamic and flexible composition mechanism, but often does not guarantee static type security. Furthermore, the degree of extensibility depends on the framework or the meta-programming tools. In general, it has to be planned ahead, for instance by using suitable design patterns typically derived from the *AbstractFactory* pattern [74]. This lack of support for unanticipated extensibility hinders a smooth evolution process substantially.

In [5], Aldrich and Chambers point out another important issue. They observe that in general, implementation languages are only loosely coupled to architectural descriptions. As a consequence, specifications of software architectures [159, 184] formally expressed in architecture description languages [130] are often quite different from the actual object-oriented implementations. This makes it difficult to trace architectural properties in the implementation, on

which basis it would be possible to verify that an implementation is consistent with the corresponding architectural description.

Related to this issue is the critique of Achermann *et.al.* that object-oriented source code exposes class hierarchies but not object interactions [2]. This lack of explicit architecture makes it difficult to adapt an application to new requirements since the code that plugs objects together is distributed among the objects themselves and therefore difficult to overlook.

For these reasons, various proposals have recently been put forward to integrate concepts from architecture description languages into object-oriented programming languages [182, 187, 5]. These so-called component-oriented programming languages offer linguistic facilities for programming software components, for defining component interactions, and for composing software from components. Their promise is to do that in an effective and type-safe way, ruling out illegal interaction patterns.

2.1.2 Coarse-Grained Composition

Existing proposals for component abstractions on the programming language level like COMPONENTJ [182] and ARCHJAVA [5] directly adopt concepts and principles of architecture description languages. They provide constructs for manufacturing components with *required* and *provided* services. A service associates a port name with a type. Components are composed by linking ports with explicit plug instructions. The type system ensures that all ports are linked and that links are established only between compatible ports or service providers.

This approach does not scale, since for linking a component with n services, n explicit plug instructions have to be issued specifying the wiring of the component. For large-scale components with a lot of services involved, linking components in such a way is tedious and error-prone. Furthermore, the sequence of plug instructions rather obscures the architecture of the system instead of making it explicit. Therefore, McDirmid, Flatt, and Hsieh argue that component systems should offer the possibility to connect many services at once [129].

We address this requirement by simplifying the interface of components and by providing means to infer the wiring of components. Components can be composed with simple operators and without explicitly plugging ports. We also support incremental linking; i.e. we allow that components get only partially linked. For instance, components can be sent around in a distributed system and only the services available at a specific location get linked until we have a fully linked component that finally can be instantiated.

2.1.3 Dynamic Manufacturing and Composition

Software component technology distinguishes two main tasks: component manufacturing and component composition. These two tasks are often presented as

separate steps being performed one after the other. But in practice, both tasks coincide when new components are built by composing other components. This form of component manufacturing is called *hierarchical component composition*.

Often it is assumed that component manufacturing is done statically before component composition takes place. Component composition itself cannot always be performed statically, since there are cases where the concrete components are only known at runtime. For this reason, component-based systems are often required to support some form of *dynamic linking*.

This observation implies that we also have to be able to manufacture software components dynamically, since component linking and manufacturing coincide in hierarchical component compositions. Thus, it makes no sense to assume that both manufacturing and composition are atomic tasks that are performed consecutively. In highly dynamic systems, component manufacturing and composition is rather an interleaved process in which components are created and linked incrementally.

2.1.4 Extensibility

When using components from external vendors, it is quite unlikely that the interfaces of these third-party components fit to the required interfaces off-the-shelf. Therefore, it is often necessary to adapt components before they can be deployed in a particular system [90, 150]. As Section 2.1.3 pointed out already, there are applications where components are only supplied at runtime. In this case it is indispensable that components can even be adapted dynamically on-the-fly.

In a prototype-based system, new entities can only be created by cloning an existing entity and by modifying the cloned entity afterwards. Our component model is *prototype-based* in this sense: new components can only be created by specializing an already existing component. As a consequence, we can derive two different components from a single base component. By doing this, we factor out potentially reusable pieces, avoiding duplicated programming effort. In addition, this technique supports software evolution. Software evolution includes the maintenance and extension of component features and interfaces. Supporting software evolution is important, since components and component systems are architectural building blocks and as such subject to continuous changes.

Extensibility is not only required for smoothly evolving single software components. It is even more desired for enabling the development of families of software applications and product-lines in general. Traditionally, components are static black-boxes emphasizing encapsulation over extensibility. Features can be added to components only by creating new components which forward all existing services to the old versions in addition to the new services they provide. This is a cumbersome and error-prone procedure that duplicates programming efforts and complicates maintenance.

2.2 Prototype-Based Components

We now introduce step by step our component abstractions in the context of a small, statically typed, object-oriented Java-like base language. Our component model relies on a nominal type system [164] of the base language. In nominal type systems, two types with the same structure but a different name are considered to be different, as opposed to structural type systems that match the structure and not the name. The model does not require other base language features like inheritance or even classes, even though we present it here in a class-based context. Therefore it should be straightforward to add similar component abstractions to other object-oriented languages with nominal object types.

2.2.1 Components and Component Instances

In our model, a component is a unit of computation that can be accessed through a well-defined interface. A component is a first-class citizen. Its interface specifies the services it provides to allow other components to interact with it. The interface also specifies the services a component requires from other components to be able to provide the own services.

The component model is prototype-based; i.e. the only way to create a new component is by specializing an already existing prototypical component. For bootstrapping purposes, there is a single predefined component that does not provide or require any services. This empty component is denoted by the keyword `component`.

We strictly distinguish between components and component instances. A component describes a template for possibly multiple component instances. It is the component instances that provide the actual services. Services are described by object types, e.g. types defined by classes or interfaces. Objects act as service providers. They usually get created at component instantiation time. Therefore, components can be seen as organizational units with well-defined interfaces that structure object interdependencies. Components have neither a unique identity, nor an observable state. They come to life through objects at the time they get instantiated.

In the remainder of this section we introduce prototype-based components by example. We derive some simple software components that could be used, for instance, in online retail stores to manage stock and clients.

2.2.2 Service Provision

We start by manufacturing a software component that provides access to a customer database. We want every customer to have a unique client number. A service that maps customer names to client numbers could be described by the following interface definition:

```
interface CustomerIDs {
  int lookupId(String name);
}
```

The CustomerIDs interface consists of a single method lookupId. Given a customer's name, this method tries to find the corresponding client number. If there is no client number yet for this customer, a new number will be issued and returned by lookupId. Imagine we have the following implementation of the CustomerIDs interface:

```
class MyCustomerIDs implements CustomerIDs {
  MyCustomerIDs() { ... }
  int lookupId(String name) { ... }
  ...
}
```

With this implementation we are able to manufacture a software component that provides a CustomerIDs service. Since we can only create new components by specializing existing ones, we have to take the empty component as a prototype and specialize it such that it provides a CustomerIDs service. In our calculus, this is done with the provides primitive:

```
c0 = component
     provides CustomerIDs as This with new MyCustomerIDs();
```

The clause d provides $\overline{C}$ as x with e returns a new component that specializes component d by providing some possibly new services $\overline{C}$. These services are implemented by an object specified with expression e. Note that we are extending a component here. Therefore, expression e only gets evaluated at component instantiation time. x is a variable that gets bound to the own component instance. In object-oriented languages this self reference corresponds to variable this or self referring to the own object. Only expression e is in the scope of x. Typically, expression e refers to other services of the own component instance via x.

We use a graphical notation to illustrate the structure of components. Figure 2.1 gives an overview. Here, a component is represented by a box. The gray part corresponds to the prototype of the component, the white part defines the specialization. In our graphical notation, services are symbolized by diamonds. Objects are black dots. An arrow from a service to an object expresses that this object implements the service. We also have outlined arrows that depict service dependencies. These dependencies are found by inspecting the code; they are not explicit in the calculus. If an object refers to other services, for instance via the self reference, then every such dependency is specified with an outlined arrow. Figure 2.2 illustrates the structure of the previously defined component c0.

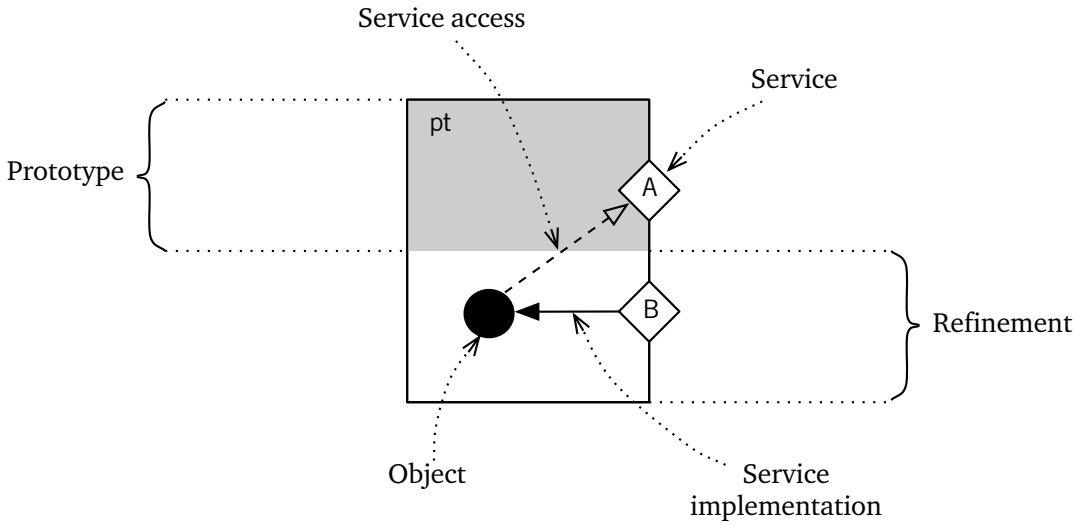


Figure 2.1: Schematic component notation.

2.2.3 Component Instantiation

We already pointed out that components have to be instantiated before services can be accessed. In our component calculus, a component gets instantiated with the `new` primitive.

```
i0 = new c0;
```

The services of a component instance like `i0` get accessed via the service selection operator ::. The expression $e :: C$ selects a service C from component instance e. C is a type name that identifies a service and at the same time describes the interface of the service. Other component models refer to services via named ports. In these models it is possible to have two distinct ports with the same interface type but different port names. In programming languages with nominal type systems like JAVA [82] or C# [86], types do not only define structural object properties like available methods or fields. They also stand for semantic specifications [38], and as such, they are well-suited for specifying roles. In those type systems it is possible to have two distinct types with the same interface description but different type names. Therefore, it is no restriction to describe a service only by its type without having a port name in addition. This simplifies the definition of components and the service access in general significantly. It also acts as a standardization of port names. One only has to know a service's type in order to access it from a component instance. It is not necessary to lookup the port name in the component specification. We will see later in Section 2.2.7 that this standardization of component port names has another advantage: it promotes automatic composition mechanisms. Of course, in the few cases where two ports

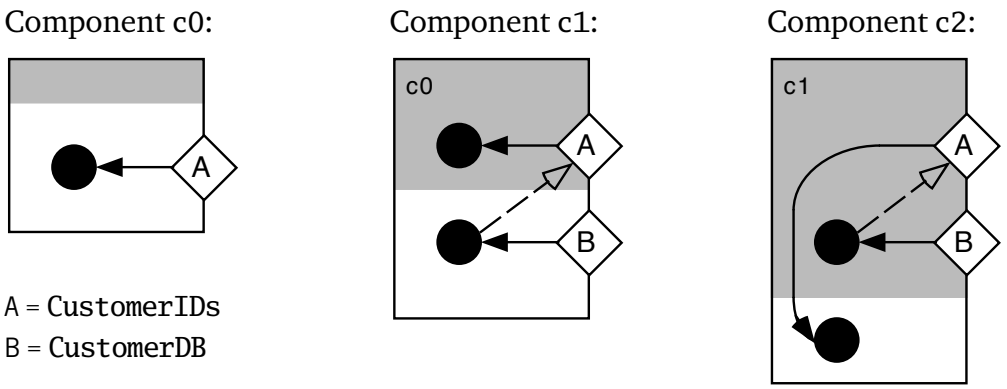


Figure 2.2: Component evolution.

could share a type, we have to create new type names and in the worst case use wrappers to adapt existing objects.

Here is an example demonstrating the usage of the component `i0`. In this example we call the `lookupId` method of the `CustomerIDs` service provided by component instance `i0`. The service selection operator :: and the . operator are both left-associative and both operators have the same precedence.

```
i0::CustomerIDs.lookupId("John_Smith");
```

2.2.4 Component Specialization

Now imagine the requirements for our customer administration component `c0` are changing and we also need the capability to store customer names and addresses. We can describe this new database service with the following interface:

```
interface CustomerDB {
  void enter(String name, String address);
  String lookupName(int id);
  String lookupAddr(int id);
}
```

Method `enter` stores a new address in the database. Whenever a new customer is entered, a new client number will automatically be assigned to this new customer. The methods `lookupName` and `lookupAddr` find a name or address for a given client number. The following class implements `CustomerDB`. It depends on a component instance that provides a `CustomerIDs` service. This component instance is passed as a parameter to the constructor. Following [182], we use the notation $[S_1, ..., S_n]$ to specify the type for component instances supporting at least the services S_1 to S_n.

```
class MyCustomerDB implements CustomerDB {
  [CustomerIDs] This;
  MyCustomerDB([CustomerIDs] This) {
    this.This = This;
  }
  ...
  This::CustomerIDs.lookupId(name)
  ...
}
```

We already mentioned that prototype-based components offer a smooth component evolution mechanism. For creating an extended version of a component, we just have to interpret the old component as a prototype. In our example, the new specialized component evolves out of the old one simply by an application of the provides primitive. The following code specializes component c0 by additionally providing the service CustomerDB.

```
c1 = c0 provides CustomerDB as This with new MyCustomerDB(This);
```

The provides primitive can also be used to specialize a component by defining a new service implementation for an already provided service. In this case we *override* the old implementation. Here is the definition of component c2 that specializes c1 by using, for instance, a more efficient client numbering service.

```
c2 = c1 provides CustomerIDs as This with new EfficientCustomerIDs();
```

The service implementation for CustomerDB, specified already in the prototype of c2, now automatically refers to this new numbering service implementation. A graphical illustration of components c1 and c2 can be found in Figure 2.2.

2.2.5 Service Forwarding

Until now, we are only able to develop new components by adding new services or by overriding existing service implementations of a prototypical component. Every service we add gets exported automatically; i.e. it can be accessed from outside the component. This "white-box approach" is necessary to keep the component extensible, because it allows one to override service implementations and to add new service implementations that refer to already existing services. But often we do not want to publish internally used services. Being able to hide internal interfaces is an important feature of component-oriented programming. Our component calculus supports this form of encapsulation with the component projection operator forwards. The clause d forwards $\overline{C}$ as x to e extends component prototype d with the services $\overline{C}$. The new component forwards accesses of these services to the component instance e. Expression e can refer to other

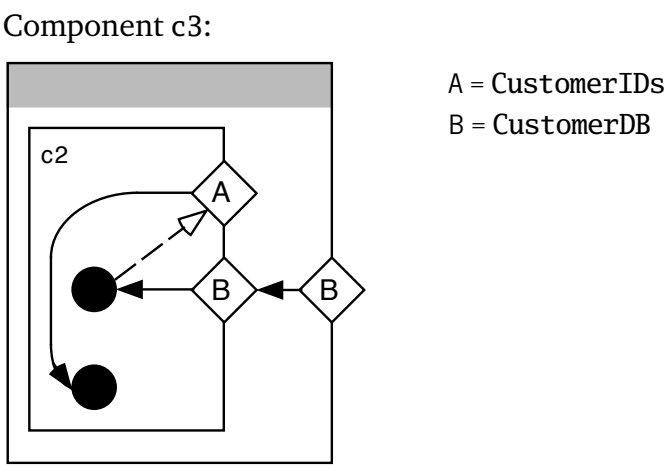


Figure 2.3: Service forwarding.

services of the own component instance via the self reference *x*. This primitive is primarily used for composing components hierarchically. In the following example it is specifically used to hide services and service interconnections. Thus, it turns a "white-box" into a "black-box" by wrapping the original component.

```
c3 = component
     forwards CustomerDB as This to new c2;
```

In this example we create a new component *c*3 which only provides a single service CustomerDB by forwarding calls to a component instance of c2. Thus, we hide the CustomerIDs service of component c2. We say, an instance of c2 is nested inside every instance of component c3. We call the hidden CustomerIDs service an internal service of component c3. An illustration of c3 instances can be found in Figure 2.3. Here, the instance of component c2 which is contained in c3 is depicted by a nested box. Service implementations are now arrows pointing from external services to internal services of nested component instances.

2.2.6 Service Abstraction

The previous sections showed how to evolve a component by incrementally adding new services either by a new service implementation or by forwarding services to a nested component instance. In both cases new services and service implementations were introduced at the same time. This approach does not cover the development of components that depend on services provided by other components. Furthermore, it does not even allow programmers to define two services where service implementations depend mutually on each other, because services get introduced linearly, one after the other.

We tackle both problems with a service abstraction facility. Before going into detail, we proceed by manufacturing a new component for handling orders of a shop. The service for placing orders is described by the following interface:

```
interface OrderDB {
  void order(int id, String article, int num);
}
```

With method order, new orders can be placed. Orders consists of a client number, an article descriptor and the number of items to deliver. If possible, this method tries to execute the order immediately. Therefore it needs access to a stock database service specified by the following interface:

```
interface StockDB {
  void enter(String article, int num);
  void remove(String article, int num);
  int available(String article);
}
```

Method order checks if the articles are available. If this is the case, it removes them from the stock database and sends the articles to the customer's address. Therefore, service implementations of OrderDB like MyOrderDB also need access to the CustomerDB service. Thus, the constructor of the following class expects a component instance providing StockDB and CustomerDB services.

```
class MyOrderDB implements OrderDB {
  [StockDB, CustomerDB] This;
  MyOrderDB([StockDB, CustomerDB] This) {
    this.This = This;
  }
  ...
}
```

Since we do not want our order system component to already commit to a specific service implementation for the StockDB and the CustomerDB service, we have to factor out these two services. In order to make use of the component later, we then either have to provide the missing service implementations from outside at composition time, or we further specialize the component and provide service implementations from inside the component.

In our component calculus, services are factored out with the service abstraction primitive `requires`. The `requires` primitive allows one to define services that are required for implementing other services without the need for specifying a concrete service implementation. We make use of this abstraction facility in the following implementation of component d0 which requires two services CustomerDB and StockDB and provides a OrderDB service. Figure 2.4 contains an illustration of component d0.

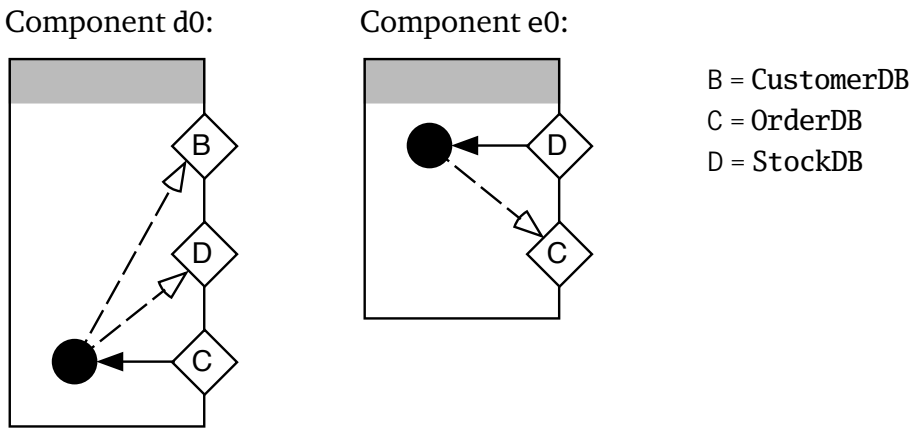


Figure 2.4: Service abstraction.

```
d0 = component
     requires CustomerDB
     requires StockDB
     provides OrderDB as This with new MyOrderDB(This);
```

The expression d `requires` C takes a prototypical component d and returns a specialized version with a service C that has to be provided before the component can be instantiated. Other service implementations can refer to this service, even though there is no implementation known yet. This is why in the example above, self reference `This` has type `[CustomerDB, StockDB, OrderDB]` and thus is a legal parameter for the constructor of `MyOrderDB`. Components have a type of the form $(R_1, \ldots, R_n \Rightarrow P_1, \ldots, P_m)$ where R_1 to R_n are services required by the component, and P_1 to P_m are the provided services. Consequently, the type of component `d0` is `(CustomerDB, StockDB ⇒ OrderDB)`. As already mentioned before, component `d0` cannot be instantiated, since not all service provisions are resolved yet. We first have to derive a new component that specifies implementations for all required services before we can actually create component instances.

We continue in our example by defining a new component `e0` that provides an implementation for a `StockDB` service.

```
e0 = component
     requires OrderDB
     provides StockDB as This with new MyStockDB(This);
```

The implementation of service `StockDB` makes use of an externally supplied `OrderDB` service. This is, because in cases where new stock arrives and orders are still pending, it would trigger the process of sending out the articles. The type of component `e0` is `(OrderDB ⇒ StockDB)`.

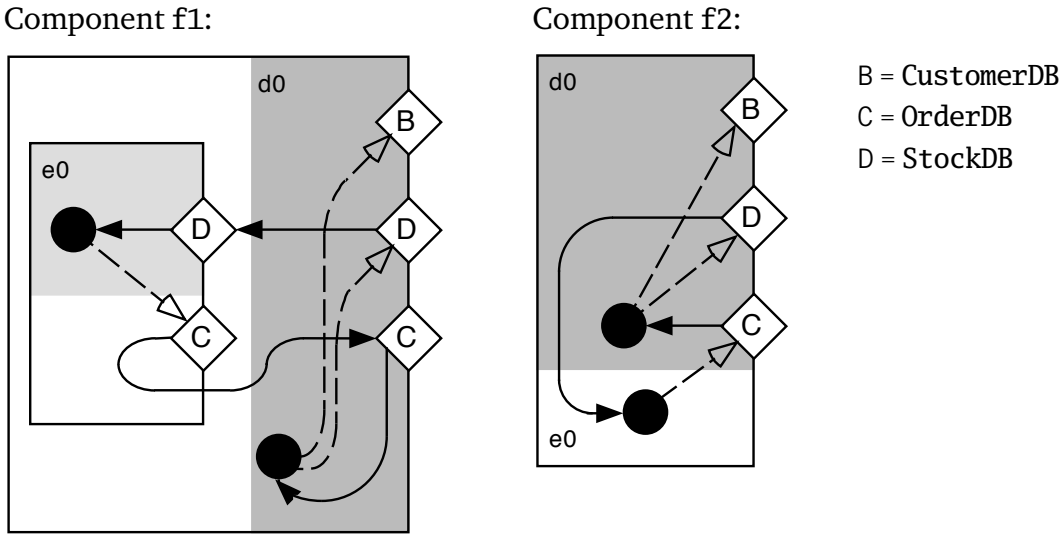


Figure 2.5: Component composition.

2.2.7 Composition of Components

In the previous section we defined two components d0 and e0 that mutually refer to each other; i.e. the service provided by one component is required by the other one. We would now like to link these two components together yielding a component which only requires a CustomerDB service and provides both a OrderDB and a StockDB service. The simplest way to achieve this is to specialize component d0 with an implementation for service StockDB. This service is provided by a specialized version of e0 that refers back to the OrderDB service provided by the enclosing d0 prototype.

```
f0 = d0 provides StockDB as This with
        (new (e0 provides OrderDB as Me with This::OrderDB))::StockDB
```

This technique does not work for components where more than two services depend mutual recursively on each other. For such cases we have to use the forwards primitive in order to link the components together. A graphical illustration of the resulting component f1 can be found in Figure 2.5.

```
f1 = d0 forwards StockDB as This to
        new (e0 provides OrderDB as Me with This::OrderDB)
```

The previously discussed composition schemes use service forwarding where the nested component instance refers back to services provided by the enclosing component being defined. Our component calculus offers an alternative to this rather complicated composition pattern. With the mixin operator it is possible to create a new component by mixing in the services provided by another component. The expression *e* mixin *d* specializes the prototypical component *e* with

component d; i.e. e gets specialized by including all the services provided by component d. Services that are already present in e are automatically overridden by the corresponding services of d. This operation identifies the self references of both components e and d by binding it to the resulting merged component. The resulting component requires services that are either required by e or d and that are not provided by any of the two components. It provides all the services that are provided by either e or d. Thus, the following expression yields a component `f2` of type `(CustomerDB ⇒ OrderDB, StockDB)`.

```
f2 = d0 mixin e0
```

When using such a mixin-based composition scheme, one has to be aware that for the expression above, all services `e0` provides get mixed in, no matter what static type `e0` has in this context. Thus, we might accidentally override services provided by component `d0`. Sometimes this is desired, for instance, when we want to express that `e0` has got the more recent or more trustworthy service implementations than `d0`. For cases where we want to define explicitly what services to override, we have to use a forwarding-based composition scheme instead. For instance, we could write `d0 forwards StockDB as This to new (e0 forwards OrderDB as Me to This)`.

All three components defined in this section are equivalent in the sense that they provide and require the same services and that services are implemented by the same objects. Though, Figure 2.5 reveals that the internal structure of components manufactured using the forwarding and the mixin technique are quite different. This is an indication that they possibly behave differently when it comes to specialize both components. In the given example, this is not the case. But one might imagine a bigger nested component instance where overriding a service of the enclosing component does not have any effect on the formerly forwarded service of the nested component, while it would have an effect on the mixin-based approach.

We finish this section by manufacturing a component that permits access to customer related services only; i.e. `CustomerDB` and `OrderDB`. We do this by first linking together the customer management component `c2` and the stock management component `f2`. The linked component `c2 mixin f2` provides all the various services introduced in this section. Since we want to restrict the access to customer related services, we have to project the resulting component to a new component `g0` offering only the desired services.

```
g0 = component
     forwards CustomerDB, OrderDB as This to new (c2 mixin f2)
```

`g0` has type `( ⇒ CustomerDB, OrderDB)`; thus, it is possible to instantiate this component. The structure of an instance of our final component `g0` is presented in

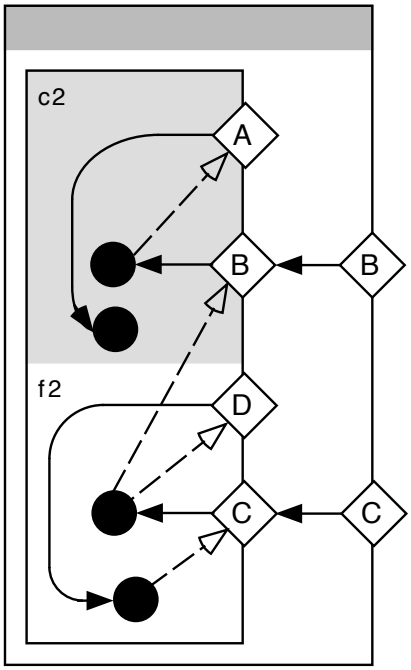


Figure 2.6: The final component g0.

Figure 2.6. Leaving out some intermediate steps, we could have composed g0 out of three essential components: c2 which administers clients, d0 which handles orders, and e0 which manages the stock.

```
g0 = component
     forwards CustomerDB, OrderDB as This to new (c2 mixin d0 mixin e0)
```

This short expression demonstrates how concise component manufacturing and linking is in the presented model if high-level composition operators like forwards and mixin are used. Furthermore the expression indicates how components are typically deployed. The sub-expression `c2 mixin d0 mixin e0`[1] first links components c2, d0, and e0, yielding a single extensible component. This component exposes internal interfaces. We might want that for instance to use this component as a basis for further specializations. But before instantiating (or even selling) it, the internals should be hidden by wrapping the component in a black-box only offering specific functionality with restricted support for extensibility. In the example above, this is achieved using the component projection primitive forwards.

[1] Please note that the mixin operator is associative.

2.3 Component Calculus

In this section we present a formalization of our prototype-based component model for a functional subset of Java. Our calculus is built on top of Featherweight Java (*FJ*) [95]. We omit type casts from the original calculus since type casts are irrelevant for our application and complicate the formal treatment unnecessarily.

2.3.1 Syntax

The syntax of the calculus is presented in Figure 2.7. Like in *FJ*, a program consists of a collection of class declarations plus an expression to be evaluated. The syntax of classes, constructors, and methods is identical to *FJ*. We only extend the set of expressions with the primitives introduced in Section 2.2. In particular,

Program

P	$=$	$\overline{L}\ ;\ e$	*program*

Class

L	$=$	$\texttt{class}\ C\ \texttt{extends}\ C\ \{\ \overline{T}\ \overline{f}\ ;\ K\ ;\ \overline{M}\ \}$	*class declaration*

Constructor

K	$=$	$C(\overline{T}\ \overline{f})\ \{\ \texttt{super}(\overline{f});\ \texttt{this}.\overline{f} = \overline{f};\ \}$	*constructor declaration*

Method

M	$=$	$T\ m(\overline{T}\ \overline{x})\ \{\ \texttt{return}\ e;\ \}$	*method declaration*

Expressions

e	$=$	x	*variable*
	$\mid$	$e.f$	*field selection*
	$\mid$	$e.m(\overline{e})$	*method invocation*
	$\mid$	$\texttt{new}\ C(\overline{e})$	*object creation*
	$\mid$	$\texttt{component}$	*empty component*
	$\mid$	$e\ \texttt{requires}\ C$	*service abstraction*
	$\mid$	$e\ \texttt{provides}\ C\ \texttt{as}\ x\ \texttt{with}\ e$	*service implementation*
	$\mid$	$e\ \texttt{forwards}\ \overline{C}\ \texttt{as}\ x\ \texttt{to}\ e$	*component projection*
	$\mid$	$e\ \texttt{mixin}\ e$	*component mixin*
	$\mid$	$\texttt{new}\ e$	*component instantiation*
	$\mid$	$e :: C$	*service selection*

Types

T	$=$	C	*object type*
	$\mid$	$\overline{C}\ \Rightarrow\ \overline{C}$	*component type*
	$\mid$	$[\overline{C}]$	*component instance type*

Figure 2.7: Syntax.

we add an empty component, a service abstraction and implementation primitive, a component projection primitive as well as a component mixin operator. In addition, we have a construct for instantiating components and a service selection operator for accessing services from a component instance. In the calculus, a service is characterized by a class name.

In contrast to the presentation in Section 2.2.2, the calculus only supports a `provides` primitive that introduces a single service. This is no restriction since we can easily model the former semantics by using the more general `forwards` construct in combination with a nested component that implements several services with a single object. For instance, we could encode the component defined by the expression `component provides C, D as This with new Impl(This)` in the following way:

```
component
forwards C, D as This to createNested(new Impl(This))
```

This code relies on a function `createNested` which could have the following implementation:

```
[C, D] createNested(Impl impl) {
    return new (component
                provides C as This with impl
                provides D as This with impl);
}
```

FJ's types only consist of class names. For simplicity, even Java's interface types are left out. For working with components and component instances we need syntactical forms for expressing component types and component instance types. Please note that compared to the explanations in Section 2.2.6 on page 28, we use a slightly simplified syntax for component types without enclosing parenthesis. As in *FJ*, we write $\overline{T}$ as a shortcut for $T_1, \ldots, T_n$. We use similar shorthands for sequences like $\overline{C}, \overline{f}, \overline{e}$, etc. as well as for pairs of sequences like $\overline{T}\ \overline{f}$. Such a pair of sequences is a shorthand for $T_1 f_1, \ldots, T_n f_n$.

We assume that sequences of field declarations, parameter names, and method declarations do not contain duplicate names. Furthermore, the service implementation and the component projection operators always introduce fresh names for their self reference variable. For the presentation of the operational semantics in the next section we assume to apply alpha-renaming whenever necessary to avoid name capture.

2.3.2 Semantics

The semantics of our calculus are formalized in Figure 2.8 and Figure 2.9 as a small-step operational semantics. The reduction relation has the form $e \longrightarrow e'$

$$\text{(R-Fld)}\ \frac{\text{fields}(C) = \overline{T}\,\overline{f}}{\text{new}\,C(\overline{e}).f_i \longrightarrow e_i} \qquad \text{(R-Serv)}\ \frac{\text{service}(\text{new}\,e, e, C) = e'}{\text{new}\,e :: C \longrightarrow e'}$$

$$\text{(R-Inv)}\ \frac{\text{mbody}(m, C) = (\overline{x}, e_0)}{\text{new}\,C(\overline{e}).m(\overline{d}) \longrightarrow [\overline{d}/\overline{x}, \text{new}\,C(\overline{e})/\texttt{this}]\,e_0}$$

$$\text{(R-Req)}\ e\ \texttt{requires}\ C \longrightarrow e \qquad \text{(R-MixC)}\ e\ \texttt{mixin component} \longrightarrow e$$

$$\text{(R-MixP)}\ \begin{array}{c} e\ \texttt{mixin}\ (e_0\ \texttt{provides}\ C\ \texttt{as}\ x\ \texttt{with}\ d) \\ \longrightarrow\ (e\ \texttt{mixin}\ e_0)\ \texttt{provides}\ C\ \texttt{as}\ x\ \texttt{with}\ d \end{array}$$

$$\text{(R-MixF)}\ \begin{array}{c} e\ \texttt{mixin}\ (e_0\ \texttt{forwards}\ \overline{C}\ \texttt{as}\ x\ \texttt{to}\ d) \\ \longrightarrow\ (e\ \texttt{mixin}\ e_0)\ \texttt{forwards}\ \overline{C}\ \texttt{as}\ x\ \texttt{to}\ d \end{array}$$

Figure 2.8: Operational semantics.

$$\text{(RC-Fld)}\ \frac{e \longrightarrow e'}{e.f \longrightarrow e'.f} \qquad \text{(RC-InvR)}\ \frac{e \longrightarrow e'}{e.m(\overline{d}) \longrightarrow e'.m(\overline{d})}$$

$$\text{(RC-InvA)}\ \frac{e_i \longrightarrow e_i'}{d.m(\ldots, e_i, \ldots) \longrightarrow d.m(\ldots, e_i', \ldots)}$$

$$\text{(RC-NewA)}\ \frac{e_i \longrightarrow e_i'}{\text{new}\,C(\ldots, e_i, \ldots) \longrightarrow \text{new}\,C(\ldots, e_i', \ldots)}$$

$$\text{(RC-Inst)}\ \frac{e \longrightarrow e'}{\text{new}\,e \longrightarrow \text{new}\,e'} \qquad \text{(RC-Serv)}\ \frac{e \longrightarrow e'}{e :: C \longrightarrow e' :: C}$$

$$\text{(RC-Req)}\ \frac{e \longrightarrow e'}{e\ \texttt{requires}\ C \longrightarrow e'\ \texttt{requires}\ C}$$

$$\text{(RC-Prv)}\ \frac{e \longrightarrow e'}{e\ \texttt{provides}\ C\ \texttt{as}\ x\ \texttt{with}\ d \longrightarrow e'\ \texttt{provides}\ C\ \texttt{as}\ x\ \texttt{with}\ d}$$

$$\text{(RC-Fwd)}\ \frac{e \longrightarrow e'}{e\ \texttt{forwards}\ \overline{C}\ \texttt{as}\ x\ \texttt{to}\ d \longrightarrow e'\ \texttt{forwards}\ \overline{C}\ \texttt{as}\ x\ \texttt{to}\ d}$$

$$\text{(RC-MixL)}\ \frac{e \longrightarrow e'}{e\ \texttt{mixin}\ d \longrightarrow e'\ \texttt{mixin}\ d} \qquad \text{(RC-MixR)}\ \frac{d \longrightarrow d'}{e\ \texttt{mixin}\ d \longrightarrow e\ \texttt{mixin}\ d'}$$

Figure 2.9: Congruence rules for the operational semantics.

Field lookup

$$\text{fields}(\texttt{Object}) = \emptyset$$

$$\frac{CT(C) = \texttt{class } C \texttt{ extends } D \,\{\, \overline{T}\ \overline{f}\,;\ K\,;\ \overline{M}\,\} \qquad \text{fields}(D) = \overline{U}\ \overline{g}}{\text{fields}(C) = \overline{U}\ \overline{g}, \overline{T}\ \overline{f}}$$

Method body lookup

$$\frac{CT(C) = \texttt{class } C \texttt{ extends } D \,\{\, \overline{U}\ \overline{f}\,;\ K\,;\ \overline{M}\,\} \qquad T'\ m(\overline{T}\ \overline{x})\,\{\,\texttt{return } e;\,\}\ \in \overline{M}}{\text{mbody}(m, C) = (\overline{x}, e)}$$

$$\frac{CT(C) = \texttt{class } C \texttt{ extends } D \,\{\, \overline{T}\ \overline{f}\,;\ K\,;\ \overline{M}\,\} \qquad m \text{ not defined in } \overline{M}}{\text{mbody}(m, C) = \text{mbody}(m, D)}$$

Service lookup

$$\text{service}(e,\ e_0\ \texttt{provides } C\ \texttt{as } x\ \texttt{with } d, C) = [e/x]\,d$$

$$\text{service}(e,\ e_0\ \texttt{forwards } \overline{C}\ \texttt{as } x\ \texttt{to } d, C_i) = [e/x]\,d :: C_i$$

$$\frac{D \neq C}{\text{service}(e,\ e_0\ \texttt{provides } C\ \texttt{as } x\ \texttt{with } d, D) = \text{service}(e, e_0, D)}$$

$$\frac{D \notin \overline{C}}{\text{service}(e,\ e_0\ \texttt{forwards } \overline{C}\ \texttt{as } x\ \texttt{to } d, D) = \text{service}(e, e_0, D)}$$

Figure 2.10: Auxiliary definitions for evaluation.

expressing that expression e evaluates to expression e' in a single step. Figure 2.8 specifies the basic evaluation rules, Figure 2.9 defines the evaluation contexts.

We adopt all reduction rules from *FJ* and define various new rules for the new syntactical constructs. Service abstractions simply reduce to the prototype component, so they do not have any computational effect. The semantics of mix-ins are described by three reduction rules, depending on the form of the right operand. Mixing in the empty component results in the same component. For service implementations and component projections the prototype of the right operand is mixed into the left operand and the specialization itself is applied to that new component. Thus, the two operands are incrementally combined into a single component where concrete (i.e. non-abstract) service definitions of the right operand override definitions of the left operand.

The reduction rule for service selection operations relies on an auxiliary function service(e', e, C) which searches the component definition e of component instance e' for a service C. Note that the service lookup performed by service(e', e, C) is only defined on service implementation and component pro-

Well-formed types

$$\texttt{Object}\ \text{wf} \qquad \frac{CT(C) = \texttt{class}\ C\ \texttt{extends}\ D\ \{\ldots\}}{C\ \text{wf}}$$

$$\frac{\overline{C}, \overline{C'}\ \text{wf} \quad \overline{C} \cap \overline{C'} = \emptyset}{\overline{C} \Rightarrow \overline{C'}\ \text{wf}} \qquad \frac{\overline{C}\ \text{wf}}{[\overline{C}]\ \text{wf}}$$

Subtyping

$$C <: C \qquad \frac{C <: D \qquad D <: E}{C <: E} \qquad \frac{CT(C) = \texttt{class}\ C\ \texttt{extends}\ D\ \{\ldots\}}{C <: D}$$

$$\frac{\overline{C} \subseteq \overline{D} \qquad \overline{D'} \subseteq \overline{C'}}{\overline{C} \Rightarrow \overline{C'} <: \overline{D} \Rightarrow \overline{D'}} \qquad \frac{\overline{D} \subseteq \overline{C}}{[\overline{C}] <: [\overline{D}]}$$

Figure 2.11: Well-formed types and subtyping.

jection terms. Thus, even for cases where e provides a service C, evaluation of service(e', e, C) may not be well-defined if e has not been evaluated far enough. In such a case, we first have to apply rules (RC-Inst) and (RC-Serv) to further evaluate the component before making use of the actual service selection rule (R-Serv). An overview of all auxiliary definitions used by the operational semantics of Figure 2.8 are given in Figure 2.10.

2.3.3 Type System

There are three different forms of types: object types, component types and component instance types. An object type is simply denoted by a class name C. An object type is well-formed if the class name appears in the domain of the class table CT. The class table is a mapping from class names to class declarations. As in the presentation of *FJ*, we assume that we have a fixed predefined class table to simplify the notation. Otherwise we would have to parameterize all typing rules with CT. It is assumed that CT satisfies some sanity conditions: $\texttt{Object} \notin \text{dom}(CT)$, all types appearing explicitly in CT are well-formed, and there are no cycles in the subtype relation induced by CT.

Component types have the form $\overline{C} \Rightarrow \overline{C'}$ where $\overline{C}$ specifies the services required by the component and $\overline{C'}$ specifies the provided services. Services are described by object types. A component type is only well-formed if the sets of the provided and required services are disjoint. $[\overline{C}]$ types a component instance that provides the services $\overline{C}$. Figure 2.11 summarizes the well-formedness criteria on types.

Method types cannot be written explicitly. In the type system, we use the notation $\overline{T} \to T'$ for a method with the argument types $\overline{T}$ and the result type

Expression typing

$$\text{(T-Var)}\ \Gamma \vdash x : \Gamma(x) \qquad \text{(T-Fld)}\ \frac{\Gamma \vdash e : C \qquad \text{fields}(C) = \overline{T}\ \overline{f}}{\Gamma \vdash e.f_i : T_i}$$

$$\text{(T-Inv)}\ \frac{\Gamma \vdash d : C \qquad \text{mtype}(m, C) = \overline{T} \to T' \qquad \Gamma \vdash \overline{e} : \overline{U} \qquad \overline{U} <: \overline{T}}{\Gamma \vdash d.m(\overline{e}) : T'}$$

$$\text{(T-New)}\ \frac{\text{fields}(C) = \overline{T}\ \overline{f} \qquad \Gamma \vdash \overline{e} : \overline{U} \qquad \overline{U} <: \overline{T}}{\Gamma \vdash \texttt{new}\ C(\overline{e}) : C}$$

$$\text{(T-Inst)}\ \frac{\Gamma \vdash e : \varnothing \Rightarrow \overline{C}}{\Gamma \vdash \texttt{new}\ e : [\overline{C}]} \qquad \text{(T-Serv)}\ \frac{\Gamma \vdash e : [\overline{C}]}{\Gamma \vdash e :: C_i : C_i}$$

$$\text{(T-Com)}\ \Gamma \vdash \texttt{component} : \varnothing \Rightarrow \varnothing$$

$$\text{(T-Mix)}\ \frac{\Gamma \vdash e : \overline{C} \Rightarrow \overline{C'} \qquad \Gamma \vdash d : \overline{D} \Rightarrow \overline{D'}}{\Gamma \vdash e\ \texttt{mixin}\ d : (\overline{C} \cup \overline{D}) \setminus (\overline{C'} \cup \overline{D'}) \Rightarrow \overline{C'} \cup \overline{D'}}$$

$$\text{(T-Req)}\ \frac{C\ \text{wf} \qquad \Gamma \vdash e : \overline{D} \Rightarrow \overline{D'}}{\Gamma \vdash e\ \texttt{requires}\ C : \overline{D} \cup C \Rightarrow \overline{D'} \setminus C}$$

$$\text{(T-Prv)}\ \frac{C\ \text{wf} \qquad \Gamma \vdash e : \overline{D} \Rightarrow \overline{D'} \qquad \Gamma, x : [\overline{D} \cup \overline{D'} \cup C] \vdash d : B \qquad B <: C}{\Gamma \vdash e\ \texttt{provides}\ C\ \texttt{as}\ x\ \texttt{with}\ d : \overline{D} \setminus C \Rightarrow \overline{D'} \cup C}$$

$$\text{(T-Fwd)}\ \frac{\overline{C}\ \text{wf} \qquad \Gamma \vdash e : \overline{D} \Rightarrow \overline{D'} \qquad \Gamma, x : [\overline{D} \cup \overline{D'} \cup \overline{C}] \vdash d : [\overline{B}] \qquad \overline{C} \subseteq \overline{B}}{\Gamma \vdash e\ \texttt{forwards}\ \overline{C}\ \texttt{as}\ x\ \texttt{to}\ d : \overline{D} \setminus \overline{C} \Rightarrow \overline{D'} \cup \overline{C}}$$

Method and class typing

$$\text{(T-Meth)}\ \frac{\begin{array}{c}\overline{T}\ \text{wf} \qquad T'\ \text{wf} \qquad \overline{x} : \overline{T}, \texttt{this} : C \vdash e : U \qquad U <: T' \\ CT(C) = \texttt{class}\ C\ \texttt{extends}\ D\ \{\ldots\} \qquad \text{override}(m, D, \overline{T} \to T')\end{array}}{T'\,m(\overline{T}\ \overline{x})\ \{\,\texttt{return}\ e;\,\}\ \text{ok in } C}$$

$$\text{(T-Class)}\ \frac{\begin{array}{c}D\ \text{wf} \qquad \overline{T}\ \text{wf} \qquad K = C(\overline{U}\ \overline{g}, \overline{T}\ \overline{f})\ \{\,\text{super}(\overline{g});\ \texttt{this}.\overline{f} = \overline{f};\,\} \\ \text{fields}(D) = \overline{U}\ \overline{g} \qquad \overline{M}\ \text{ok in } C\end{array}}{\texttt{class}\ C\ \texttt{extends}\ D\ \{\overline{T}\ \overline{f};\ K;\ \overline{M}\}\ \text{ok}}$$

Figure 2.12: Type system.

Method type lookup

$$\frac{CT(C) = \texttt{class}\ C\ \texttt{extends}\ D\ \{\,\overline{U}\ \overline{f}\,;\ K\,;\ \overline{M}\,\} \qquad T'm(\overline{T}\ \overline{x})\ \{\,\texttt{return}\ e;\,\} \in \overline{M}}{\text{mtype}(m, C) = \overline{T} \to T'}$$

$$\frac{CT(C) = \texttt{class}\ C\ \texttt{extends}\ D\ \{\,\overline{T}\ \overline{f}\,;\ K\,;\ \overline{M}\,\} \qquad m \text{ not defined in } \overline{M}}{\text{mtype}(m, C) = \text{mtype}(m, D)}$$

Valid method overriding

$$\frac{\text{mtype}(m, C) = \overline{U} \to U' \text{ implies } \overline{U} = \overline{T} \text{ and } U' = T'}{\text{override}(m, C, \overline{T} \to T')}$$

Figure 2.13: Auxiliary definitions for typing.

T'. Note that depending on the context, $\overline{T}$ denotes either a sequence of types $(T_1, \ldots, T_n)$ or a set of types $\{T_1, \ldots, T_n\}$. We use shorthands of the form $\overline{C} \cup D$ for expressing $\overline{C} \cup \{D\}$.

Figure 2.11 also defines a subtype relation $T <: T'$ between two types T and T'. Subtyping of object types is identical to subtyping in *FJ*. A component instance type is a subtype of another component instance type if the services provided by the supertype constitute a subset of the subtype's provided services. A component type $\tau_1 = \overline{C} \Rightarrow \overline{C'}$ is a subtype of another component type $\tau_2 = \overline{D} \Rightarrow \overline{D'}$, if τ_1 requires less and provides more services than τ_2; i.e. $\overline{C} \subseteq \overline{D}$ and $\overline{D'} \subseteq \overline{C'}$. This corresponds to the typical co/contravariant subtyping rule for function types [46] adopted already by related approaches to component subtyping [71, 182, 79]. Section 2.3.6 discusses an alternative subtyping rule which is more flexible but also more complex.

The type system is presented in Figure 2.12. There are three different typing judgment forms. The one for classes has the form "L ok" meaning that class declaration L is type correct. The judgment for method declarations has the form "M ok in C," expressing that the method declaration M typechecks as a declaration of class C. Both rules are directly taken from *FJ*. The judgment for expressions $\Gamma \vdash e : T$ relates a type T to an expression e. Most typing rules for expressions are straightforward. (T-Prv) and (T-Fwd) are among the interesting rules. Here, the service provision expression is typed under an extended environment, including the self reference to the own component instance. We assume that the type of the self reference variable corresponds to a component instance type offering both, the services that are required and provided by the component being specialized. The auxiliary definitions used for typing field and method selections as well as object creations are directly adopted from *FJ* and summarized in Figure 2.13.

2.3.4 Type Soundness

The main purpose of a static type system is to prevent the occurrence of errors during the execution of a program. In the context of our language, errors would typically arise when unknown methods get invoked, or undefined names or component services are accessed. In this section we show that our type system from Figure 2.8 prevents such errors. Our reasoning follows the style of Wright and Felleisen [209]. It is based on a *subject reduction* theorem stating that for a well-typed term e which evaluate to a new term e', this new term e' has to be well-typed as well, with a type that is a subtype of the original type.

Unfortunately, this property does not hold for the type system and the semantics presented so far. This is due to the fact that our type system supports two different ways to abstract over services: explicitly via the `requires` primitive, and implicitly via subtyping. The following example exhibits the problems. Suppose we have the following class definition which refers to an arbitrary type `A`:

```
class C {
  A a;
  C(A a) {
    this.a = a;
  }
  (A ⇒ C) foo((A ⇒) x) {
    return x provides C as This with new C(This::A);
  }
}
```

Under the assumption that the identifier a refers to an object of type A, evaluating the expression `new C(a).foo(component)` of component type $A \Rightarrow C$ will yield the term `component provides C as This with new C(This::A)`. Obviously, this term is not well-typed anymore, since `This` has only type $[C]$ and therefore does not provide the service `A` which is selected in the service implementation term `new C(This::A)`.

We could easily fix the problem by making component subtyping invariant in the required part. This approach would not restrict the expressiveness, but require that programmers coerce components explicitly to the right type by issuing extra `requires` clauses. Such coercions could also be done automatically by an appropriate type system formalism which makes implicit introductions of required services explicit by inserting additional `requires` clauses during type assignment.

Since we do not want to make sacrifices regarding the subtype relation, we pursue a different approach. Instead of making component subtyping invariant in the required part, we give up the necessity to declare required services explicitly before accessing them in another service implementation. In such a setting, the type checker has to infer the requirements of service implementations. In

the type system, this is achieved by weakening the typing rules for `provides` and `forwards` terms in the following way:

$$\text{(T-Prv')}\ \frac{C\ \text{wf} \qquad \Gamma \vdash e : \overline{D} \Rightarrow \overline{D'} \qquad \Gamma, x : [\overline{D''}] \vdash d : B \qquad B <: C}{\Gamma \vdash e\ \texttt{provides}\ C\ \texttt{as}\ x\ \texttt{with}\ d : (\overline{D} \cup \overline{D''}) \setminus (\overline{D'} \cup C) \Rightarrow \overline{D'} \cup C}$$

$$\text{(T-Fwd')}\ \frac{\overline{C}\ \text{wf} \qquad \Gamma \vdash e : \overline{D} \Rightarrow \overline{D'} \qquad \Gamma, x : [\overline{D''}] \vdash d : [\overline{B}] \qquad \overline{C} \subseteq \overline{B}}{\Gamma \vdash e\ \texttt{forwards}\ \overline{C}\ \texttt{as}\ x\ \texttt{to}\ d : (\overline{D} \cup \overline{D''}) \setminus (\overline{D'} \cup \overline{C}) \Rightarrow \overline{D'} \cup \overline{C}}$$

In this weaker system, which uses the rules (T-Prv') and (T-Fwd'), we allow that `provides` and `forwards` primitives introduce service abstractions in a non-deterministic way. We show type soundness for this weaker type system. As a consequence, the type system with the stronger typing rules, presented in Figure 2.12, is sound as well in the sense that term evaluation will not get stuck due to "symbol not found," "unknown method," or "unknown service" errors.

Programs written by users are initially typed with the stronger typing rules. The stronger type system has the advantage that typings are deterministic. Furthermore, its design follows the principle that service abstractions have to be declared explicitly. Weakening the type system is only necessary for subject reduction to hold. We present the type soundness results for our weaker type system in the style of Wright and Felleisen [209]. The full proof can be found in Appendix A.

Theorem 2.3.1 (Subject reduction) *If all types in* Γ *are well-formed,* $\Gamma \vdash e : T$ *and* $e \longrightarrow e'$*, then* $\Gamma \vdash e' : T'$ *for some* $T' <: T$*.*

For a well-typed term which can be reduced to a second term, Theorem 2.3.1 states that this second term is also well-typed. Furthermore, the type of the second term is a subtype of the type of the first term.

In addition to that we can show that the evaluation of every well-typed term does not get stuck. To formalize this, Figure 2.14 introduces a term subset denoting values.

A value is either a component, a component instance or an object. For component values we have three different constructors. One denotes the empty component, one adds a new service to an existing component value, and a third one adds services by forwarding them to another component instance. Note that during evaluation, service abstractions are eliminated in expressions with reduction rule (R-Req). Therefore, the definition of component values does not include the `requires` primitive.

Theorem 2.3.2 states that every well-typed term is either a value or it can be reduced to another term. In other words, evaluation does not get stuck for well-typed terms.

Theorem 2.3.2 (Progress) *If* $\vdash e : T$ *then* e *is either a value or* $e \longrightarrow e'$ *for some* e'*.*

Values

$$
\begin{array}{rcll}
v & = & c & \textit{component} \\
 & | & \texttt{new}\ c & \textit{component instance} \\
 & | & \texttt{new}\ C(\overline{v}) & \textit{object}
\end{array}
$$

Component values

$$
\begin{array}{rcl}
c & = & \texttt{component} \\
 & | & c\ \texttt{provides}\ C\ \texttt{as}\ x\ \texttt{with}\ e \\
 & | & c\ \texttt{forwards}\ \overline{C}\ \texttt{as}\ x\ \texttt{to}\ e
\end{array}
$$

Figure 2.14: Term values.

2.3.5 Instantiation Evaluation

The operational semantics presented in Figure 2.8 formalizes an evaluation strategy that does not allow for the reduction of service implementation expressions inside of component instances. At component instantiation time, in fact none of these terms get evaluated. A term specifying a service implementation, for example in `provides` or `forwards` primitives, only gets evaluated when the service is accessed via the :: operator. Evaluating a service implementation expression more than once does not cause any problems in our calculus, since we only have functional objects without any side-effects. In real-world systems, this form of *lazy* evaluation can be efficiently implemented using a memoization technique, so that for multiple accesses to the same service, the service implementation expression will be evaluated only once. We decided to have this restriction in our calculus for several reasons. First, it keeps the calculus simple. But lazy evaluation also constitutes a reasonable evaluation strategy for service implementations. A *strict* evaluation order would be difficult to define. For instance, we could evaluate the service implementations in the order the component evolution primitives introduce a service. But this would be a completely arbitrary choice, since services can be introduced using the `requires` primitive in any order, not implying any dependencies.

With any fixed strict evaluation order one risks to access a not yet initialized service from the service implementation that is currently being evaluated. With a lazy service evaluation strategy one still faces this problem, but only for recursive service references. With our operational semantics, such recursive dependencies could possibly lead to infinite computations. We avoided this problem in the examples of the previous sections by not accessing services of the own component instance in service provision expressions directly. Instead, objects that implement a service of a component access other services of the same component instance only at the time a method of the other service actually has to be called, which happens typically after the component got instantiated.

$$\text{(S-Emb)}\ \frac{e \longrightarrow e'}{d\,;\overline{D} \vdash e \hookrightarrow e'}$$

$$\text{(S-Prv)}\ \frac{[d/x]\,e \longrightarrow e' \qquad C \notin \overline{D}}{d\,;\overline{D} \vdash e_0\ \texttt{provides}\ C\ \texttt{as}\ x\ \texttt{with}\ e \hookrightarrow e_0\ \texttt{provides}\ C\ \texttt{as}\ x\ \texttt{with}\ e'}$$

$$\text{(S-Fwd)}\ \frac{[d/x]\,e \longrightarrow e' \qquad \overline{C} \backslash \overline{D} \neq \emptyset}{d\,;\overline{D} \vdash e_0\ \texttt{forwards}\ \overline{C}\ \texttt{as}\ x\ \texttt{to}\ e \hookrightarrow e_0\ \texttt{forwards}\ \overline{C}\ \texttt{as}\ x\ \texttt{to}\ e'}$$

$$\text{(SC-Prv)}\ \frac{d\,;\overline{D} \cup C \vdash e_0 \hookrightarrow e_0'}{d\,;\overline{D} \vdash e_0\ \texttt{provides}\ C\ \texttt{as}\ x\ \texttt{with}\ e \hookrightarrow e_0'\ \texttt{provides}\ C\ \texttt{as}\ x\ \texttt{with}\ e}$$

$$\text{(SC-Fwd)}\ \frac{d\,;\overline{D} \cup \overline{C} \vdash e_0 \hookrightarrow e_0'}{d\,;\overline{D} \vdash e_0\ \texttt{forwards}\ \overline{C}\ \texttt{as}\ x\ \texttt{to}\ e \hookrightarrow e_0'\ \texttt{forwards}\ \overline{C}\ \texttt{as}\ x\ \texttt{to}\ e}$$

Figure 2.15: Operational semantics for component instantiation.

In order to support *any* reasonable evaluation strategy[2] for component instantiations, we could extend our operational semantics. We only have to replace rule (RC-Inst) of Figure 2.8 with the following rule (R-Inst):

$$\text{(R-Inst)}\ \frac{\texttt{new}\ e\,;\emptyset \vdash e \hookrightarrow e'}{\texttt{new}\ e \longrightarrow \texttt{new}\ e'}$$

This rule relies on a context dependent reduction semantics for service implementations during component instantiation. Intuitively, the clause $d\,;\overline{D} \vdash e \hookrightarrow e'$ expresses that evaluation of term e within component instance d results in term e'. Furthermore, services contained in $\overline{D}$ are overridden and excluded from evaluation. This service exclusion ensures that we do not execute service implementations that are superseded by other more recently defined implementations. A definition of the service evaluation semantics can be found in Figure 2.15. Rule (S-Emb) embeds the original reduction relation $\longrightarrow$ into $\hookrightarrow$ making sure that the new semantics are a conservative extension of the previous version. Rules (S-Prv) and (S-Fwd) evaluate a service implementation expression. The rules (SC-Prv) and (SC-Fwd) propagate evaluation to more deeply nested services.

2.3.6 Component Subtyping

The subtyping rule presented so far only supports *width*-subtyping for component types; i.e. subtypes provide more and require less services. We could relax this rule to support a form of *depth*-subtyping which incorporates subtyping of

[2] An evaluation strategy is considered to be *reasonable* if it does not evaluate overridden service implementation expressions.

service interface types. Here, $\tau_1 <: \tau_2$ would hold for two component types τ_1 and τ_2, if the required service types of τ_1 are supertypes of the required service types of τ_2. Similarly, the provided service types of τ_1 are supposed to be subtypes of the provided service types of τ_2. The following alternative subtyping rule expresses exactly this relationship:

$$\frac{\forall i \exists j : D_j <: C_i \qquad \forall i \exists j : C'_j <: D'_i}{\overline{C} \Rightarrow \overline{C'} <: \overline{D} \Rightarrow \overline{D'}} \qquad \frac{\forall i \exists j : C_j <: D_i}{[\overline{C}] <: [\overline{D}]}$$

To make use of such a rule in our type system, we would also have to update the subtyping rule for component instances together with the typing rules (T-Mix), (T-Req), (T-Prv), and (T-Fwd). Furthermore, the service lookup function would have to be modified to reflect the fact that we can now override a service by introducing a new service with a specialized type.

Overall, depth-subtyping was not considered in the formalization of our calculus since it would have complicated the technical treatment significantly. It would require many subtle technical restrictions dealing with "overlapping services" (service types that share a supertype) and service overriding in general, which would yield both complex and unintuitive semantics and typing rules.

2.4 Discussion

We now summarize the main ingredients of our component model, explain design decisions, and compare the constructs with related work. The following features are reviewed in detail:

1. Components require and provide services,
2. Components are templates for component instances,
3. Components are composed with explicit links (forwarding), or implicitly by wiring inference (mixins), and
4. Components are extensible through inheritance and overriding.

Prototype-based components. In the presented model, components are first-class abstractions that have neither state nor identity. Components define the structure of component instances in the same way as classes define the structure of objects. In most class-based languages, classes are either second-class entities, or they are first-class and specified using meta-classes. For simplicity, and in order to avoid such a meta-regress [201], our first-class components are prototype-based [1]. Thus, instead of instantiating components from meta-component descriptions, new components are derived from prototypical components by a set of specialization primitives. Since components are stateless, we do not need a cloning operation known from object-based programming languages [49, 201]. This approach emphasizes the reuse of components in the creation of new, extended components by specialization. In fact, even component composition, which is mostly regarded as the only form of component reuse, is explained in terms of component specialization.

Services. Components specify implementations for a set of provided services. These implementations may rely on services provided by other components. Thus, component types are characterized by a set of required and provided services. The presented model requires that required and provided services are specified explicitly. Alternatively, it would be possible to fully infer such information, as Nierstrasz explains in his work on *contractual types* [148].

Services are described by nominal object types. In Section 2.2.3 we explained already why this approach does not constitute a restriction compared to component models with named ports [182, 187, 5]. Our service abstraction does not only allow us to conveniently refer to an aggregate of functionality, as opposed to individual methods, for instance. It also facilitates to override an aggregate of functionality consistently and promotes distinct, non-interfering views of components. Service specifications that are solely based on nominal object types were inspired by COM [172, 93].

Composition. Services are added to a component using the service abstraction and service implementation primitives. For composing components, two mechanisms are supported: forwarding and mixin-based composition. Forwarding delegates the implementation of a set of services to another, possibly nested component instance. The significance of the forwarding primitive is two-fold: On the one hand it enables hierarchical component compositions, on the other hand, it is used to hide internal services of encapsulated components.

Unlike forwarding, the mixin-based approach merges two components by specializing one component with the services provided by another component and by rebinding the self reference to the merged component. Compared to the approach based on forwarding where the services of the nested component cannot be overridden and are therefore statically linked, component composition based on mixins yields a fully extensible component where it is possible to redefine service implementations by overriding. On the other hand, forwarding allows one to specify exactly what services to include, in contrast to the mixin-based approach which always mixes in all provided services. As mentioned already in Section 2.2.7, this may lead to accidental overrides. This weakness of the type system could be addressed, for example, by making overriding explicit and by including negative information in component types. Discussions about forwarding versus delegation (object-based inheritance), which is sometimes also used as an implementation technique for mixins, can be found, for instance, in [195, 105, 39]. Support for dynamic object-based inheritance in a class-based context is provided by Büchi's and Weck's *generic wrappers* [39], Kniesel's object model Darwin [105], Ostermann, Mezini, and Wittman's work on FamilyJ [208], and by Ostermann's *delegation layers* [156].

Mixins were first identified as linguistic abstractions for generalizing inheritance by Bracha and Cook [29]. It was also Bracha who observed that inheritance can be seen as a mechanism for modular program composition [31]. With his work on the programming language Jigsaw [28], he lifts the notion of class-based inheritance and overriding to the level of modules.

A formal account of mixins and mixin-based inheritance is given in [26, 72, 9]. In particular, Bono, Patel, and Shmatikov's calculus of first-class classes and mixins is similar to our work [26]. Bono's mixins correspond to components in our model. Classes correspond roughly to components without required services. Based on the same framework, Bettini, Bono, and Venneri recently showed that mixins are suitable abstractions for mobile software components [24]. As opposed to the work by Bono *et al.*, the programming language Scala [151] does not distinguish between classes and mixins. It only has the notion of classes that are interpreted as mixins when used in mixin-based class compositions (inheritance). This is identical to the way components are interpreted in the presented model. Scala's mixins are formalized in [153]; the design was inspired by Strongtalk [17, 30], an extension of the programming language Smalltalk.

Chapter 3

Static Component Evolution with Extensible Modules

This chapter presents KERIS, a pragmatic, backward-compatible extension of the programming language JAVA [82] with explicit support for modular, component-oriented programming of extensible software.

The design of the programming language KERIS was driven by the observation that extensibility on the module level can help to develop highly extensible applications [94]. KERIS tries to facilitate the development of extensible software in JAVA by providing an additional layer for structuring software components. This layer features extensible modules as the basic building blocks of software. KERIS provides primitives for creating and linking modules as well as mechanisms for extending modules or even fully linked programs statically. Programs written in KERIS are *closed* in the sense that they can be executed, but they are *open* for extensions that statically add, refine or replace modules or even whole subsystems of interconnected modules. Extensibility does not have to be planned ahead and does not require modifications of existing source code, promoting a smooth software evolution process. KERIS is a strongly typed language. The type system ensures that the definition, assembly, and evolution of modules is safe.

The overall design of the language was guided by the aim to develop a pragmatic, implementable, and conservative extension of JAVA which fully reuses JAVA's compilation model and target platform. This allows for a seamless integration of existing JAVA code with KERIS. For this reason, KERIS does not even utilize customized class loaders, which could have complicated the use of existing JAVA technology, like the RMI API, which relies already on special class loading techniques itself. KERIS adopts JAVA's compilation model because it proved to work well in practice. Furthermore, changes would be confusing for many programmers that are used to a development process which exploits separate compilation, as well as dynamic class loading and linking, and which relies on the notion of binary compatibility for relating different versions of binary components, i.e. classfiles.

The remainder of this chapter first discusses shortcomings of the package system of JAVA for implementing reusable software components. This is followed by a presentation of the design of KERIS. In this section, the new language features are introduced incrementally and explained with the help of many small examples. After that, various application scenarios are given to explain how extensible modules support the safe development of extensible software. The following section explains how the KERIS compiler translates the high-level language to plain JAVA code. Finally, benchmarks are used to evaluate this source-level translation with respect to code size and runtime performance. The chapter concludes with a discussion about module systems in general and the module system of KERIS in particular.

3.1 The Java Package System

Like many popular object-oriented languages, JAVA provides relatively weak abstractions for *programming in the large* [56]. In this section we argue that JAVA's packages are not expressive enough to be useful as abstractions for reusable and extensible software components. We do this by looking at three important properties: *modularity*, *genericity*, and *extensibility*. A more extensive discussion about module system related issues can be found in Section 3.6.

3.1.1 Modularity

Modularity is about the separation of components from other components both logically and physically. Therefore, modularity is essential to allow software components to be developed and compiled independently. This is typically achieved by means of encapsulation and by the explicit specification of contracts between components. These contracts define explicitly what services a component provides and what other components are needed to render the services.

JAVA's package system offers relatively good support for modular program development. It allows that context dependencies are specified explicitly and it has support for separate compilation. On the other hand, JAVA's package abstraction is often too coarse-grained, so that structuring software systems consisting of many smaller subsystems can become very difficult on the package level. For instance, large libraries often require means for internal structuring. It is possible to nest packages, but this also limits access to non-public members. Therefore all classes that need to access library internal data, which does not get exposed to the outside world, have to reside in the same package.

JAVA's package mechanism was designed mainly for structuring the name space and for grouping classes. A package does not even allow programmers to fully encapsulate a set of classes since the JAVA programming language does not offer a way to close packages.[1] Thus, like in most popular object-oriented languages, classes are predominantly used to structure software systems.

Classes on the other hand do not fully support modular programming either [192, 37]. In general, classes cannot be compiled separately; mutually dependent classes have to be compiled simultaneously. Since classes do not define context dependencies explicitly, it is difficult to find out on what other classes a class depends. This can only be found out by inspecting code.

Even though classes are the basic building blocks for object-oriented programming, most classes do not mean anything in isolation. They have a role in a spe-

[1] In JAVA, class loaders can be used at runtime to ensure that only a fixed set of classes is loaded from a package. The concept of *sealed packages* exploits this mechanism to restrict class loading for classes of such a package only to a particular *Jar* file. Regarding the open nature of packages, it is surprising to see that adding classes to a JAVA package is a fragile and unsafe operation. It can break programs that import all classes of a package via the star-import command.

cific program structure, but there is only limited support to formulate this role or to make this role explicit. A priori, class interactions are implicit, if not using a special design pattern that emphasizes cooperating classes. Due to the lack of expressing dependencies between classes explicitly, formulating design patterns, software components, the architecture of a system, and even expressing the notion of a library on the level of the programming language turns out to be extremely difficult in general.

A good example for this problem is the way how industrial component models represent software components in class-based object-oriented languages. In these models, the implementation of a software component is typically guided by a relatively weekly specified programming protocol (e.g. JAVABEANS [190]). The composition of software components is even mostly performed outside of the programming language, using meta-programming technologies. Thus, neither the process of manufacturing a component nor the component composition mechanism are type-safe.

3.1.2 Genericity

Modularity is essential for the independent development of software components. But modularity alone does not allow programmers to deploy components independently of each other. Support for independent deployment requires that modules are generic with respect to their context dependencies; i.e. they have to abstract over depending modules. Furthermore, a mechanism is needed to instantiate a component and resolve its context dependencies by linking it with concrete instances of depending components. Thus, genericity is required whenever one wants to reuse a single component in different contexts with different cooperating components.

JAVA packages are not generic. Packages hard-wire their context dependencies (imports) by referring to other concrete packages. Thus, there is no explicit support for the reuse of packages in different contexts or with different compatible dependent packages. Even though references to other packages are specific, the JAVA runtime environment offers possibilities to adjust the "linking context" so that a different implementation of a cooperating package is chosen at load-time; for instance by modifying the class path or by using special class loaders [163]. Such hacks are statically unsafe and therefore do not provide acceptable alternatives for genericity.

3.1.3 Extensibility

Besides modularity and genericity, *extensibility* is another important property. As the previous chapters pointed out already, extensibility is important because in general, independently developed components do not fit off-the-shelf into arbitrary deployment contexts. They first have to be adapted to make them compliant

with a particular deployment scenario. Apart from this, extensibility is an essential requirement for facilitating the evolution of software. Software evolution includes the maintenance and extension of component features and interfaces. A typical software evolution process yields different versions of a single component being deployed in different contexts [132]. Extensibility is also required when developing *families of software applications* [158, 27]. For instance, *software product-lines* [99, 205] rely heavily on a mechanism for creating variants of a system which share a common structure but which are configured with possibly different components.

JAVA supports the development of extensible software only on a very low level by means of class inheritance and subtype polymorphism. Extensibility has to be planned ahead through the use of design patterns, typically derived from the *AbstractFactory* pattern [74]. Furthermore, extensibility can often only be achieved by using type tests and type casts due to the lack of appropriate means to refine abstractions, in particular types and classes, covariantly. In general, such techniques circumvent the static type system and are therefore dangerous to apply in practice.

With JAVA's late binding mechanism and its support for reflection, developing open software which can be extended with *plug-ins* is relatively easy. Again, this has to be planned ahead and allows only extensions in a restricted framework [132]. For writing applications that are open for extensions that have not been anticipated, often complicated programming protocols have to be strictly observed. An example for such a protocol is the *Context/Component* design pattern described in Section 4.2.1.

3.2 The Programming Language Keris

KERIS[2] extends the programming language JAVA with an expressive module system that aims at facilitating the development of extensible software components [213, 211]. With the module abstractions of KERIS, it is possible to give concrete implementations for concepts like design patterns, libraries, applications, or subsystems. All this is done in a completely extensible fashion, allowing to refine existing software or to derive new extended software from existing pieces. To keep software extensible, KERIS promotes programming without hard links which are frequently found in JAVA programs in form of class instantiations or accesses to static methods or fields. The module system of KERIS is designed to fit smoothly between JAVA's class and package level. With support for true modules, the package system is now mainly used to structure the module name space. Of course, it would easily be possible to add module name space management facilities to the module abstractions of KERIS if backward compatibility to JAVA would be irrelevant.

3.2.1 Declaring Modules

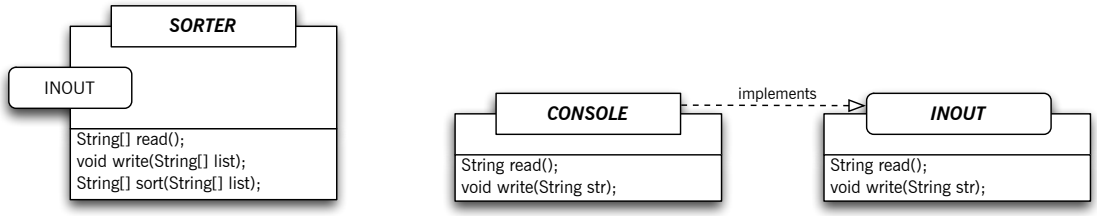
In KERIS, modules are the basic top-level building blocks of software supporting separate compilation as well as function and type abstraction in an extensible fashion. In general, modules depend on functionality provided by other modules. In KERIS, such context dependencies are specified explicitly. A module can only be deployed in contexts that meet such dependency requirements.

We now present a small example that defines a module SORTER which provides functions for reading a list of words, for sorting, and for printing out lists. Throughout this chapter we write module names in capital letters to distinguish them clearly from class, method, and variable names. A formal grammar describing the syntax of KERIS can be found in Appendix B.

```
module SORTER requires INOUT {
  String[] read() { ... INOUT.read() ... }
  void write(String[] list) { ... INOUT.write(list[i]) ... }
  String[] sort(String[] list) { ... }
}
```

Explicit context dependencies. The header of the module declaration states explicitly that module SORTER depends on functionality provided by another module INOUT. Within the body of a module it is possible to access the members of the own module as well as all the members of modules that are declared to be

[2] A *Keris* is a double edged dagger originating in the Javanese culture. It was considered a magical weapon, filled with great spiritual power. Today it is an object of reverence and respect, symbolizing strength and safety.



(a) Sorter module (b) Console module

Figure 3.1: Schematic illustration of modules SORTER and CONSOLE.

required. Members of modules are generally accessed by qualifying their member names with the corresponding module reference. This distinguishes *requirements* from *imports* of many traditional module systems which make members of other modules accessible so that they can be used in unqualified form. Some module systems have both forms; e.g. Modula-2 [206] supports regular imports as well as *qualified imports* which correspond to our requirements.

Module members. In Keris, modules encapsulate functions, fields, and class abstractions, like interfaces and classes. The syntax for module member declarations corresponds to the Java syntax used on the class level. Later we will see that modules may also contain submodules.

Module interfaces. Regarding the previous listing, it remains to show a specification of module INOUT. We do this by defining a module interface that specifies the signature of this module. Such a module interface does not contain program code, it only lists all members provided by concrete implementations of this module together with their corresponding types.

```
module interface INOUT {
  String read();
  void write(String str);
}
```

We will now define a module CONSOLE that implements this interface and thus is a possible candidate for being used in conjunction with module SORTER.

```
module CONSOLE implements INOUT {
  String read() {
    ... System.in.read() ...
  }
  void write(String str) {
    System.out.println(str);
  }
}
```

This module implements the functions `read` and `write` by forwarding the calls to appropriate methods of the standard JAVA API for text in- and output on a terminal. An alternative implementation for `INOUT` based on functionality provided by a third module `LOG` is given by the following program:

```
module LOGIO implements INOUT requires LOG {
  String read() { ... System.in.read() ... }
  void write(String str) { LOG.log("log: " + str); }
}
```

The previous example code shows that a module interface can be implemented by many modules. On the other side, a module implementation can implement many module interfaces. Note that modules are not explicitly required to implement a module interface. This is not strictly necessary since every module implementation implicitly defines a module interface of the same name containing all exported module members. Nevertheless, the separation of module implementations from interfaces is an important mechanism that is essential to enable separate compilation of recursively dependent modules. Some module systems, e.g. OBERON's module system, provide means to support separate compilation without separating module interface definitions from module implementations, but this works only for modules without recursive dependencies.

Access control. Beside separate compilation, explicit module interfaces are also important as a facility for hiding concrete representations of module members. Furthermore, they can be used as a vehicle for presenting different views on a single module implementation. This gives the programmer more flexible control over access rights to module members than JAVA does with its fixed scoping levels expressed with access modifiers like `public`, `private`, and `protected`.

The only access modifiers KERIS supports on the module level are `public` and `private`. Module members which are not explicitly tagged with an access modifier are considered to be `public`. Only the public members get exported and can therefore be accessed from clients of the module.

Figure 3.1 illustrates some of the modules defined so far. Module illustrations consist of two parts: one part shows both the module to define (the topmost box) and the required modules (the boxes on the left side). The other part lists all the other module members. We use the convention that boxes refer to modules and rounded boxes refer to module interfaces.

3.2.2 Linking Modules

Before discussing the module composition mechanism, we have to stress the distinction between modules and module instances. A module can be seen as a "template" for multiple module instances of the same structure and type. We have

to differentiate between the two, since we want to be able to deploy a module more than once within a software system. For instance, we could have two different instances of the SORTER module that are linked together with different INOUT module instances.

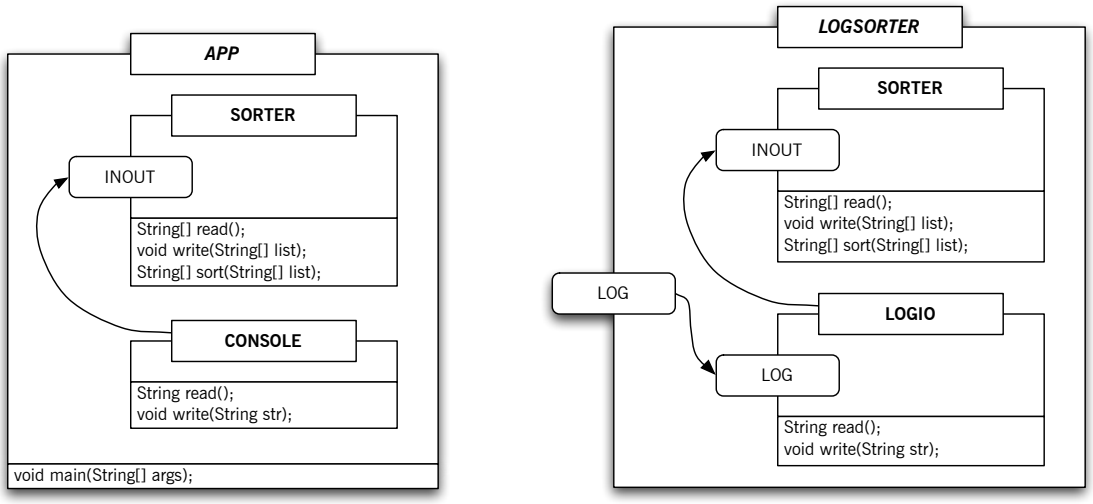
Hierarchical composition. In Keris, modules are composed by aggregation. More concretely, a module does not only define functions and variables. It may also define module instances as its members. These nested module instances, we also call them *submodules*,[3] can depend on other modules visible in the same context. The following definition for module APP links module SORTER with module CONSOLE by declaring both to be submodules of the enclosing module APP.

```
module APP {
  module SORTER;
  module CONSOLE;
  void main(String[] args) {
    String[] list = SORTER.read();
    list = SORTER.sort(list);
    SORTER.write(list);
  }
}
```

Submodule definitions start with the keyword module followed by the name of the module implementation. The enclosing module aggregates for every submodule definition an instance of the specified module. Thus, in the example above, module APP aggregates two module instances SORTER and CONSOLE. A submodule can only be defined if its deployment context, given by the enclosing module, satisfies all the requirements of the submodule. The requirements of a submodule are satisfied only if all modules required from the submodule are either provided as other submodules, or if they are explicitly required from the enclosing module.

The program above defines two submodules SORTER and CONSOLE. Module SORTER requires a module instance INOUT from the deployment context, CONSOLE does not have any context dependencies. The module definition of APP is well-formed since it defines a CONSOLE submodule that implements INOUT, and therefore provides the module that is required by the SORTER submodule. Note that module CONSOLE is only present in module APP for that reason. Module APP does not refer to members of CONSOLE directly. Figure 3.2(a) illustrates the structure of module APP. The submodules of APP are displayed as nested modules. The wiring of the submodules, which is implicit in Keris programs, is made explicit with an arrow from the implementing module to the requirement.

[3] We use a terminology here which is not fully consistent with the one on the class level. *Submodules* denote *nested modules* and have nothing to do with *subclassing*. The motivation for naming nested modules *submodules* comes from nested modules modeling *subsystems*. Our terminology is consistent with other module systems supporting hierarchical compositions of modules.



(a) Executable module APP (b) Module LOGSORTER

Figure 3.2: Schematic illustration of modules APP and LOGSORTER.

Unresolved context dependencies. Similarly to the previous code, we could try to link module SORTER with module LOGIO, as shown by the following definition of BUGGYAPP.

```
module BUGGYAPP {
  module SORTER;
  module LOGIO;  ↯
}
```

A verification of the context dependencies reveals that this module declaration is not well-formed. LOGIO requires a module instance LOG which does not get declared within BUGGYAPP. Since we want BUGGYAPP to be parametric in the cooperating module LOG, we have to abstract over the LOG instance by requiring it from the context. This has the effect that inside of the module body we are now able to refer to a module instance LOG without actually giving a concrete definition. Therefore the following definition of module LOGSORTER is well-formed. Figure 3.2(b) gives a schematic illustration which shows that for LOGSORTER, all requirements of submodules are resolved.

```
module LOGSORTER requires LOG {
  module SORTER;
  module LOGIO;
}
```

Discussion. As the previous examples show, modules get composed by hierarchically aggregating submodules. A module that hosts a set of submodules is only well-formed if it satisfies the context requirements of all of its submodules. A module satisfies the requirements of a submodule if modules required from that submodule are either present in form of other submodules, or are explicitly required by the host module, or are subsumed by the host module itself.

This hierarchical composition mechanism has the advantage that the static architecture of a system becomes explicit. Furthermore, module composition does not require to link modules explicitly by specifying how context dependencies are satisfied at deployment time. Instead, the module interconnection gets *inferred*. With this approach we avoid linking modules by hand which can be a tedious task that raises scalability issues [212]. On the other hand, our inference technique only succeeds if we avoid ambiguities; i.e. our type system has to ensure that references to module instances identify modules uniquely in every context. One implication of this is that a module can never define or require two nested module instances (submodules) that implement the same module. If this would be the case, a simple module name could not identify a module implementation unambiguously anymore.

A system like this is reminiscent of classical module systems for imperative programming languages like Modula-2 or Ada. Such module systems allow only one implementation for each module globally, whereas Keris has this restriction only locally for every module context. Globally, there are no restrictions, allowing systems to include as many instances of a single module as required.

An intuitive motivation for only allowing a single implementation of a module per context is that it would be redundant to have identical module instances providing exactly the same services. Of course, with regard to genericity and the possibility to encapsulate state, this argument is weak. But we will see later, that these limitations can be easily overcome by using *module specializations* introduced in Section 3.2.6. This mechanism makes it possible to define different, specialized versions of a module together in the same context. Furthermore, it is, of course, always possible to introduce nested modules for instantiating multiple instances of a single module. The type system, discussed in Section 3.2.8, ensures that such multiple instances are used in a consistent, non-conflicting manner.

3.2.3 Accessing Modules

Accessing submodules. The code of module `APP`, presented in the previous section, shows that aggregated submodule instances are accessed just like required modules simply via the module name. For accessing exported submodules of other modules one has to use the `::` operator. For instance, the expression `SYSTEM::APP::SORTER` accesses the `SORTER` module instance, which is a submodule of `APP`, which is a submodule of `SYSTEM`. Such expressions are also called *module paths*.

With the :: operator it is possible to access partial units of other modules. There is a whole body of literature on programming style that tries to motivate why it is good "only to talk to your immediate friends, and never to strangers" [117, 116, 114]. This principle is often called the *Law of Demeter* [115]. It states that every unit should only communicate directly with its immediate neighbors in a system. The :: operator allows programmers to break this principle. On the other hand, KERIS programs that do not make use of the :: operator conform to the Law of Demeter on the module level. Here, modules only access services of other modules that are either explicitly required or aggregated. This restriction, and the Law of Demeter in general, is comparable to the *communication integrity* property stated by Aldrich, Chambers, and Notkin [145, 5, 6].

Importing modules. For accessing members of deeply nested module instances, it is not very practical to require that one always has to fully qualify module instances by prefixing them with the right module path. Therefore KERIS offers a mechanism to shorten module paths by importing nested module instances. Here is an example:

```
module SYSTEM {
  module APP;
  import APP::SORTER;
  void main(String[] args) {
    SORTER.write(SORTER.sort(args));
  }
}
```

In this program, module SORTER nested in module APP gets imported in the body of SYSTEM so that one can refer to this module instance simply via the unqualified module name. There is no need anymore to prefix this name with module APP. An import statement is only legal if it does not create ambiguities. It really must be seen as a means to introduce shorthands for modules, since it will never have an influence on the inferred wiring of submodules.

Similar to JAVA's import mechanism on the package level, KERIS offers a second import form which imports all members of a specified module instance. This "star-import" statement is used in the following program, which is otherwise identical to the listing above.

```
module SYSTEM {
  module APP;
  import APP::SORTER.*;
  void main(String[] args) {
    write(sort(args));
  }
}
```

Keris' lookup procedure for module members first searches the current module and only if no member is found there, it proceeds by searching all "start-imported" modules. If a member is found in more than one "star-imported" module, this is considered to be an ambiguity. Ambiguities have to be resolved by the programmer, who has to explicitly qualifying the member name with the corresponding module instance.

3.2.4 Initializing Modules

Executing modules. Modules without context dependencies like module `APP` from the previous section can be executed if they define a `main` method with the following signature: `void main(String[] args)`. When a module is execute, a module instance gets created and the `main` method of this instance is invoked.

Initializing modules. A Keris module implementation can define an arbitrary number of module initializers which initialize variables and perform other side-effects. Similar to the class initialization process of Java, Keris modules are initialized lazily right at the time one of the module members gets accessed for the first time.[4] This approach can provoke cycles in the module initialization process if module initializers of mutually dependent modules refer to each other recursively. In Keris, this problem is resolved in the same way Java handles static initializers in classes: The cycle is broken dynamically, making it possible that a module accesses in its initializer a recursively dependent module which is only partially initialized. For uninitialized variables this means that they refer to some default value. Here is an example for a lazy module initialization:

```
module MAIN {
  module OPTIONS;
  void main(String[] args) {
    if (args.length > 0)
      OPTIONS.parse(args);
  }
}
```

```
module OPTIONS {
  HashMap options = new HashMap();
  { // initialization code
    options.put("-silent");
  }
  void parse(String[] args) {
    ...
  }
}
```

In this program, submodule `OPTIONS` only gets initialized if the length of array `args` in the main method of module `MAIN` is greater than `0`. Otherwise module `OPTIONS` does not get accessed and therefore the initializers will not be executed. The module remains uninitialized in this case.

[4] In Keris, accessing a submodule of a module does not trigger execution of this module's initializers.

Controlling initialization. Sometimes, the designer of a module wants the module to be seen from the outside world as an atomic unit even though internally it is constructed by aggregating several submodules. One can achieve this by declaring the submodules to be private, and thus hide them from external clients. This means does not go far enough, if the designer of such a compound module even wants the module to get fully initialized at the same time. Since submodules are initialized lazily, initializing the compound module does not imply the initialization of all submodules. To address this issue, KERIS allows the programmer to interlink the initialization of a submodule with its host module such that whenever the host module is initialized, initialization of the submodule is triggered automatically. In KERIS such a coupling can be expressed by tagging a submodule definition with the `synchronized` modifier. All synchronized submodules get initialized before the host module in the order they are declared, as long as module dependencies do not enforce a different order dynamically.

Synchronized submodules are needed when the initializer of a submodule performs side-effects that are supposed to take place before the host module initializer is executed. They are also helpful to control the initialization of submodules whose initializers depend recursively on each other. Here, a wrong initialization order might result in a submodule initializer accessing an uninitialized field of another partially initialized submodule. See Section 3.4.1 on page 98 for a discussion of related issues.

3.2.5 Refining Modules

We now come to the problem of extending modules. Since we do not want to break code that uses existing modules, we are neither allowed to touch binaries nor the source code of existing modules. In short, extensibility has to be *additive* instead of being *invasive*.

Non-invasive extensions. KERIS has support for non-invasive extensions through a *module refinement* mechanism. This mechanism allows programmers to refine an existing module by providing new functionality or by overriding existing functionality. The refined version of a module is backward compatible to the original module in the sense that it can be substituted for it. Thus, KERIS lifts the notion of compatibility between classes expressed by a subtyping relation to the more coarse-grained level of modules.

Creating refinements. We now present a refinement of module `SORTER` that provides a more efficient implementation for the sort function. In the example below, we use a merge-sort technique for sorting. Apart from a new implementation of `sort` which overrides the existing implementation, we also define various other helper functions. One of them is declared to be `private`, which hides it

from clients of the module. Such functions do not get exported and can only be used internally.

```
module FASTSORTER refines SORTER {
  private String[] sub(String[] list, int start, int end) { ... }
  String[] merge(String[] first, String[] second) { ... }
  String[] sort(String[] list) {
    return (list.length < 2) ? list :
           merge(sort(sub(list, 0, list.length/2)),
                 sort(sub(list.length/2, list.length)));
  }
}
```

Module FASTSORTER is a refinement of module SORTER. Sometimes we also call SORTER the *parent module* of FASTSORTER. A refinement inherits the implemented interfaces and the member implementations, including all submodules, from its parent module. Note that it also inherits all the context dependencies. For the example above, this actually means that module FASTSORTER requires an INOUT module. Note that the set of required and aggregated modules has to be disjunct. Otherwise, a reference to a module that is both required and aggregated would be ambiguous. It is therefore not possible that a refinement aggregates a required module of the parent module.

Similar to the refinement of module implementations and their implicit interfaces, it is also possible to refine plain module interfaces like INOUT.

Overriding members. Keris uses, like most object-oriented languages, overriding as the primary means to modify members of modules and classes. The overriding rules for module members are almost identical to the rules of Java. In particular, it is not possible to override variables, and method overriding is invariant in the parameter types. As opposed to Java, method overriding is covariant in the result type; i.e. an overriding method may have a result type that is a subtype of the result type of the overridden method. This overriding rule is reminiscent of the rule used in GJ [32].

Covariant return type specializations are often extremely useful when writing extensible software. In particular, they allow one to refine factory methods of *AbstractFactories* so that one can access newly introduced functionality of the created objects without using type casts. Return type specializations are also important when it comes to implement a clone method for a hierarchy of classes [35].

In Keris, module refinements see the module they refine as *gray-boxes*, as opposed to clients that see a module as a *black-box* with a well-defined interface. Thus, module refinements can freely access and override private members of the parent module. The `private` modifier only indicates that a module member does not get exported and consequently can only be used internally — this includes the use in refinements which do not logically constitute a different module, but

can rather be seen as amendments yielding a new version of an existing module. Members which are not supposed to be modified by refinements have to be declared `final`.

Overriding submodules. So far, we only saw how to refine the functionality of atomic modules. Such refinements are non-invasive; i.e. they do not affect existing code. The question is now, how to integrate refinements, like the more efficient sorting module, into a system that already makes use of the old SORTER module? Since systems are represented by modules, it is probably not surprising that this is done again with a refinement. As explained before, KERIS promotes programming without hard links. Following this idea, we allow overriding submodule declarations in module refinements. The following code refines the executable module APP by covariantly overriding submodule SORTER.

```
module XAPP refines APP {
  module FASTSORTER;
}
```

The refined module XAPP replaces the nested module implementation SORTER with module FASTSORTER. Consequently, the inherited `main` method now refers to the FASTSORTER submodule. In fact, we can now access the FASTSORTER submodule via both module names, SORTER and FASTSORTER. The only difference is that when accessed via FASTSORTER, we can refer to the new functions. The ability to refine a module interface stepwise to allow different access levels is called *incremental revelation* [44].

Interface ascription. Submodules may be overridden by refinements, but it is not possible to replace a submodule with a compatible version which is not a refinement. For instance, it is not possible to override the submodule CONSOLE in refinements of APP with alternative implementations of module interface INOUT, even though SORTER just requires an arbitrary INOUT implementation.

KERIS offers a facility to constrain the interface of a submodule M to a single implemented module interface I. Syntactically this is expressed with the following submodule declaration form: `module` M `implements` I. Only members of the designated module interface I may be accessed by the host module or other submodules. This artificial restriction of a module implementation to one of its interfaces is called *interface ascription*.[5] It allows a submodule to be overridden with a module that implements this interface but which is not necessarily a refinement. Here is a reformulation of module APP which makes it possible to replace CONSOLE with an alternative INOUT implementation.

[5]The term *interface ascription* was chosen following the module system terminology of ML which supports a similar concept called *signature ascription*.

```
module APP {
  module SORTER;
  module CONSOLE implements INOUT;
  ...
}
```

```
module XXAPP refines APP {
  module GUI implements INOUT;
}
module GUI implements INOUT {
  ...
}
```

Discussion. The previous examples demonstrate that the module assembly and refinement mechanism is not only restricted to the extension of atomic modules. It also allows programmers to extend fully linked programs, represented by modules with aggregated submodules, by simply replacing selected submodules with compatible versions. There is no need to establish module interconnections again; the fully linked program structure is reused and only the submodules and functions to replace or add have to be specified.

This extensibility mechanism features *plug-and-play* programming. It neither requires source code, nor touches existing binaries. After having refined application APP with module XAPP it is still possible to run the old application APP. It would even be possible to assemble a system that makes use of both modules without risking unpredictable interferences.

3.2.6 Specializing Modules

Refinements vs. specializations. *Refining* a module is the process of extending a module by adding new functionality or by modifying existing functionality through overriding. A module refinement yields a new version of an existing module. This new version subsumes the old one; i.e. it is backward compatible to the old version. As a consequence, it is always possible to replace a module with one of its refinements.

In the following code, module BUGGYMOD aggregates a submodule that subsumes another submodule. In other words, BUGGYMOD defines a context in which two different versions of one module are present. Since references to submodule SORTER are ambiguous within module BUGGYMOD, the program is ill-formed and rejected by the Keris compiler.

```
module BUGGYMOD requires INOUT {
  module SORTER;
  module FASTSORTER;  ↯
}
```

As Section 3.3.2 will motivate, we would sometimes like to reuse a general module implementation when defining a new module with a a more specialized structure and functionality and use different specializations side by side in the

same context. This process of creating new distinct modules which reuse the definition of an existing module is called *module specialization*. It is similar to module refinement in that it is based on inheritance; it is different, because specializations yield new modules compared to refinements which only create new versions of existing modules. While refinements are backward compatible to their original module and therefore can safely replace original module instances, specializations represent new, independent modules which do not subsume their original module. This is also why different specializations may be used jointly in the same context.

Creating specializations. As an example, we define a specialization of the SORTER module in the following code. Module SETSORTER implements a set semantics for sorting and is due to this change in semantics not implemented as a refinement of SORTER.

```
module SETSORTER specializes SORTER {
  String[] filterDuplicates(String[] list) { ... }
  String[] sort(String[] set) {
    return super.sort(filterDuplicates(set));
  }
}
```

Module SETSORTER inherits all members and requirements from SORTER and defines two new functions. Function `filterDuplicates` can be used to filter out duplicate entries in lists. Function `sort` overrides the corresponding function in SORTER, but its implementation is still able to refer to the former implementation via the keyword `super`.

As a specialization of SORTER, module SETSORTER is not required to be backward compatible to module SORTER. One of the consequences is that module specializations in general are free to break invariants or contracts established by their parent module without risking erroneous deployments. Such risks get ruled out technically because module specializations do neither subsume their parent module, nor do they automatically implement any of the module interfaces that are implemented by their parent module. Therefore, it is not possible to substitute an instance of the parent module with a module instance of one of its specializations. This restriction turns SORTER and SETSORTER into completely different modules. Moreover it is perfectly legal to define a module with both a SETSORTER and a SORTER submodule, like in the following program.

```
module SORTING requires INOUT {
  module SORTER;
  module SETSORTER;
}
```

While module refinements promote the *substitutability* of modules, module specializations support the notion of *conceptual abstraction* on the module level [174]. Conceptual abstraction refers to the ability to factor out code and structure shared by several modules into a common parent module which gets specialized independently into different directions. The specializations represent distinct modules that cannot be substituted for the common parent module [52].

Rewiring modules. Often, mutual referential modules have to be specialized at the same time consistently. The ability to refer to a specialized version of a module requires that we are able to specialize context dependencies as well. This "rewiring" is expressed in the following code using the as operator. The as operator allows programmers to replace a module with one of its specializations. In the following code, the MYSORTER module specializes module SORTER and instead of requiring the original INOUT module, it now refers to a specialized MYINOUT module instance.

```
module MYSORTER specializes SORTER requires MYINOUT as INOUT {
  ...
}
```

Of course it is also possible to specialize submodule definitions if required. We could, for instance, specialize module APP of Section 3.2.2 and use MYSORTER instead of SORTER. Note that for doing this correctly, we also have to specialize module CONSOLE, otherwise we would break a link established in the parent module. Here, module SORTER's required INOUT module is wired to submodule CONSOLE which implements INOUT.

```
module MYAPP specializes APP {
  module MYSORTER as SORTER;
  module MYCONSOLE as CONSOLE;
}
module MYCONSOLE specializes CONSOLE implements MYINOUT {
  ...
}
```

This example shows that module specializations cannot break arbitrary contracts established in parent modules. To ensure type safety, modules linked in parent modules still have to be linked in module specializations.

A more complete example for module specializations and the rewiring of modules by specializing context dependencies and submodule definitions can be found in Section 3.3.2 on page 90.

3.2.7 Class Abstractions

Until now we only considered modules with function and variable members. With these modules, JAVA's static variables and static methods get superfluous. Static class members can easily be implemented as module members with the benefit of extensibility and improved reusability. Even though functions on the module level can be quite useful to model global behavior, it is more common for object-oriented languages to have modules that contain class definitions. Classes defined in a module can freely refer to other members of the module as well as to modules required from the enclosing module. The following module defines a class for representing points.

```
module SPACE {
  class Point {
    int x, y;
    Point(int x, int y) {
      this.x = x;
      this.y = y;
    }
    int getX() { return x; }
    int getY() { return y; }
  }
}
```

Module systems for JAVA-like programming languages that allow to abstract over classes are not only difficult to handle in theory, they are also extremely difficult to implement in practice if one wants to stick to JAVA's compilation model. In such module systems, classes can, for instance, extend classes of required modules for which only the interface might be given. Consequently, at compile-time, a compiler has to translate the class without knowing its concrete superclass. This raises implementation issues, but also more fundamental questions, e.g. about the possibility to create cycles in the inheritance graph or about methods that override superclass methods accidentally. Ancona and Zucca discuss problems related to this trade-off between class abstraction and implementation inheritance in greater detail in [11].

Separating interfaces from implementations. Since KERIS is designed to be fully compatible with JAVA, including full support for JAVA's compilation model, while being implementable on the standard JAVA platform, KERIS does not offer a facility for abstracting over regular classes. Instead, it introduces the notion of *class fields* as an alternative type and class definition and extension facility. The design philosophy of KERIS follows one of the most important features of module systems, which is the separation of interfaces from implementations, and regards regular classes as implementations of abstract data types. Interfaces, on the other hand, are regarded as nominal signatures of abstract data types. Class fields are

abstractions that connect interfaces with implementations. As opposed to classes and interfaces which are considered to be static and immutable, Keris allows one to abstract over class fields. Furthermore, class fields are extensible; they can be covariantly overridden in module refinements and specializations. The rest of this section will discuss Keris' class abstractions in detail.

Class fields. In most object-oriented languages class definitions introduce many entities at the same time. They define a type, a default implementation of that type, and a constructor which creates new instances of this implementation. The notion of inheritance allows types and class implementations to be reused in the definition of new types and new class implementations, but the original entities itself are usually immutable at the language level. So changes to these entities have to be carried out destructively, possibly raising consistency issues.

In support of extensible class abstractions, Keris introduces the notion of *class fields*. A class field is a class abstraction which separately defines an interface and an implementation. While the interface specification typically refers to a set of regular Java interfaces, the class field implementation consists generally in a reference to a regular class.

The following example code defines an interface, a class, and a class field within a single module `POINTS`. The definition of interface `IPoint` within module `POINTS` shows that interfaces in Keris can also specify the signature of constructors, in contrast to regular interfaces in Java where this is not possible. In addition to interface `IPoint`, module `POINTS` declares a class `CPoint` which implements `IPoint` and therefore can be used to represent concrete point objects. Finally, a class field `Point` is defined by separately specifying its interface and implementation.

```
module POINTS requires INOUT {
  interface IPoint {
    IPoint(int x, int y);
    int getX();
    int getY();
    Point move(int dx, int dy);
  }
  class CPoint implements IPoint {
    int x, y;
    CPoint(int x, int y) { this.x = x; this.y = y; }
    int getX() { return x; }
    int getY() { return y; }
    Point move(int dx, int dy) { return new Point(x + dx, y + dy); }
  }
  class Point implements IPoint = CPoint;
  Point root() { return new Point(0, 0); }
  void print(Point p) { INOUT.write(p.getX() + "/" + p.getY()); }
}
```

In general, the definition `class` T `implements` $I_1, ..., I_n = C$ defines a class field T which implements the interfaces I_1 to I_n with class C. More precisely, each class field declaration introduces a new nominal type which is a subtype of all the implemented interfaces $I_1, ..., I_n$. Furthermore, it specifies a default implementation C which is used to represent instances of the new type at runtime.

The implementations of the functions `print` and `root` show that class fields are used just like regular classes: They denote types, they can be instantiated, and members of corresponding objects can be accessed. The main difference to regular classes is that class fields are virtual and therefore can be covariantly overridden in refined modules. Covariant overriding of class fields includes the extension of the set of implemented interfaces as well as the ability to specify new class field implementations. On the other hand, class fields do not support implementation inheritance — a feature which is only supported by regular classes. Therefore, class fields must be rather seen as abstractions that complement classes than abstractions that fully replace them.

Overriding. The next example program explains how class fields can be overridden in module refinements. In this program, refinement `COLORPOINTS` declares that class field `Point` now also supports the `IColor` interface and is implemented by the `CColPoint` class. Furthermore, `print` is overridden to include the color in the output. At this point, one might wonder what happens to method `root` of the original module `POINTS` which instantiates class field `Point`. In fact, for the refined module which inherits this method, `root` now returns a colored point since class field `Point` is overridden in the refinement.

```
module COLORPOINTS refines POINTS requires COLOR {
  interface IColor {
    void setColor(COLOR.Color col);
    COLOR.Color getColor();
  }
  class CColPoint extends CPoint implements IColor {
    COLOR.Color col = COLOR.black;
    CColPoint(int x, int y) { super(x, y); }
    void setColor(COLOR.Color col) {
      this.col = col;
    }
    COLOR.Color getColor() {
      return col;
    }
  }
  class Point implements IPoint, IColor = CColPoint;
  void print(Point p) {
    super.print(p);
    INOUT.write("␣col␣=␣" + p.getColor());
  }
}
```

Type refinement. The ability to covariantly refine types (or class fields in our case) is essential for extending object-oriented software. Most object-oriented languages support interface and implementation inheritance. But inheritance alone does not support software refinement or software specialization well. Existing code refers to the former type and often cannot be overridden covariantly in a type-safe way to make use of the extended features. For special cases like binary methods, some languages support the notion of *self types* [36, 35, 151]. But these are not suitable for mutually referential classes that have to be refined together to ensure consistency [59]. Here, only virtual types are expressive enough [96, 199, 58, 124]. Unfortunately, virtual types rely in general on dynamic type-checking. Therefore recent work concentrated on restricting the mechanism to achieve static type safety [200, 34]. A formal account of type-safe virtual types is given in [152], which introduces a calculus of classes and objects with *abstract type* members.

Class fields are statically type-safe in Keris. This is mainly due to the nature of module refinements: A refined module subsumes the former module and cannot coexist with the former module in the same context. It rather replaces the former module consistently in explicitly specified contexts. Module specializations do not compromise type safety either, since they conceptually yield new modules with class fields that do not necessarily have a (subtype) relationship with the original class fields in the parent module.

The explicit separation of interface and implementation definitions does not only promote modular programming, it also helps enormously to evolve software more flexibly, because modifications on the level of interfaces do not impose anything on the implementation level, and vice versa. It is possible to safely extend the interface of a class field without changing its implementation (e.g. to expose more functionality to clients), but it is also possible to extend or even fully exchange the class field implementation without modifying anything on the interface level.

As we will discuss in Section 3.4.6, this strict separation of interfaces from implementations also eases hot swapping of modules which might cause a dynamic change of class field implementations.

Opaque class fields. It is possible to leave out an implementing class when defining a class field. Class fields without implementations are called *opaque*. They are similar to abstract methods which only define a signature and defer the concrete implementation. Modules with opaque class fields or abstract functions have to be declared `abstract`. *Abstract modules* cannot be deployed and their only function is to serve as a parent module for refinements and specializations. In this sense, module interfaces are abstract by default. Opaque class fields are mainly used within module interfaces to define new class types. To illustrate this, we reformulate the `POINTS` example by cleanly separating the module interface from its implementation. This example also nicely shows that by separating the

module interface from the implementation, we can express that the POINTS abstraction does not rely on an INOUT module itself. It is, in fact, only the concrete implementation that has this context dependency.

```
module interface POINTS {
  interface IPoint {
    IPoint(int x, int y);
    int getX();
    int getY();
    Point move(int dx, int dy);
  }
  class Point implements IPoint;
  Point root();
  void print(Point p);
}
module SIMPLEPOINTS implements POINTS requires INOUT {
  class Point implements IPoint = {
    Point(int x, int y) { ... }
    int getX() { ... }
    int getY() { ... }
    Point move(int dx, int dy) {
      return (dy == 0) && (dy == 0) ?
                this : new Point(x+dx, y+dy);
    }
  }
  Point root() {
    return new Point(0, 0);
  }
  void print(Point p) {
    INOUT.write(p.getX() +"/"+ p.getY());
  }
}
```

Anonymous class field implementations. In module SIMPLEPOINTS we use an *anonymous class declaration* to provide a concrete implementation for class field Point. An anonymous class declaration consists of a block defining class members. This block can optionally be preceded by a reference to a super class. The use of anonymous classes is sometimes necessary to give the self reference this the right type. In anonymous classes, this is given the type of the corresponding class field. In the example above, it is therefore possible for method move of class field Point to simply return this if both its parameters are zero. This optimization would have not been possible in our former implementation of module POINTS.

Note that in selections of the form this.methodOrVariable we type the self reference this with the anonymous implementation type so that one is still able to access private fields and methods.

Abstract class fields. Class fields can also be tagged with the `abstract` modifier. Like abstract classes, such class fields cannot be instantiated. Their main aim is to define extensible types. Since abstract class fields cannot be instantiated, there is also no need to specify an implementation. Our terminology follows the one of Java which is unfortunately inconsistent regarding the use of modifier `abstract`: Abstract classes possibly define an implementation, whereas abstract methods never provide a concrete implementation.

The next paragraph will present an example which motivates the use of abstract class fields in Keris. This example refers to class field `Shape` defined in the following declaration of module `GEO`.

```
module GEO requires POINTS {
  import POINTS.*;
  interface IShape {
    boolean inShape(Point pt);
  }
  abstract class Shape implements IShape;
  void registerShape(Shape s) {
    registeredShapes.add(s);
  }
  HashMap registeredShapes = new HashMap();
}
```

Note that this module does not have to be declared `abstract`. It has indeed a class field without an implementation, but since this class field cannot be instantiated, we can safely omit the `abstract` modifier in the module definition.

Dependencies. So far we explained how to declare and how to evolve class fields. The presented mechanism does not allow one to relate different class fields to each other; every class field represents an own isolated class abstraction. We need a mechanism similar to subclassing (on the implementation level) and subtyping (on the interface level) that makes it possible to introduce dependencies between class fields. For this reason, it is possible in Keris to specify subtype relationships between otherwise unrelated class fields explicitly. The following code illustrates this feature.

```
module SHAPES requires GEO, POINTS {
  import GEO.*;
  import POINTS.*;
  interface IBox {
    IBox(Point topleft, Point botright);
  }
  class Box extends Shape implements IShape, IBox = {
    Box(Point topleft, Point botright) { GEO.registerShape(this); ...}
    boolean inShape(Point pt) { ... }
  }
}
```

In this program we refer to module GEO defined in the previous paragraph. GEO defines an abstract class field Shape. As we mentioned already before, abstract class fields are like abstract classes: They cannot be instantiated but they define a type which can be extended. Module SHAPES defines a class field Box which extends Shape. This *extends* declaration turns Box into a subtype of Shape requiring that Box implements at least all the interfaces implemented by Shape. The type checker has to make sure that it is not possible to link refinements of GEO and SHAPES where this invariant is broken. Thus, such subtype dependencies between class fields promote the consistent refinement or specialization of class field hierarchies.

Here is an example which successfully links modules GEO and SHAPES in the context of module GEOSHAPES.

```
module GEOSHAPES requires POINTS {
  module GEO;
  module SHAPES;
}
```

Imagine, we develop a refinement XGEO of GEO that adds a new method scale to Shape:

```
module XGEO refines GEO {
  interface INewShape { void scale(int factor); }
  abstract class Shape implements IShape, INewShape;
}
```

We now cannot simply refine GEOSHAPES and introduce the refined XGEO module in place of the previous GEO module. This would break our dependency, since SHAPES.Box now would not cover the newly introduced XGEO.INewShape interface. Thus, we first have to consistently refine SHAPES as well, such that class field Box also implements the new interface XGEO.INewShape. This refinement can then be linked with XGEO, as the following code fragment shows. The concrete implementation of XSHAPES.Box is irrelevant and therefore left out in the following program.

```
module XSHAPES refines SHAPES requires XGEO {
  class Box extends GEO.Shape implements GEO.IShape,XGEO.INewShape =...
}
module XGEOSHAPES refines GEO {
  module XGEO;
  module XSHAPES;
}
```

Note that extends declarations have no implications on the implementation side. For our example this means that Box can be implemented by an arbitrary

class which can be a subclass of the implementation of Shape, but which is not required to be one.

It is possible for two or more class fields to depend mutually on each other (yielding a cycle in the extension relationship). In such a case, all recursively dependent class fields have to implement the same interfaces — this is a consequence of our requirement that a class field that extends another class field has to implement all the interfaces of this other class field. Since subtyping is antisymmetric in Keris, mutually dependent class fields denote the same type. However, they can still have different implementations.

In contrast to this, classes (i.e. class field implementations) many not introduce cycles in their inheritance hierarchy. More precisely, a class may not extend other classes that extend already the first class, no matter with what module path the classes are prefixed.

Inheritance and subtyping. Inheritance is a powerful reuse mechanism which is in most statically typed object-oriented languages coupled with subtyping. Sometimes this is unfortunate if one wants to reuse a class without establishing a *is-a* relationship. For instance, many introductory texts to object-oriented programming motivate inheritance with an example where class `Circle` inherits from class `Point`. While on the level of code reuse, this makes perfectly sense, on the level of subtyping, this relationship is questionable: Intuitively, circles are not really specialized points.

In Keris it is possible to express the intention of reusing an implementation without implying anything on the typing side. The following module introduces a class field `Circle` which is a subtype of Shape but which is implemented by a an anonymous subclass of `CPoint`. In other words, we implement a new subtype of class field Shape by reusing class `CPoint` which is completely unrelated to Shape.

```
module MYSHAPES requires GEO, POINTS {
  import GEO.*;
  import POINTS.*;
  interface ICircle { ICircle(int x, int y, int r); }
  class Circle extends Shape implements IShape, ICircle = CPoint {
    int radius;
    Circle(int x, int y, int r) { super(x, y); radius = r; }
    boolean inShape(Point pt) { ... }
  }
}
```

From the viewpoint of static typing, class `CPoint` is also unrelated to `Circle`. The anonymous subclass of `CPoint` acts as the implementation of class field `Circle` so that at runtime, instances of `Circle` are actually instances of the anonymous subclass of `CPoint`. Thus, dynamically, there is a strong relation between the

two.[6] This relationship is hidden statically to promote extensibility by allowing programmers to exchange implementations easily with compatible ones.

Opposed to this example where subtyping is decoupled from inheritance, KERIS also supports the more common approach where types and implementations are refined simultaneously, as the following program will exemplify.

```
module NUSHAPES requires GEO, POINTS, MYSHAPES {
  import GEO.*;
  import POINTS.*;
  import MYSHAPES.*;
  interface IRing {
    IRing(int x, int y, int ro, int ri);
  }
  class Ring extends Circle implements IShape,ICircle,IRing = Circle {
    int innerRadius;
    Ring(int x, int y, int ro, int ri) {
      super(x, y, ro); innerRadius = ri;
    }
    Ring(int x, int y, int ro) {
      super(x, y, ro); innerRadius = 0;
    }
    boolean inShape(Point pt) {
      ... super.inShape(pt) ...
    }
  }
}
```

In this program, we define a class field `Ring` which depends on the class field `Circle`.[7] It is implemented by an anonymous class that inherits from the implementation of class field `Circle`. Two things have to be noted:

1. When a superclass of an anonymous class field implementation refers to another class field, this anonymous class actually inherits from the implementation of this other class field.
2. It is only possible to inherit from the implementation of another class field if there is also a dependency on the typing side.

Since types defined by regular classes are never subtypes of class field types, the second restriction also excludes that regular classes inherit from class field implementations.

[6] Dynamically, there is no difference between a class field and the class field implementation. Therefore it is no coincidence that KERIS uses the = letter to specify class field implementations

[7] This example shows that it is not always a good idea to put constructors into regular interfaces. A consequence is that dependent class fields have to implement such constructors as well. This can make sense, but often it does not. Therefore KERIS allows interfaces to be tagged with the modifier `constructor`. Such constructor interfaces are collections of constructor signatures. They do not define a type and therefore do not have to be supported by depending class fields.

The second restriction is necessary for type safety reasons. Since the self reference inside of an anonymous class field implementation is typed with the corresponding class field type, it has to be guaranteed that it can only be subclassed by classes where the self reference is assigned a subtype. Otherwise we could refine the original class field independently breaking our new class field implementation (where the self reference is typed with the refined class field type).

In the implementation of class field `Circle` in module `MYSHAPES` we do not get this problem, because class `CPoint`, from which the anonymous class field implementation inherits, is immutable — it cannot be refined. Nevertheless, it is important to understand what happens if class `CPoint` exposes its self reference in some way as an object of type `CPoint`. Since this code is inherited to the implementation class of `Circle`, there exists the possibility that at runtime, an object which is statically typed as a `Circle`, is used as a `CPoint`. This does not endanger type safety, since `Circle` objects are anyway instances of a `CPoint` subclass, but when used as a `CPoint` it allows access to fields and methods that might not explicitly be specified in the implemented class field interfaces.

Here one can see that interfaces given in class field definitions act as views on the actual class field implementations such that clients of a class field object can only access the specified class field members. Other parts of the system have possibly a different view with different access rights.

3.2.8 Type System

In this section we informally review the basic principles of the type system of Keris. We explain what restrictions have to be made to ensure statically that a system assembled from modules is sound. Furthermore we explain how the type system helps to evolve software consistently.

Types. Type systems of Java-like object-oriented languages are usually *nominal*; i.e. types are identified by their names, not by their structure. Not considering the package system and class nesting, reference types in Java have simply the form C, where C corresponds to a class name. In Keris, on the other hand, classes are typically not defined on the top-level, but rather within modules. Since modules can aggregate submodules arbitrarily, a reference type in Keris corresponds to a class name C which is qualified with a sequence of module names $M_1 :: M_2 :: ... :: M_n$ where M_{i+1} is a submodule of M_i for all i and C is a member of module M_n. This module name sequence, also called *module path*, identifies uniquely a nested instance of module M_n. Thus, types in Keris have the general form $M_1 :: M_2 :: ... :: M_n.C$. Two types are equal, if the class names are equivalent and if the module path qualifications refer to the same module instance. In a given context, two module paths $M_1 :: M_2 :: ... :: M_m$ and $N_1 :: M_2 :: ... :: N_n$ refer to the same module instance if $m = n$ and M_i is equiva-

lent to N_i for all i.[8] Module equivalence is considered modulo refinements and module implementations (which can be seen as a special case of module refinement), so that all the following types would be considered equal for the modules defined in the paragraph about class field dependencies of Section 3.2.7: `XGEOSHAPES::GEO.Shape`, `GEOSHAPES::GEO.Shape`, and `XGEOSHAPES::XGEO.Shape`. Since types depend on module instances, our system distinguishes between the types `O::M.C` and `O::N::M.C` in the following listing. Both types refer to the same physical class C, but qualified with different module instances. This distinction is necessary since it is possible to refine both instances of M independently, possibly yielding versions of class field $M.C$ with different interfaces and implementations. Even if $M.C$ does not refer to a class field, we have to consider the module instance qualifications, since $M.C$ itself might refer to a class field $M.D$ which possibly evolves differently for different module instances.

```
module M {
  class C = {}
}
```

```
module N {
  module M;
}
```

```
module O {
  module M;
  module N;
}
```

Such a *dependent type system* is characteristic for strongly typed programming languages supporting *family polymorphism* [59] (see Section 3.3.1 for a discussion). For technical details we refer to Odersky, Cremet, Röckl, and Zenger's formalization of dependent types for class-based object-oriented languages [153, 152].

Type coherence. The previous section discussed when types are considered to be equivalent. This question is in particular relevant for types appearing in the interface of external modules. Consider the following example in which module interface `A` defines a class field `C`. We have two modules `M` and `N` which both require a module implementing `A`. Both of these modules are required by a third module `O`. The question is now, whether the type `A.C` mentioned in `M` and the type `A.C` mentioned in `N` are equivalent in the context of module `O`. This would turn `N.bar(M.foo())` into a well-typed expression.

```
module interface A {
  interface I { I(); }
  class C implements I;
}
module O requires M, N {
  ... N.bar(M.foo()) ...  ?
}
```

```
module M requires A {
  A.C foo() { return new A.C(); }
}
module N requires A {
  void bar(A.C a) {}
}
```

[8] This notion of module path equality is a conservative approximation which guarantees at compile-time that two module paths refer to the same module instance at runtime. Of course, there might be module paths of different shape referring to the same module instance dynamically.

In ML-like module systems it is up to the programmer to declare that both types are considered to be equivalent by explicitly introducing *sharing constraints*. This *sharing by specification* approach allows the programmer to introduce only a minimal set of equations identifying types. On the other hand, if a large set of interconnected cooperating modules with many type members is used, it becomes quickly unhandy to specify all the required sharing constraints by hand.

For a language without generic context dependencies like Java, the coherence problem is trivial. Two fully qualified type references $P.C$ are always considered to be equivalent, no matter what context they appear in, since there will be always exactly one implementation available at runtime. We consider this implicit "agreement" on types (or, in fact, packages) essential for making the development of huge, recursively dependent, object-oriented libraries practical. Thus, Keris adopts a similar policy and rigorously identifies types with the same name and equivalent module sequence qualification, even for types appearing in required, external modules, and without giving up genericity. Since type equivalence in Keris is dependent on a compile-time notion of equivalence of module instances, it is essential for the type system of Keris to enforce that modules which get identified statically in the context of a module like `O` are actually also implemented at runtime by a single module implementation for all possible module composition scenarios involving this module.

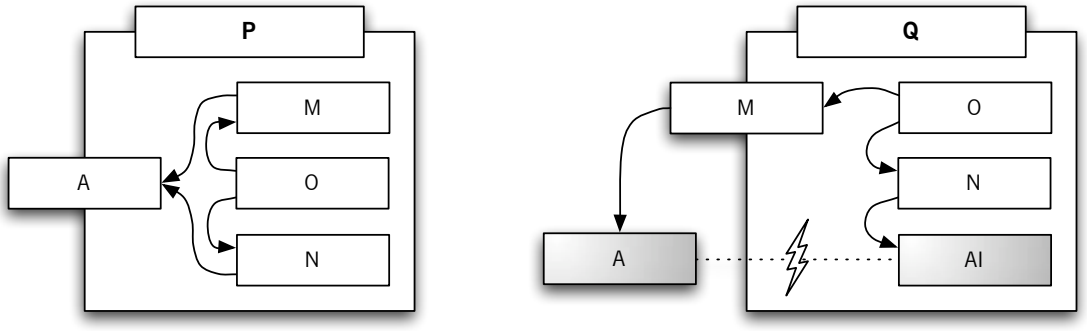
Here is an example for a legal composition of modules `M`, `N`, and `O`. Module `P` is well-typed, because both modules `M` and `N` are referring to the same implementation of module `A` (which gets required by `P`). This is necessary since the context of the third submodule `O` identifies the `A` instances seen by `M` and `N`. The dependencies in this program are illustrated in Figure 3.3.

```
module P requires A {
  module M;
  module N;
  module O;
}
```

Under the assumption that module `AI` implements module interface `A`, the following definition of module `Q` is not well-typed. This is because now, module `N` gets linked with `AI`, whereas module `M` is required from `Q` and therefore can never refer to implementation `Q.AI` for satisfying its own context dependency on `A`. Details about the dependencies are shown in Figure 3.3.

```
module Q requires M {
  module AI;
  module N;
  module O;   ↯
}
```

```
module AI implements A {
  ...
}
```



(a) Legal composition (b) Illegal composition

Figure 3.3: Illustration of modules P and Q.

Note that the ill-formedness of Q is solely due to the definition of submodule O, because only O requires that modules M and N both access the same implementation for A. Omitting O in the example above would turn Q into a well-typed module.

In general, for submodules M of N, the KERIS compiler checks the following to ensure a coherent use of types:

- The *resolved context dependencies* of M in N, written resolved(M,N), is the smallest set of modules with $M \in$ resolved(M,N) where $O \in$ resolved(M,N) implies that all required modules of O that are submodules of N are also contained in resolved(M,N).
- The *unresolved context dependencies* of M in N, written unresolved(M,N), is the smallest set of modules which contains all modules required by both M and N and where $O \in$ unresolved(M,N) implies that all modules required by O are also contained in unresolved(M,N).
- The set of resolved context dependencies and the set of unresolved context dependencies have to be disjunct; i.e. resolved$(M,N) \cap$ unresolved$(M,N) = \emptyset$.

Together with the invariants for submodule definitions discussed in the next paragraph, this rule guarantees that modules identified in a submodule context do also get identified in the deployment context. For the example above we have resolved$(O,Q) = \{N, AI\}$ and unresolved$(O,Q) = \{M, A\}$. Since A is included in both sets (again modulo refinements and implementations of module interfaces), we have a violation of the rule above.

Regarding the mechanism for handling type coherence, Keris trades simplicity for expressiveness. The previous example demonstrates that the explicit identification of modules can indeed restrict the number of possible deployment scenarios for a given module artificially if external modules get identified unnecessarily. But our experience shows that by specifying minimal views (module interfaces) for external modules, one can often avoid unintended module iden-

tifications and improve reusability significantly. Furthermore, a simple mechanism for handling type coherence is essential in practice, since in the presence of software evolution features, complex coherence management facilities get even more complex, ultimately yielding unmanageable formalisms.

Module composition. Following traditional module systems of imperative languages, Keris associates with every module name implicitly a module interface. This is the basis for inferring the wiring of submodules. *Wiring inference* on the other hand requires that references to modules have to be unambiguous; i.e. the type system has to ensure that a module name identifies an implementing module instance uniquely. We impose a set of restrictions on submodule aggregations to enforce this.

The following explanation of the relevant restrictions is based on the notion of a module *subsuming* other modules. A module A subsumes another module B, if A either refines B (directly or indirectly), or if A implements module interface B, or if there is a module C which gets refined by A and which implements module interface B.

Based on this definition, we can now formulate the restrictions on submodule aggregations enforced by the Keris type system. The aggregation of a submodule M in a host module N is subject to the following terms:

- N may not define another submodule, or require a module which subsumes modules that are subsumed by M *(uniqueness)*.
- M or any direct or indirect submodule of M may not subsume modules that are subsumed by host module N *(linearity)*.
- All of M's required modules have to be present either as other submodules of N, or have to be required by N, or have to be subsumed by N itself *(dependency satisfaction)*.
- N has to identify at least those module instances identified by submodule M; i.e. Keris' compile-time notion of module equivalence, discussed in the last paragraph, has to be used in a coherent way *(coherence)*.
- Class field dependencies specified by M or any direct or indirect submodule of M involving modules from M's deployment context (i.e. required modules of M) have to also hold for the resolved concrete context dependencies in N *(consistency)*.

While the uniqueness and linearity restrictions rule out ambiguities, it is the dependency satisfaction rule which guarantees that all requirements of submodules are met by the deployment context. The coherence rule addresses type equivalence issues discussed in the previous paragraph. The consistency rule is responsible for validating class field dependencies in a concrete deployment context. The following program is an example for an illegal aggregation of a submodule `M` in `N` due to a violation of the consistency rule.

```
module interface A {
  interface I { void foo(); }
  class C implements I;
}
module AI implements A {
  interface J { void bar(); }
  class C implements I, J = {
    ...
  }
}
```

```
module M requires A {
  class D extends A.C
          implements A.I = {
    ...
  }
}
module N {
  module AI;
  module M;    ↯
}
```

The module definition of M is well-formed because class field D implements all interfaces implemented by A.C. As a submodule of N, module M would get linked with the other submodule AI. But since AI refines C covariantly by implementing an additional interface J, we cannot successfully link it with M who's class field C does not support J.

Note that the linearity restriction from the list above goes beyond the minimal requirements for avoiding ambiguities. The restriction also rules out cases with indirect recursion. According to the rules above, it is illegal for a module to aggregate an instance of itself even indirectly, as module M is doing in the following program.

```
module M {
  module N;
  class A extends N.B = {}
}
```

```
module N {
  module M;
  class B extends M.A = {}
}
```

Since KERIS initializes modules lazily at the time a module gets accessed for the first time, such a program would not necessarily create an infinite number of nested module instances. But as the program above shows, it can introduce class field types that are subtypes of infinitely many supertypes. For this reason and the fact that a program with recursively nested submodules does not correspond to a finite system with a finite architecture, recursively nested modules are rejected by the type system.

Interface ascription. The interface ascription mechanism of KERIS allows programmers to constrain the interface of a submodule implementation M to a single implemented module interface I. To guarantee that such submodules are deployed and overridden consistently, KERIS restricts this mechanism with the following rule:

> If $\overline{N}$ denotes the closure of the requirements of M and $\overline{J}$ is the set of all interfaces implemented by M, then M can be constrained to $I \in \overline{J}$ only if $\overline{N} \cap \left(\overline{J} \setminus \{I\}\right) = \varnothing$ (modulo module refinement and implementation).

This rule verifies that all modules that are directly or indirectly required by M do not refer back to M or one of the interfaces implemented by M which are not captured by I. In the definition of M such a scenario would be interpreted as a recursive dependency, while in the context of a submodule ascription to I, this invariant could never hold.

Module refinement. Module refinements have to be backward compatible so that an instance of a refinement can safely replace an instance of the original module. Consequently, invariants established by the parent module are not allowed to be broken in refinements.

Module refinements inherit all required modules and all submodules from their parent module. It is possible to add new requirements or to refine requirements to express a simultaneous mutual refinement of several modules. Furthermore, one can covariantly override submodules with instances of refined modules. Like in Java, it is not possible to override variables. Method overriding is invariant in the arguments and covariant in the return type, similar to GJ [32].

Module specialization. A module specialization yields a new distinct module which inherits members from an existing "prototype." Excluded from inheritance are the implemented module interfaces. Integrating specialized modules into existing systems requires a mechanism for replacing a module with a specialized version. This *rewiring* is possible for the requirements and the submodules of specialized modules, as explained in Section 3.2.6. The rewiring of requirements is essential to express that a set of modules is specialized together. The type checker has to check that depending modules are specialized consistently. Here is an example for an inconsistent specialization:

```
module M {}
module N requires M {}
module O requires M, N {}
module M1 specializes M {}
module N1 specializes N requires M1 as M {}
module O1 specializes O requires N1 as N {}  ↯
```

We define three modules `M`, `N`, and `O`, where module `N` depends on `M` and module `O` depends on `M` and `N`. Next we specialize `O` to `O1` and `N1` to `N` where the specialized module `N1` now refers to `M1` instead of `M`. The specialization of `O` rewires the original requirement `N` to `N1` and inherits the old requirement `M`. This is a violation of an invariant established in the original module `O`. Here, module `O` and depending module `N` agree on a single implementation of module `M`, according to the coherence discussion in Section 3.2.8. In the specialization `O1` on the other hand, there is no agreement on `M` anymore, since the rewired dependency `N1` now refers to `M1` instead of `M`. A consistent specialization of `O` rewires both `N` to `N1` and `M` to `M1`. The scenario is illustrated in Figure 3.4.

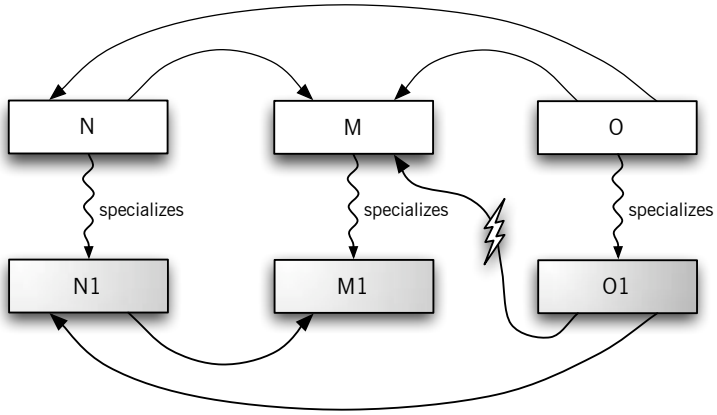


Figure 3.4: Illustration of module dependencies in the original and specialized system.

For checking the consistent specialization of a module, the type checker first has to collect all specializations of requirements and submodules. In a second step, it has to verify that the "agreement" on modules in the old system still persists in the rewired system (possibly involving instances of specialized modules). This check can be done modularly, but it involves all interfaces of modules contained in the closure of the old and the new module's requirements.

This example and the examples given in the previous sections demonstrate how the type system can help to evolve systems consistently by enforcing invariants established by the original system in refined or specialized variants. Without explicit support for software evolution on the programming language level, programmers are often required to check compatibility and consistency between different versions at runtime or with special tools at link-time.

3.2.9 Runtime Types and Reflection

3.2.9.1 Runtime Types

In JAVA it is possible to query the exact runtime type for any given object. On the language level, the type operator `instanceof` exploits this feature to determine if the dynamic type of an object is a subtype of a statically specified type. Similarly, type casts have to verify the type and fail if necessary. Types at runtime are also used in JAVA to close the hole in the static type system dynamically, which is caused by covariant array subtyping.

Support for runtime typing in Java. For KERIS we cannot directly rely on JAVA's native support for runtime type information due to its more expressive type system. The following KERIS program illustrates what problems dependent types are posing when implemented on a platform that only supports purely (non-dependent) nominal types.

```
module M {
  class C {}
}
module N {
  module M;
}
module O {
  module M;
}
```

```
module P {
  module N;
  module O;
  void main(String[] args) {
    Object obj = new N::M.C();
    System.out.println(
      obj instanceof O::M.C);
  }
}
```

We have two modules N and O which both aggregate an instance of module M. In the main function of module P we first create an object of type N::M.C and determine then if the runtime type of this object corresponds to O::M.C. According to the explanations of Section 3.2.8, the two types are different in the context of module P, even though we physically refer to the same class C defined in module M. Consequently, we should expect that our runtime type check in module P will fail even if objects of type O::M.C and N::M.C are represented at runtime as instances of a single JAVA class M.C.

Dynamic transparency. The implementation of KERIS has full support for runtime type checks according to the static type system of the language. It also handles type casts in a safe way. Note that in general, class fields are not dynamically opaque; i.e. at runtime it is possible to determine what concrete class is used as the implementation of a class field. Leaving class field implementations transparent at runtime is required because of the possibility that class field implementation objects might export themselves as objects of the implementation type as described in Section 3.2.7 on page 68 in the paragraph about inheritance and subtyping. An advantage of this implementation transparency is also that when modules are hot swapped (see next section), it is possible to dynamically distinguish old implementations from new ones.

3.2.9.2 Reflection

KERIS has a built-in reification system for modules and module instances that exposes meta-information and internal configuration data. Furthermore, there is support for changing the configuration of modules dynamically. In combination, these two features allow the development of powerful reflective libraries that are able to introspect, modify, and even assemble module-based systems dynamically. To allow the coexistence of different reflective libraries which possibly use different module composition techniques, the implementation of KERIS does not rely on a specific reflection API. It strictly separates the (possibly user supplied) reflective system from the implementation of the language. The code supporting reflection which gets generated by the KERIS compiler for every module definition does not provide any means of security which could ensure that

systems which are assembled or modified dynamically do not violate static invariants of the module system of KERIS. It is the responsibility of the corresponding reflective library to implemented such checks.

In Section 3.4.6 we describe a Reflection API for KERIS. The design of this library is based on the architectural design pattern *Context/Component* discussed in Section 4.2.1. It uses meta-module representations and module contexts (i.e. sets of interconnected module instances) for creating, linking, and hot swapping module instances dynamically.

3.3 Applications of Keris

This section presents several examples that show how KERIS can be used to safely implement extensible software components. We focus on well-known design problems and show how some design patterns can be implemented in an extensible way.

3.3.1 Generic Class Families

Family polymorphism. In the first example we use modules as a means to encapsulate sets of related classes. Since modules are extensible, it is possible to create refinements and specializations of such class families. The term *family polymorphism* refers to the ability to statically declare and manage relationships between several classes polymorphically; i.e. in a way that a given set of classes may be known to constitute a family, but where it is not known statically exactly what classes they are [59]. We will proceed by explaining how KERIS supports family polymorphism through extensible modules with class fields.

Generic graph representations. We start with the implementation of a generic module for representing graphs. Listing 3.1 shows a definition of a suitable module interface. This module interface defines class fields for graphs, nodes, and edges together with the corresponding interfaces. Since members of module interfaces are never concrete, we define class fields only by specifying

```
module interface GRAPH {
  class Graph implements IGraph;
  class Node implements INode;
  class Edge implements IEdge;
  interface IGraph {
    IGraph();
    Node[] nodes();
    Node addNode();
  }
  interface INode {
    Edge[] edges();
    Edge connectTo(Node node);
  }
  interface IEdge {
    Node from();
    Node to();
  }
}
```

Listing 3.1: A module interface for graphs.

```
module DIRECTED_GRAPH implements GRAPH {
  class Graph implements IGraph = {
    Node[] nodes = new Node[0];
    Node[] nodes() { return nodes; }
    Node addNode() { ... new Node() ... }
  }
  interface IDNode {
    IDNode();
    boolean reachableFrom(Node node);
  }
  class Node implements INode, IDNode = {
    Edge[] edges = new Edge[0];
    Edge[] edges() { return edges; }
    Edge connectTo(Node node) { ... new Edge(this, node); ... }
    boolean reachableFrom(Node node) { ... }
  }
  interface IDEdge {
    IDEdge(Node from, Node to);
  }
  class Edge implements IEdge, IDEdge = {
    Edge(Node from, Node to) { ... }
    Node from() { return from; }
    Node to() { return to; }
  }
}
```

Listing 3.2: An implementation of module interface Graph.

their implemented interfaces. Such class fields are called *opaque*, because they do not reveal their implementation.

In our implementation, a graph is a data structure that consists of a list of nodes. This data structure is extensible, offering a method for adding new nodes. A node has a number of adjacent edges. It offers a method for adding new edges by connecting the node with another node. Edges are simply objects that have a root and a target node.

A possible implementation of module interface GRAPH is given by module DIRECTED_GRAPH in Listing 3.2. We use anonymous class declarations to specify the concrete implementations for all class fields of module DIRECTED_GRAPH. As Section 3.2.7 pointed out already, the use of anonymous classes is sometimes necessary to give the self reference this the right type. In anonymous classes, this is typed with the corresponding class field type. In Listing 3.2, the implementation of class field Node needs this to be of type Node, otherwise we could not pass it to constructors of Edge.

Note that in DIRECTED_GRAPH, both Node and Edge extend the set of implemented interfaces specified in module interface Graph each with one of the new interfaces IDEdge and IDNode. For both class fields this is essential to enable the

```
module WEIGHTED_GRAPH specializes DIRECTED_GRAPH {
  interface IWNode {
    int shortestPathTo(Node node);
  }
  class Node implements INode, IDNode, IWNode = super {
    int shortestPathTo(Node node) { ... edges[i].weight() ... }
  }
  interface IWEdge {
    void setWeight(int w);
    int weight();
  }
  class Edge implements IEdge, IDEdge, IWEdge = super {
    int weight;
    Edge(Node from, Node to) { super(from, to); }
    void setWeight(int w) { weight = w; }
    int weight() { return weight; }
  }
}
```

Listing 3.3: A specialization of directed graphs.

construction of concrete objects, since module interface GRAPH does not explicitly specify the signature of object constructors for them.

In Listing 3.3 we specialize the DIRECTED_GRAPH module. The specialized version WEIGHTED_GRAPH adds weights to edges. The new implementation of Edge subclasses the previous (anonymous) implementation by referring to this implementation via super. We can now create systems which deal with both weighted graphs and directed graphs. The type system guarantees for such cases that it is not possible to mix elements of the two. The following module definition does not type check because of exactly this reason: In function main we try to link a node of a directed graph with one of a weighted graph.

```
module GRAPHAPP {
  module DIRECTED_GRAPH;
  module WEIGHTED_GRAPH;
  void main(String[] args) {
    DIRECTED_GRAPH.Graph dg = new DIRECTED_GRAPH.Graph();
    WEIGHTED_GRAPH.Graph wg = new WEIGHTED_GRAPH.Graph();
    dg.addNode().connectTo(wg.addNode());  ↯
  }
}
```

Programming languages like gBeta [58] and Scala [151] which support types as object members allow even stronger couplings between members of a class family. Here one could nest the types Edge and Node in class Graph to even disable mixing nodes from different directed graphs.

The next section discusses KERIS' ability to express families of recursively dependent classes in a modular fashion; i.e. in a way that does not require to define all related classes as members of a single enclosing class or module. This example will show that the expressiveness of KERIS goes beyond what is required for family polymorphism. In KERIS, families can be split up modularly into *loosely coupled sub-families* [60] without sacrificing type safety.

3.3.2 Design Patterns as Module Aggregates

Design patterns. In the remainder of this section we use modules to develop generic implementations of design patterns in a modular fashion. Design patterns are general solutions to recurring design problems. They are ways to describe best practices, good designs, and capture experience in a way that it is possible for others to reuse this experience. In the literature [74], design patterns are often presented as descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context.

Subject/Observer pattern. We start with a discussion of the *Subject/Observer* pattern [74], which is also known as the *Publish/Subscribe* pattern. This pattern lets one define a one-to-many dependency between objects so that when one object, called *Subject*, changes its state, all its dependents, called *Observers*, are notified and updated automatically.

Listing 3.4 introduces three modules as the building blocks of this pattern. The observer type is defined in module `OBSERVER` by the class field `Observer`. Module `OBSERVER` has to require the corresponding `SUBJECT` module since the observer type directly refers to the subject. Similarly, module `SUBJECT` requires module `OBSERVER` in its definition of class field `Subject`. Each instance of class field `Subject` contains a list of observers which get notified whenever an event is sent to the subject. We have no concrete implementation for events, so the `EVENT` module is simply described by a module interface.

We can now link the mutually dependent modules together yielding a single module `SUBJECT_OBSERVER` which represents the complete Subject/Observer pattern. In addition to the aggregated modules we also define a function `attach` which attaches an observer object to a subject. The composed module `SUBJECT_OBSERVER` is a natural place for defining functions that belong logically to the whole pattern, and not to a specific participant.

```
module SUBJECT_OBSERVER requires EVENT {
  module SUBJECT;
  module OBSERVER;
  void attach(SUBJECT.Subject s, OBSERVER.Observer o) {
    s.add(o);
  }
}
```

```
module OBSERVER requires SUBJECT, EVENT {
  interface IObserver {
    IObserver();
    void notify(SUBJECT.Subject subj, EVENT.Event evt);
  }
  class Observer implements IObserver = {
    void notify(SUBJECT.Subject subj, EVENT.Event evt) { ... }
  }
}
module interface EVENT {
  class Event;
}
module SUBJECT requires OBSERVER, EVENT {
  interface ISubject {
    ISubject();
    void add(OBSERVER.Observer obs);
    void notify(EVENT.Event evt);
  }
  class Subject implements ISubject = {
    OBSERVER.Observer[] obs;
    void add(OBSERVER.Observer obs) { ... }
    void notify(EVENT.Event evt) {
      for (int i = 0; i < obs.length; i++)
        obs[i].notify(this, evt);
    }
  }
}
```

Listing 3.4: A modular Subject/Observer implementation.

We could create refined versions of that pattern with alternative properties, but here, we are mainly interested in specializing it for a specific application. Following the example in [199], we derive a data structure for modeling a window manager by consistently specializing the mutually referential modules SUBJECT and OBSERVER. We start with the covariant specialization of the SUBJECT module.

```
module MANAGER specializes SUBJECT
               requires WINDOW as OBSERVER, WINEVENT as EVENT {
  interface IManager { ... }
  class Subject implements ISubject, IManager = ...
}
```

Module MANAGER also has to specialize the requirements of the original SUBJECT module with the as construct. This "rewiring" has the effect that all former references to the OBSERVER module now refer to module WINDOW. The same holds for EVENT. Without this specialization we could not link module MANAGER with the cor-

responding module WINDOW since WINDOW is distinct from OBSERVER and therefore cannot play its role in the general pattern.

```
module WINDOW specializes OBSERVER
              requires MANAGER as SUBJECT, WINEVENT as EVENT {
  interface IWindow { ... }
  class Observer implements IObserver, IWindow = ...
}
module WINEVENT specializes EVENT {
  ...
}
```

Finally, we compose the modules to represent the window manager pattern as a specialization of the Subject/Observer pattern. Here we have to specialize the submodules accordingly. We cannot simply override the original SUBJECT and OBSERVER submodules, since our new modules are module specializations that do not subsume their parent modules.

```
module WIN_SYSTEM specializes SUBJECT_OBSERVER
                  requires WINEVENT as EVENT {
  module MANAGER as SUBJECT;
  module WINDOW as OBSERVER;
}
```

In the definition of module WIN_SYSTEM, the type system also enforces the specialization of the required EVENT module. Otherwise, we could evolve SUBJECT_OBSERVER inconsistently with respect to SUBJECT and OBSERVER, whose specializations both require WINEVENT instead of EVENT in module WIN_SYSTEM.

3.3.3 Modular Extensions of Design Patterns

3.3.3.1 Extensible Visitors

Another prominent design pattern is the *Visitor* pattern [74]. The aim of this pattern is to represent operations on elements of a data structure as regular objects. The pattern describes a dispatch mechanism for invoking such operations. It lets you define new operations without changing the classes of the elements on which they operate. On the other hand, this pattern makes it very hard to define new element types. Therefore it is complementary to the *Interpreter* pattern which facilitates the creation of new element classes, but does not allow the addition of new operations. The problem of extending a system simultaneously with new element types and new operations is often called *extensibility problem* or *expression problem*. A detailed discussion of this problem can be found in Section 4.1.1 on page 150.

The implementation of the *Visitor* pattern that is getting derived on the following pages does not have such restrictions. It allows programmers to safely add new element classes and new operations and can therefore be seen as a solution of the extensibility problem.

We start with an abstract description of the pattern's participants in form of module interface VISITORS. This module interface contains a class interface Data for describing general element types. Classes implementing Data have to provide at least an accept function which takes an operation (a visitor object) and applies it to the given object. The second module interface member is class field Visitor which represents a type for operations.

```
module interface VISITORS {
  interface Data {
    void accept(Visitor v);
  }
  abstract class Visitor;
}
```

In contrast to the Subject/Observer implementation from the last section, we cannot factor out the full logic of the Visitor pattern into an abstract implementation and then use specialization to instantiate this abstraction for specific applications. The programmer has to know the full protocol, including the development of concrete visitors and concrete element classes. The purpose of the given infrastructure is to help the programmer setup his concrete visitor framework so that it can simply be reused and extended.

The following listing presents a specialization of the VISITORS interface. It will be used to interface modules that allow to represent and evaluate arithmetic terms consisting of numbers and binary plus operations.

```
module interface EXPRESSIONS specializes VISITORS {
  interface ExprVisitor {
    void caseNum(int x);
    void casePlus(Data left, Data right);
  }
  abstract class Visitor implements ExprVisitor;
  interface EvalVisitor {
    EvalVisitor();
    int result();
  }
  class Eval extends Visitor implements ExprVisitor, EvalVisitor;
}
```

The EXPRESSIONS module interface describes visitors handling the two element types for numbers and plus operations. The module interface does not expose the concrete representation of such data. Therefore it can only be used to process

```
module LANG implements EXPRESSIONS {
  class Num implements Data {
    int x;
    Num(int x) { this.x = x; }
    public void accept(Visitor v) { v.caseNum(x); }
  }
  class Plus implements Data {
    Data left, right;
    Plus(Data l, Data r) { left = l; right = r; }
    public void accept(Visitor v) { v.casePlus(left, right); }
  }
  class Eval extends Visitor implements ExprVisitor, EvalVisitor = {
    int res;
    public int result() { return res; }
    public void caseNum(int x) { res = x; }
    public void casePlus(Data left, Data right) {
      Eval eval = new Eval();
      left.accept(eval); right.accept(this);
      res += eval.result();
    }
  }
}
```

Listing 3.5: A concrete Visitor implementation.

data. We could imagine to define a second view which provides functionality for actually creating data.

In addition to the covariant refinement of the visitor type, module interface EXPRESSIONS also defines class field Eval, a concrete visitor which evaluates terms to integer values. Module LANG which implements module interface EXPRESSIONS is shown in Listing 3.5. The implementation of this module is straightforward; it is guided by the corresponding interface description.

As Listing 3.6 shows, it is quite easy to extend module LANG to incorporate new element types. One simply has to refine class field Visitor in order to add new visitor methods for the new element types. Since all concrete visitors, like Eval, depend on this class field, they have to be updated accordingly. Otherwise the compiler would complain about broken dependencies for all concrete visitor class fields that implement less interfaces than Visitor does.

In Listing 3.6 class field Eval is refined by extending the set of implemented interfaces and by simultaneously extending the anonymous implementation.

In mainstream object-oriented languages, visitor extensions in the presented fashion can only be implemented by circumventing the type system, e.g. by using type casts or reflection. Examples for such extensible visitor design pattern variations are Krishnamurthi, Felleisen and Friedman's *Extensible Visitors* [106], Zenger and Odersky's *Extensible Visitors with Defaults* [214], and Palsberg and

```
module LANG1 refines LANG {
  interface Expr1Visitor extends ExprVisitor {
    void caseMult(Data left, Data right);
  }
  abstract class Visitor implements Expr1Visitor;
  class Mult implements Data {
    Data left, right;
    Mult(Data l, Data r) { left = l; right = r; }
    public void accept(Visitor v) { v.caseMult(left, right); }
  }
  class Eval extends Visitor implements Expr1Visitor,
                                        EvalVisitor = super {
    public void caseMult(Data left, Data right) { ... }
  }
}
```

Listing 3.6: Extension of the Visitor implementation.

Jay's *Generic Visitors* [157]. In C++ it is possible to use the template mechanism to implement extensible visitors without type casts, but here, type checking is deferred until link time. In languages with dependent object types, like e.g. SCALA, it is possible to implement extensible visitors safely, in a fashion similar to the one presented here. But solutions in these languages require more effort and technical knowledge due to more low-level language constructs (which can, on the other side, also be used more flexibly).

3.3.3.2 Extensible Interpreters

The previous section explained how to use the Visitor pattern to separate data from operations in an extensible fashion. We can solve the same problem in a more object-oriented way, using the *Interpreter* design pattern. We will explain how this pattern can be implemented in KERIS such that both data and operations are extensible.

Listing 3.7 defines two modules, `INTERPRETER` and `LANG`. `INTERPRETER` is a small module which encapsulates a supertype `Data` for our various data representations. Its interface lists all the operations which subtypes of `Data` have to support. Module `LANG` defines concrete `Data` variants in form of the class fields `Num` and `Plus`. The following module demonstrates how to deploy both components.

```
module APP {
  module INTERPRETER;
  module LANG;
  void exec(INTERPRETER.Data data) { System.out.print(data.eval()); }
}
```

```
module INTERPRETER {
  interface IData { int eval(); }
  abstract class Data implements IData;
}
module LANG requires INTERPRETER {
  import INTERPRETER.*;
  interface INum {
    INum(int x);
  }
  class Num extends Data implements IData, INum = {
    int x;
    Num(int x) { this.x = x; }
    int eval() { return x; }
  }
  interface IPlus {
    IPlus(Data l, Data r);
  }
  class Plus extends Data implements IData, IPlus = {
    Data left, right;
    Plus(Data l, Data r) { left = l; right = r; }
    int eval() { return left.eval() + right.eval(); }
  }
}
```

Listing 3.7: Extensible interpreter framework.

```
module INTERPRETER1 refines INTERPRETER {
  interface IData1 extends IData { String asString(); }
  abstract class Data implements IData1;
}
module LANG1 refines LANG requires INTERPRETER1 {
  import INTERPRETER1.*;
  class Num extends Data implements IData1, INum = super {
    Num(int x) { super(x); }
    String asString() {
      return String.valueOf(x);
    }
  }
  class Plus extends Data implements IData1, IPlus = super {
    Plus(Data l, Data r) { super(l, r); }
    String asString() {
      return left.asString() + "+" + right.asString();
    }
  }
}
```

Listing 3.8: Language extension in the interpreter framework.

In Listing 3.8 we now extend the base type Data by refining module INTERPRETER. In the new version INTERPRETER1, data variants are supposed to implement an asString method in addition to the previously declared methods. Obviously, the type system of Keris does not allow us to link the old LANG module with this new version of module INTERPRETER, since in this module, subtypes of Data do not implement the new interface IData1. Consequently, we have to derive a new version of module LANG, called LANG1, in which all Data subtypes provide a method asString, and implement interface IData1. We can also develop new variants of class field Data modularly in separate modules, like MYLANG in the following listing.

```
module MYLANG requires INTERPRETER1 {
  import INTERPRETER1.*;
  interface IMult { IMult(Data l, Data r); }
  class Mult extends Data implements IData1, IMult = {
    Data left, right;
    Mult(Data l, Data r) { left = l; right = r; }
    int eval() { return left.eval() * right.eval(); }
    String asString() {
      return left.asString() + "*" + right.asString();
    }
  }
}
```

A system which makes use of all the modules presented so far, can be derived by refining APP and by including the newly developed modules. This example shows that the extensible interpreter pattern supports the notion of *independent extensibility* [194]; i.e. extensions can be developed independently but used jointly.

```
module APP1 refines APP {
  module INTERPRETER1;
  module LANG1;
  module MYLANG;
  void exec(INTERPRETER1.Data data) {
    System.out.print(data.asString() + " = "); super.exec(data);
  }
}
```

The presented technique roughly corresponds to the implementation scheme presented by Findler and Flatt in [64]. The main difference is that Findler and Flatt's approach is untyped and would only work in a typed context, if the type system is structural. A nominal type system requires at least dependent class types to support the presented implementation technique.

3.4 Implementation of Keris

This section discusses the implementation of the KERIS programming language. The KERIS compiler translates KERIS programs first to plain JAVA code, which is then compiled to regular JAVA classfiles. These classfiles are annotated so that the original type information does not get lost in the translation to JAVA. This enables separate compilation. The translation from KERIS to JAVA was designed in a way such that

- reflective operations are not required (apart from type tests and type casts),
- structural or additive changes of a module's source code do not endanger binary compatibility of the generated code,
- generated code runs on every standard JAVA implementation without the need to customize the runtime environment, e.g. by a specialized class loading mechanism, and
- generated code is stand-alone and does not rely on a special runtime library or other middleware.

These issues contribute to a robust and efficient implementation that facilitates the development and the deployment of KERIS programs. Furthermore, the compilation model of JAVA gets preserved allowing existing software development processes and tools for JAVA to be adopted for KERIS.

3.4.1 Basic Modules

Module representation. The KERIS compiler translates every module into a regular JAVA class. Instances of such a class represent module instances. The class constructor takes a parameter representing the *deployment context* in which the module is going to be instantiated. From such a deployment context object, a module instance can get access to all its required module instances.

We discuss the details of the KERIS to JAVA translation using the following example program. We are predominantly interested in the translation of module `M`, which requires module `N` and aggregates an instance of module `O`.

```
module M requires N {
  module O;
  int n = 1;
  int foo() {
    return N.bar() + O::N.bar() + n;
  }
}
```

```
module N {
  int bar() {
    return 0;
  }
}
module O {
  module N;
}
```

```
class M {
  Object $context;                                          1) Enclosing deployment
  M(Object context) { $context = context; }                 context
  M self$0;
  N req$0;                                                  2) Required modules
  O sub$0;                                                  and submodules
  O M$O() { return $configure().sub$0; }
  O $create$O() { return new O($prop); }
  class $Propagator implements $M, $N, $O {                 3) Local deployment
    M $M() { return self$0; }                               context
    N $N() { return req$0;  }
    O $O() { return sub$0;  }
  }
  $Propagator $prop = new $Propagator();
  M $access() {                                             4) Initialization
    if ($prop != null)
      synchronized (this) {
        if ($prop != null) {
          $configure(); $prop = null; $init();
        }
      }
    return this;
  }
  M $configure() {                                          5) Configuration
    if (self$0 == null)
      synchronized (this) {
        if (self$0 == null) {
          self$0 = this;
          req$0 = (($N)$context).$N();
          sub$0 = $create$O();
        }
      }
    return this;
  }
  void $init() { n = 1; }                                   6) Initializer
  int n;                                                    7) Module members
  int foo() {
    return req$0.$access().bar() + sub$0.O$N().$access().bar() + n;
  }
}
interface $M {
  M $M();
}
```

Figure 3.5: Generated Java code for module M.

The JAVA program generated by the KERIS compiler can be found in Figure 3.5. Details like access modifiers are left out for simplicity. Figure 3.5 divides the generated code into seven parts. Part 1 shows the constructor and the instance variable which refers to the deployment context of this module instance. Part 2 declares for every required module and for every submodule a local reference. These local references are used whenever the corresponding module instance is accessed within the module. In addition to these private module reference variables, the KERIS compiler generates two more methods for every submodule: A factory method for instantiating a submodule and a public access method for querying a submodule instance from a client. The access method is used to implement the `::` operator (see method `foo` in Part 7 of Figure 3.5 for an example). Note that access methods cannot simply return the corresponding module instance, because this instance might not have been created yet. This is because references to cooperating module instances are initialized *lazily*. More precisely, constructors of module classes simply store the deployment context in a local variable, they never configure module instances immediately. Therefore, all access methods first have to ensure that submodules are already created before the corresponding submodule reference is returned. This check is performed by calling method `$configure`. Parts 3–6 of the translated module are dealing with module configuration and initialization.

Initialization protocol. KERIS modules are instantiated and linked into an existing module context in several stages:

1. *Instantiation:* For the instantiation of the module, the deployment context object is passed to the module constructor which stores this object in a local variable `$context`. The rest of the newly created module instance remains temporarily uninitialized.
2. *Configuration:* When a submodule gets accessed for the first time, the module will be configured. This means that all references to external modules are resolved and submodules get instantiated.
3. *Initialization:* When a module member, i.e. a function, a variable, or a class constructor, is accessed for the first time, the module will get initialized by executing all its initializers. Before running the module initializers, the system has to make sure that the module is already properly configured.

Module instantiation has to be separated from module configuration, because modules may depend recursively on each other so that at module instantiation time other cooperating modules are not necessarily already instantiated themselves. For a similar reason, it is not a good idea to combine the configuration and initialization of modules. This increases the dependencies between modules artificially, so that the initialization of modules with mutually dependent module initializers can fail, while a strict separation between the two phases would allow the initializers to run conflict free. Here is a program which illustrates this:

```
module A {
  module B;
  Object x = B::C.foo();
  String y = B.z;
  Object bar() { return y; }
}
```

```
module B requires A {
  module C;
  String z = A.x.toString();
}
module C requires B {
  Object foo() { return "keris"; }
}
```

First we assume that module configuration is always immediately preceded by the module initialization. We look at the function call `A.bar()` where `A` is not yet initialized. In this case, the function call will trigger the instantiation of submodule `B` followed by the execution of variable `A.x`'s initializer. This initializer will first configure and initialize submodule `B`; i.e. it will instantiate submodule `C` and execute `B.z`'s initializer `A.x.toString()`. Since `A.x` is not yet initialized and therefore refers to default value `null`, the program will terminate with a `NullPointerException`.

We now look at the way KERIS handles the case. Here, the call `B::C.foo()` does not immediately trigger the initialization of `B`. `B` gets only configured, i.e. submodule `C` is instantiated, and the access to function `foo` induces the initialization of `B`'s submodule `C`. Therefore the initializer of variable `A.x` terminates and returns the string `"keris"`. It is now the access to variable `B.z` in the initializer of variable `A.y` that triggers the initialization of module `B`. But this time, there will be no abnormal program termination caused by a `NullPointerException`.

Note that the instantiation and linking protocol could be split up into even more steps, for instance by initializing variables lazily. As the following example will show, such a refinement of the instantiation protocol could even resolve conflicts that still exist for some KERIS programs.

```
module M {
  module N;
  Object x = N.y;
  int bar() { return x; }
}
```

```
module N requires M {
  Object y = new Object();
  String z = M.x.toString();
}
```

In KERIS, `M.bar()` forces module `M` to be configured and initialized. The initialization process of `M` will evaluate the initializer of `M.x`. This triggers the initialization of module `N`; i.e. first a new `Object` instance gets created and assigned to `N.y`, and afterwards `N.z`'s initializer will be executed. This initializer refers back to variable `x` of module `M`, which is not yet initialized. Consequently, a `NullPointerException` will be thrown.

With a module initialization scheme where variables of modules are initialized lazily, this `NullPointerException` could be avoided. Here, it would be the

body of method `M.bar` that triggers the evaluation of `M.x`'s initializer. This involves the initialization of `N` and subsequently, the evaluation of the value of `N.y`. The method call `M.bar()` would then return the newly created object and terminate without even executing the initializer of variable `N.z` (which caused the `NullPointerException` originally).

Despite the second approach being more robust with respect to recursive dependencies, it did not get adopted for KERIS. KERIS' instantiation and linking protocol was chosen because it was relatively efficient to implement (in contrast to lazy variable initialization) and more transparent and predictable for programmers. Furthermore, it corresponds to the dynamic loading and initialization mechanism of classes in the JAVA virtual machine, which is, in general, well-understood by JAVA programmers.

Configuration. During the module configuration phase, context dependencies get resolved and submodules get instantiated. A module extracts the required module instances from the deployment context which was stored in variable `$context`. The deployment context is basically a mapping from module names to module instances. Every module defines implicitly such a deployment context. This context allows access to the required modules and all the submodules. For every module, the KERIS compiler generates a special class `$Propagator` which implements a module's deployment context in form of an *AbstractFactory* [74]. This class provides for every member module of the deployment context a "factory method" that returns the instance of the corresponding module in the given context (it actually does not create it). In Figure 3.5 we can see that every module instance has exactly one deployment context object stored in variable `$prop`.

From the code of Figure 3.5 we can also see that deployment contexts passed into module instances are always of type `java.lang.Object`. In order to access the factory methods of such an object, we use a trick which circumvents the use of the reflection API. For every module `M` we also generate an interface `$M` which has a single member, the factory method `M $M()`. Deployment contexts which provide an instance for module `M` implement this interface. Therefore, for retrieving an instance of module `M`, we simply have to cast the deployment context object to interface type `$M` and call the appropriate factory method afterwards.[9]

The whole configuration stage is implemented by function `$configure` (see Part 5 of Figure 3.5 on page 99). This function is synchronized so that concurrent programs do not possibly execute the stage twice.[10] `$configure` makes use

[9] In a language with compound types [38, 216] it would be possible to describe the type of the deployment context precisely, as opposed to our approach which always uses the type `java.lang.Object`. For a module which requires other modules M_1, ..., M_n, the context object would be typed with the intersection of the types $\$M_1$, ..., $\$M_n$. Consequently, it would not be necessary to resort to type casts for retrieving the cooperating module instances from the deployment context.

[10] The method is synchronized via the *double-checked locking* technique [178]. For the current JAVA memory model, this technique is known to be unsafe if concurrent programs are compiled

of the technique described in the previous paragraph for resolving context dependencies. Submodules are instantiated simply by calling the appropriate factory methods. These methods pass the local module context stored in variable `$prop` to the various module constructors.

Initialization. A module gets initialized by executing the module initialization blocks and variable initialization expressions defined by the programmer. This phase is implemented by method `$access` (see Part 4 of Figure 3.5 on page 99). It is again synchronized to guarantee that the module initializers are executed exactly once, even in a concurrent system. First, method `$access` configures the module if necessary. Then it runs method `$init` which contains the concrete initialization code written by the programmer. For generating this method, the Keris compiler has to collect all the initialization code scattered throughout a module declaration.

3.4.2 Module Refinements and Specializations

Refinements. The previous section revealed how new modules are translated to Java. We now focus on the extension of existing modules. Again we explain this by looking at a concrete example program in which module `M` of Section 3.4.1 gets refined.

In the following program, module refinement `MR` extends module `M` by covariantly overriding submodule `O`, by adding a new submodule `Q`, and by extending the context dependencies.

```
module MR refines M requires P {
  module OR;
  module Q;
}
```

```
module OR refines O requires Q, P {}
module P {}
module Q {}
```

Since our explanations regarding the translation from Keris to Java did not cover extensibility yet, we will first complete the Java translation of module `M` presented in Figure 3.5 with code that handles extensibility, before turning towards the translation of module refinement `MR`. This missing functionality is shown in Figure 3.6. The code introduces a second `$configure` method which is parameterized with a *configuration context*. Similar to deployment contexts, configuration contexts offer methods for retrieving module instances. This time, simply all required modules and all submodule instances are extracted from the configuration context; i.e. we assume that the caller of the `$configure` method created all submodule instances already externally.

with highly optimizing compilers or code is executed on a shared-memory multi-processor [16]. Our implementation relies on an execution platform that conforms to a proposal [127, 119] that revises Java's memory model so that the double-checked locking idiom provides an efficient solution for implementing lazy initialization.

```
class M {
  ...
  abstract class $Configurator {                    8) Configurator type
    abstract M $M();
    abstract N $N();
    abstract O $O();
  }
  void $configure($Configurator $c) {               9) Configuration by extension
    self$0 = this;
    req$0 = $c.$N();
    sub$0 = $c.$O();
  }
  ...
}
```

Figure 3.6: Functionality missing in Figure 3.5.

The second `$configure` method is called in extensions of `M`. Figure 3.7 shows such a case. Here we can see that even though refinements inherit submodules and requirements, they have an own set of local module references, named `sub$`*n*, `req$`*n*, and `self$0`. During the configuration of such a refined module, these local instances get initialized in method `$configure()` by extracting the required module instances from the deployment context and by calling the factory methods for creating the submodule instances. Afterwards the `$configure($Configurator c)` method of the superclass is called with a configurator object that enables the superclass to initialize its own module references by referring back to the subclass which has already all local module references initialized. So, configuration contexts (`$Configurator` objects) are used to link the various layers (refinements, specializations) of a module while deployment contexts (`$Propagator` objects) are used to link submodules with their host module. Figure 3.8 illustrates the different usage of propagator and configurator objects to link submodules and to configure parent modules. In this diagram, arrows without labels depict data flow.

Specializations. We will now focus on the implementation of module specializations. Here is again an example which extends the code from Section 3.4.1:

```
module MS specializes M requires NS as N, P {
  module OS as O;
  module O;
}
module NS specializes N {}
module OS specializes O requires O {}
```

The example was chosen to exhibit specific difficulties in the translation of module specializations with rewiring instructions. Module `MS` for instance specializes

```
class MR extends M {
  MR(Object $context) { super($context); }                    1) Deployment context
  MR self$0; P req$0; N req$1;
  OR sub$0; Q sub$1;                                          2) Required modules
  O MR$O() { return $configure().sub$0; }                        and submodules
  OR MR$OR() { return $configure().sub$0; }
  Q MR$Q() { return $configure().sub$1; }
  OR $create$OR() { return new OR($prop); }
  Q $create$Q() { return new Q($prop); }
  class $Propagator implements $MR,$P,$N,$OR,$Q {             3) Local deployment
    M $M() { return self$0; }                                    context
    MR $MR() { return self$0; }
    P $P() { return req$0; }
    N $N() { return req$1; }
    O $O() { return sub$0; }
    OR $OR() { return sub$0; }
    Q $Q() { return sub$1; }
  }
  $Propagator $prop = new $Propagator();
  MR $access() { ... }                                                4) Initialization
  MR $configure() {                                           5) Configuration as
    if (self$0 == null) ...                                      new module
      req$0 = (($P)$context).$P(); req$1 = (($N)$context).$N();
      sub$0 = $create$OR(); sub$1 = $create$Q();
      self$0 = this; super.$configure(new $Configurator());
      ...
    return this;
  }
  class $Configurator extends M.$Configurator                 8+9) Configuration
                      implements $MR,$P,$N,$OR,$Q {               by extension
    M $M() { return self$0; }
    MR $MR() { return self$0; }
    N $N() { return req$1; }
    O $O() { return sub$0; }
    P $P() { return req$0; }
    OR $OR() { return sub$0; }
    Q $Q() { return sub$1; }
  }
  void $configure($Configurator $c) {
    req$0 = $c.$P(); req$1 = $c.$N();
    sub$0 = $c.$OR(); sub$1 = $c.$Q();
    self$0 = this; super.$configure(new $Configurator());
  }
  void $init() { super.$init(); }                                     6) Initializer
}
```

Figure 3.7: Generated Java code for module refinement MR.

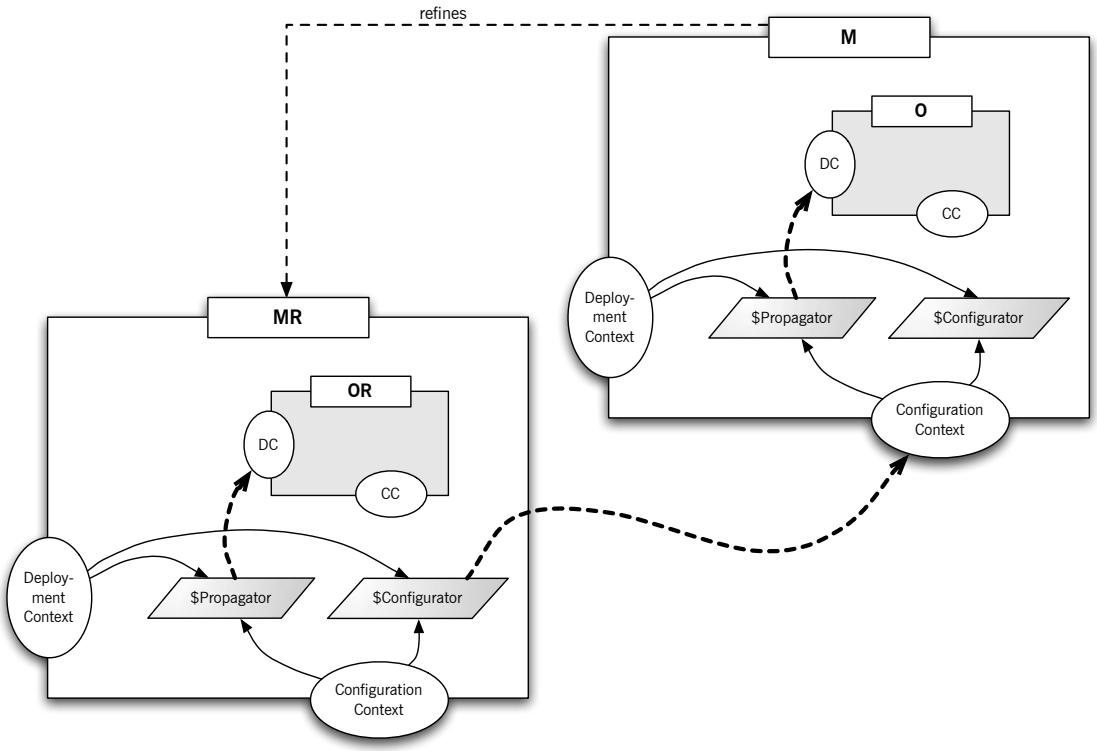


Figure 3.8: Use of propagators and configurators in module refinements and specializations.

the existing submodule O to OS, and at the same time introduces a new submodule O. This new O is distinct from the former O in parent module M which is now implemented by the specialization OS. The definition of OS even requires this new introduction of O in the context of module MS, since it specializes O and simultaneously requires an external O module from the deployment context.

Figure 3.9 shows the JAVA code generated for module MS. It is very similar to the code for module refinement MR. The main difference is that the `$Propagator` and the `$Configurator` class generally do not match anymore for specializations. In refinements on the other hand, there is theoretically no need to distinguish the two. The two arrows in Figure 3.9 indicate where submodules are differently configured than the parent module. While new submodules referring to module O are linked to the newly introduced module O (variable `sub$1`), the variable representing O in the parent module gets linked to the specialized submodule instance OS (variable `sub$0`). Note that the type system ensures that submodules introduced in the parent module of MS that refer to O have to be specialized as well so that they now refer to OS. Otherwise we would have the situation that an old submodule that requires an O instance is actually linked to an instance of module OS, which is not a legal representative of O.

```
class MS extends M {
  MS(Object $context) { super($context); }                    1) Deployment context
  MS self$0; NS req$0; P req$1;
  OS sub$0; O sub$1;                                          2) Required modules
  OS MS$OS() { return $configure().sub$0; }                      and submodules
  O MS$O() { return $configure().sub$1; }
  OS $create$OS() { return new OS($prop); }
  O $create$O() { return new O($prop); }
  class $Propagator implements $MS,$NS,$P,$OS,$O {            3) Local deployment
    MS $MS() { return self$0; }                                  context
    NS $NS() { return req$0; }
    P $P() { return req$1; }
    OS $OS() { return sub$0; }
    O $O() { return sub$1; }                    ⇦
  }
  $Propagator $prop = new $Propagator();
  MS $access() { ... }                                                4) Initialization
  MS $configure() {                                           5) Configuration as
    ...   self$0 = this;                                        new module
          req$0 = (($NS)$encl).$NS();
          req$1 = (($P)$encl).$P();
          sub$0 = $create$OS();
          sub$1 = $create$O();
          super.$configure(new $Configurator());  ...
  }
  class $Configurator extends M.$Configurator                 8+9) Configuration
                      implements $MS,$NS,$P,$OS,$O {             by extension
    MS $MS() { return self$0; }
    M $M() { return self$0; }
    N $N() { return req$0; }
    O $O() { return sub$0; }                    ⇦
    NS $NS() { return req$0; }
    P $P() { return req$1; }
    OS $OS() { return sub$0; }
  }
  void $configure($Configurator $c) {
    self$0 = this;
    req$0 = $c.$NS();
    req$1 = $c.$P();
    sub$0 = $c.$OS();
    sub$1 = $c.$O();
    super.$configure(new $Configurator());
  }
  void $init() { super.$init(); }                                     6) Initializer
}
```

Figure 3.9: Generated Java code for module specialization MS.

Discussion. The implementation scheme presented so far relies on the following principles:

- Classes implement modules, objects represent module instances.
- Modules come to life in three stages: they first are instantiated, then they are configured, and finally they are initialized. *Module configuration* refers to the stage where context dependencies are resolved. *Module initialization* refers to the execution of the user-defined module initializers.
- Module contexts (i.e. sets of module instances) are used to configure modules. Module contexts are implemented with *AbstractFactory*-like classes that provide access methods to all available module instances.
- Module instances configure itself by querying all required module instances from the deployment context (a special module context that contains all modules available in the host module). Every module defines implicitly a deployment context which is implemented by the inner class `$Propagator`.
- Every module layer (the different refinement and specialization deltas) configures itself by querying all required and contained modules from a configuration context (a special module context created by the direct subclass). In the presented implementation scheme, configuration contexts are implemented by the inner class `$Configurator`.
- Module implementations define private variables for referring to required modules and for storing submodule references. Submodule references can be accessed from outside using special accessor methods.

The most characterizing feature of this implementation scheme is probably the extensive use of explicit context objects for linking modules. This approach was chosen for the following reasons:

- It enables modules and/or module layers to configure itself by extracting all context information from a single context object. This is necessary to hide all internal module references.
- By hiding internal module references to external modules, a module does not have to trust the outside world to configure it correctly. A module encapsulates its own trusted logic for linking itself into a given context.
- Since a module encapsulates its own linking procedure, structural changes to a module, like additional requirements or submodules (which might change module internal references), do not necessarily require clients of this module to be recompiled. Therefore, context objects help to decouple modules from their concrete deployment location and promote improved reusability of module binaries by making modules more robust with respect to binary compatibility.

- Furthermore, context objects allow an intuitive implementation of KERIS' module rewiring functionality of module specializations.

Of course, there are many design alternatives. For instance it would be straightforward to avoid the creation of explicit context classes for all modules. One could use a generic representation, e.g. in form of a hash table, to represent module contexts.[11] The advantage of this solution is a reduced number of generated classes. On the other hand, the costs for initializing such hash tables would be linear to the number of contained modules and module versions. Furthermore, this approach gives up any form of type safety. The main drawback of explicit context classes like `$Propagator` and `$Configurator` is the class loading overhead imposed by the current implementation of JAVA. Otherwise, context objects are created in constant time since they do not encapsulate any state like references to the contained module instances. Access to the module instances is gained exclusively via the outer class.

3.4.3 Module Access

Top-level module access. A fully configured module has internal references, represented by the variables sub$*n*, and `req`$*n*, to all directly accessible cooperating modules (i.e. required external modules and submodules). Whenever a member of such a module is accessed, we cannot simply take the module object as the receiver for the member access. Due to lazy module initialization, we first have to make sure that the module is already initialized before the member is actually accessed. In the presented framework, this can be achieved by calling the `$access` method of the corresponding module instance. Here is an example which shows a fragment of a KERIS program and the corresponding JAVA code.

KERIS

```
module M requires N {
  void bar(int y) {
    N.foo(N.x + y);
  }
}
```

JAVA

```
class M {
  N req$0; ...
  void bar(int y) {
    req$0.$access().foo(
      req$0.$access().x + y);
  }
  ...
}
```

As this example suggests, the KERIS compiler prefixes every access to a member of another module with a call to method `$access`. Consequently, every function invocation consists at runtime actually of two virtual method calls. These double

[11] The implementation of the official JAVA compiler JAVAC from Sun Microsystems uses a generic context representation for linking the various compiler components. JAVAC basically deploys a generic variation of the *Context/Component* design pattern presented in Section 4.2.1.

dispatch costs are a price we have to pay for lazy module configuration and initialization. In one of the next paragraphs we present an optimization technique which can help to reduce the number of $access calls significantly and therefore contribute to more compact and efficient code.

Submodule access. Submodules cannot directly be accessed from clients of the host module. The submodule access methods introduced in Section 3.4.1 on page 98 have to be used to retrieve submodule instances from a module. Here is some code which demonstrates this procedure:

KERIS

```
module M requires N {
  void bar(int y) {
    N::O.foo(
      N::O::P.x + y);
  }
}
```

JAVA

```
class M {
  N req$O; ...
  public void bar(int y) {
    req$O.N$O().$access().baz(
     req$O.N$O().O$P().$access().x+y);
  } ...
}
```

It is not necessary to explicitly call the $configure method before applying a submodule access method. The access method itself ensures that the host module is fully configured before access to a submodule instance is granted. Nevertheless, there has to be an application of method $access before a member of a submodule is accessed.

We did not explain yet why names of submodule access methods are composed out of the host module name and the submodule name. It would be simpler to drop the host module name and just use the submodule name alone. Unfortunately, module specializations do not allow such a simple naming scheme. Here, it is possible that a module specialization rewires a submodule and introduces a new instance of the former submodule, similar to the example in 3.4.2. In this case, existing code has to refer to the specialized version. Exactly this is not possible with simple names, because here, the access method of the newly introduced submodule would override the former access method. Our naming scheme does not permit overriding and thus emulates non-virtual method calls that natively do not exist in JAVA.

Optimizations. For some cases, the language allows a more efficient implementation scheme than the general one presented before. For imported submodules, the compiler keeps a direct reference to the imported module in a variable. It is therefore not necessary anymore to follow the full access path by subsequently calling all the submodule access methods just for accessing this module instance. It would also be possible to use this technique for caching other submodule instances that are frequently referred to within a module, but which are not explicitly imported.

The KERIS compiler also optimizes access to synchronized submodules. Such modules are initialized automatically by the initializer of the host module. Here, one can generally drop all calls to `$access`.

As the benchmarks of Section 3.5 and Chapter 4 will show, dropping unnecessary `$access` calls can sometimes speed up the execution of a KERIS programs significantly. The KERIS compiler therefore implements a method local optimization which tries to safely eliminate as many calls to `$access` as possible. This optimization is based on a control-flow analysis that statically determines for every module member access whether the module is already initialized and therefore a call to `$access` can be dropped without risking the module to be uninitialized. Being a conservative optimization that approximates the dynamic behavior statically, it is, of course, not possible to detect all redundant `$access` calls.

Experience with the compilation of the extensible JAVA compiler JACO2 shows that program transformations sometimes help to reduce the number of `$access` calls further. Therefore, the backend of the KERIS compiler implements a few special cases, where programs are transformed into equivalent programs which require less `$access` calls. The following example illustrates the principle.

KERIS

```
module M requires N {
  void foo(int n) {
    while (n > 0) {
      N.bar(N.baz() + n);
      n--;
    }
  }
}
```

JAVA

```
class M {
  N req$0;
  ...
  public void foo(int n) {
    if (n > 0) {
      req$0.$access().bar(
        req$0.baz() + n);
      n--;
      while (n > 0) {
        req$0.bar(req$0.baz() + n);
        n--;
      }
    }
  }
}
```

A naive translation of method `foo` yields a loop in which the second access to module `N` (the invocation `N.baz()`) does not require a call to `$access`. Nevertheless, at runtime, the number of `$access` calls is linear to n. As the right listing above shows, the KERIS compiler performs a *loop hoisting* optimization. It first unrolls the loop a single time and is then able to eliminate both `$access` calls in the actual loop. Therefore, we have in all cases totally not more than a single call to `$access`. The compiler performs such a loop unrolling only, if the code benefits from it; i.e. if at least one `$access` call can be eliminated within the loop.

Section 3.5 presents some micro-benchmarks which explain the benefits of the optimizations outlined in this paragraph. This section also benchmarks the

implementation of the KERIS compiler to investigate the effects of the implementation scheme and the proposed optimizations in a real-world application.

3.4.4 Classes and Types

Every interface contained in a module is translated just like a static nested interface definition in JAVA. Similarly, classes defined in modules are translated like inner classes in JAVA. The only difference is that for supporting type tests and type casts, the KERIS compiler has to add additional methods to interfaces and classes. This will be explained in the next section and is ignored for now.

Class field translation. The translation of class fields is more complicated and explained using the following example.

```
module M requires N {
  interface I {}
  class C extends N::O.D implements I, N.J  = {
    void foo(C c) { N.bar(new C(), c); }
  }
}
module N {
  module O;
  interface J { J(); }
  void bar(O.D d, O.D e) {}
}
module O requires N {
  class Impl implements N.J {}
  class D implements N.J = Impl;
}
```

For the beginning we restrict our attention only to class field `M.C`. This class field extends another class field and implements a new interface `M.I`. The class field is implemented with an anonymous class that inherits from `java.lang.Object`. Here is the relevant code which is generated for the class field by the KERIS compiler:

```
class M {
  static interface I {}
  interface C extends I, N.J {}
  Object new$C() { return new C$Impl(); }
  class C$Impl extends Object implements M.C, O.D {
    void foo(Object c) {
      req$O.$access().bar(M.this.new$C(), c);
    }
  }
  ...
}
```

Every class field `M.C` generally expands to three different kind of definitions:

- An interface `M.C` which represents the type defined by this class field and which extends all the interfaces `M.C` is implementing,
- Constructors `new$C` for creating new class field implementations, and
- An implementation of the class field in form of class `M.C$Impl`. This class implements `M.C` and defines all the concrete class members.

For instantiating a class field `M.C`, the appropriate constructor method `M.new$C` gets called. This method returns an instance of the class field implementation `M.C$Impl`.

Type translation. In the previous program, it is the interface `M.C` which represents the class field with the same name in the original Keris program. The implementation class `M.C$Impl` implements this interface as well as all the interfaces of dependent class fields. Otherwise the interface type `M.C` does not appear in the generated code.

Class field types are translated by erasure; i.e. any occurrence of a class field type is replaced by the type `java.lang.Object`. The disadvantage of this approach is that whenever a member of a class field gets accessed, the object first needs to be cast to the right interface type for gaining access to its member. A probably more serious disadvantage is that this approach restricts overloading and complicates overriding. It is, for instance, not possible anymore to have two overloaded methods where one has a class field parameter type and the parameter type of the other one is simply `java.lang.Object`. In the translation to Java, the method signatures will get identical. Therefore, the type checker of the Keris compiler has to verify that two overloaded method signatures never overlap in the erased counterparts and that overloaded methods do not override methods in the superclass accidentally. Such checks are required by all Java-based programming languages that erase types, e.g. like GJ [32].

Instead of erasing types always to `java.lang.Object`, it would also be possible to choose a representative interface type instead; in the example above this could be interface `M.C` or also `M.I`. This avoids for some cases type casts when accessing object members. Furthermore, the restrictions concerning overloading and overriding are slightly less severe. On the other hand, the overloading restrictions get more complicated and less transparent to users, so that it gets more hard to understand what the compiler is doing. Another technical disadvantage of this approach is that it requires the introduction of bridge methods [32]. Bridge methods introduce additional indirections in method calls artificially and therefore have an effect on runtime efficiency. Furthermore, as the experience with the implementation of Scala and GJ shows, it is quite difficult to generate bridge methods correctly in languages with multiple (interface) inheritance and type refinements.

Class field extension. Whenever a class field is overridden in a module refinement or specialization, a new interface for representing the class field is created. This interface extends the previous one. Furthermore, all constructor methods are overridden with new versions that return instances of a possibly new class field implementation.

3.4.5 Type Tests and Casts

Due to the dependent type system of KERIS, we cannot rely on JAVA's native support for runtime type tests and casts. The KERIS compiler has to explicitly generate methods that enable runtime type tests and casts that conform to the type system of KERIS. The KERIS compiler is doing this in a way that neither requires support from a library nor from a specific runtime system. All data and type checking logic is generated.

Runtime representation of dependent types. We proceed by incrementally showing all the relevant JAVA code, generated for the types defined in module `M` of the previous section. Note that we have to generate supporting code not only for class fields. For soundness reasons, all type definitions, including interface definitions, are subject to the dependent typing rules of KERIS. The code supporting the runtime type representation for interface `M.I` can be found in Listing 3.9.

With KERIS' strictly nominal type system it is possible to represent a type at runtime with a unique object; we call these objects *type representatives*. Since we are only interested in the identity of such type representatives, we simply use instances of class `java.lang.Object`. For interface `M.I`, the type representative is stored in variable `type$I`.

Every class defined in a module provides a `get$Type` method which returns the corresponding type representative. Interfaces have abstract `get$Type` members.

For every type defined in a module we create a set whose elements will be the type representatives of all known subtypes. In the code above, this set is called `subclasses$I`. Furthermore, there is a function `setupType$I` which enters a given type representative into the subclass set of `M.I` and all supertypes of `M.I` (if `M.I` would extend other interfaces). The subclass table of type `M.I` is set up during the configuration of the host module. Subclasses in other modules enter their type representatives at the time they get configured. Therefore, subclass tables are growing by the time with more and more modules getting instantiated and configured. For our case this means that whenever a module is configured which contains a class that implements interface `M.I`, this class' type representative will be entered in set `M.subclasses$I`.

Type tests and casts. With this infrastructure it is quite easy to define methods that perform type tests and casts conforming to the static type system of KERIS.

```
class M {
  static interface I {
    Object get$Type();
  }
  Object type$I = new Object();
  Set subclasses$I = new HashSet();
  boolean instanceof$I(Object $c) {
    return $c instanceof I &&
           $configure().subclasses$I.contains(((I)$c).get$Type());
  }
  Object cast$I(Object $c) {
    if ($c == null || instanceof$I($c))
      return $c;
    else
      throw new ClassCastException();
  }
  void setupType$I(Object $c) {
    $configure().subclasses$I.add($c);
  }
  M $configure() {
    ... self$0 = this;
        req$0 = (($N)$context).$N();
        setupType$I(type$I);
        setupType$C(type$C); ...
  }
  ...
}
```

Listing 3.9: Runtime type support for class fields.

Method `M.instanceof$I` implements a type test for `M.I`. It first deploys JAVA's native `instanceof` operation, then it retrieves the type representative and checks if it is contained in the subclass set. If this is not the case, the type test fails. Type casts handle `null` references specially, otherwise they make use of the type test method to find out, if a cast is safe. For safe casts they simply return the object, otherwise a `ClassCastException` is thrown. So type cast methods like `M.cast$I` simply check if a cast is safe, they actually do not perform a real type cast themselves. Here is a translation scheme for converting source level type casts and type tests to JAVA:

KERIS	JAVA
`... x instanceof M::N.C ...`	`... sub$0.M$N().instanceof$C(x) ...`
`... (M::N.C)x ...`	`... sub$0.M$N().cast$C(x) ...`

The compilation scheme of the runtime typing infrastructure for class fields is identical. Here is the relevant code generated for class field `M.C`:

```
class M {
  Object type$C = new Object();
  Set subclasses$C = new HashSet();
  boolean instanceof$C(Object $c) {
    return $c instanceof C &&
           $configure().subclasses$C.contains(((C)$c).get$Type());
  }
  Object cast$C(Object $c) {
    if ($c == null || instanceof$C($c)) return $c;
    else throw new java.lang.ClassCastException();
  }
  void setupType$C(Object $c) {
    $configure().subclasses$C.add($c);
    this.setupType$I($c);
    req$O.setupType$J($c);
    req$O.N$O().setupType$D($c);
  }
  class C$Impl extends Object implements M.C, O.D {
    Object get$Type() { return type$C; }
    ...
  }
  ...
}
```

The biggest difference can be found in method `M.setupType$C`. Since `M.C` is a subtype of `M.I`, `N.J`, and `N::O.D`, this method has to enter the type representative of `M.C` into the subclass sets of these three types.

One might wonder why the implementation of KERIS relies on subclass sets instead of superclass sets. Superclass sets are, for instance, used by some implementations of the JAVA virtual machine. Their advantage is that the sets are known in advance and are not changing during runtime. In KERIS though, superclass sets would complicate code generation significantly. For an implementation based on superclass sets, it would not be possible anymore to encapsulate all dynamic type test logic in a single method. The logic would have to be partially inlined whenever a type cast or type test is required.

Imagine compiling an expression of the form `obj instanceof M.C`. In a solution based on superclass sets, `obj` would provide a method `isInstanceOf` which takes a type representative and returns true, if this type representative is in the corresponding superclass set of `obj`. Since `obj` could refer to `null` or to an object of a plain JAVA class (which does not have the `isInstanceOf` method) it would be necessary to handle such special cases manually before it is actually possible to call the `isInstanceOf` method. Due to these complications, subclass sets were considered to allow for the more elegant solution.

Exception handling. Dynamic type tests that reflect the static type system are also required for doing exception handling properly. Here, the mechanism provided by the JAVA virtual machine cannot be used directly; it can only be used as

an approximation. In the current implementation of Keris, exception handling does not use the type tests generated by the Keris compiler. Thus, exception handling is currently statically unsafe.

Objective. The aim of the presented compilation scheme for dynamic type tests and casts is to implement type tests and casts as efficient as possible. Since a type test basically consists of a native `instanceof` call followed by a lookup in a hash set, our approach allows, in the average, type tests in constant time.

Another concern in the design of the compilation scheme was extensibility. Since class fields can be overridden such that they implement more interfaces, it is a requirement that subtype tests can be accommodated non-invasively and modularly. In the presented framework, this can be achieved by overriding the `setupType$X` methods which basically implement the subtype relationships of an interface, class, or class field.

Restrictions. Unfortunately, array types cannot be handled properly by the presented compilation scheme. Here, the compiler does not have the possibility to attach additional runtime type information to the corresponding array objects. The only possibility to store runtime type information for arrays is to maintain a global *weak hash table* which maps all created arrays to their proper element type. While this approach would allow to implement reliable type casts and type tests for arrays in a straightforward way, it's usage to implement safe array store operations seems to be unrealistic.[12] Here, every single store operation would have to be preceeded by a slow, table-based type test. Support for handling runtime types of arrays is currently not implemented in the Keris compiler.

3.4.6 Reflection

The implementation of Keris provides functionality for inspecting the configuration of modules at runtime and for re-linking module instances dynamically. These reflective features are available to the programmer in form of libraries. The compiler does not generate supporting code for a specific library, allowing to write several reflection APIs based on different module interconnection principles. We first present the compiler support and then briefly discuss the implementation of a reflection library for Keris.

3.4.6.1 Compiler Support

We use the following module declaration to illustrate what functionality is generated by the Keris compiler to support reflection. The code generated for this module can be found in Listing 3.10.

[12] Since Java supports covariant array subtyping, which is known to be unsound, it is necessary to check dynamically that the objects that are stored in an array are compatible to this array.

```
class M {
  static final String[] $requires = {"N"};
  N req$0;
  synchronized HashMap $requires() {
    HashMap $c = new HashMap();
    $c.put("N", req$0);
    return $c;
  }
  synchronized void update$requirements(HashMap $c) {
    req$0 = (N)$c.get("N");
  }
  static final String[] $contains = {"O", "P"};
  O sub$0;
  P sub$1;
  synchronized HashMap $contains() {
    HashMap $c = new HashMap();
    $c.put("O", sub$0);
    $c.put("P", sub$1);
    return $c;
  }
  synchronized void update$submodules(HashMap $c) {
    sub$0 = (O)$c.get("O");
    sub$1 = (P)$c.get("P");
  }
  Object asObject() { return this; }
  ...
}
```

Listing 3.10: Compiler support for reflection.

```
module M requires N {
  module O;
  module P;
}
```

For supporting the analysis of the static structure of modules, the compiler generates two static class fields `$requires` and `$contains` for every module. These fields contain the names of all required modules and all submodules.

There are virtual methods of the same name that can be used to retrieve the dynamic configuration of a given module instance. Such configurations are simply hash tables that map module names to module instances. To allow dynamic modifications of module configurations, the compiler generates two methods `$update$requirements` and `$update$submodules`. The first updates all local references to external modules, the latter updates references to submodules.

In the reflective system, module instances are treated like regular objects. Since KERIS does not allow programmers to pass module instances around as regular object values, every module provides a function `asObject` which returns the

```
public final class Module {
  // Is this a module interface?
  boolean isInterface();
  // Is this a module refinement?
  boolean isRefinement();
  // Is this a module specialization?
  boolean isSpecialization();
  // Returns the fully qualified module name.
  String getName();
  // Returns the modifier set.
  int getModifiers();
  // Returns the parent module.
  Module getParent();
  // Returns an array of all parent modules.
  Module[] getPreviousVersions();
  // Returns all directly implemented interfaces.
  Module[] getInterfaces();
  // Returns the package in which this module is defined.
  Package getPackage();
  // Returns all required modules.
  Module[] getRequired();
  // Returns all submodules.
  Module[] getSubmodules();
  // Checks if the given module is required by this module.
  boolean isRequired(Module mod);
  // Checks if the given module is a submodule of this module.
  boolean isSubmodule(Module mod);
  // Is this module a refinement of the given module?
  boolean refinementOf(Module that);
  // Is this module a specialization of the given module?
  boolean specializationOf(Module that);
  // Does this module "overlap" with the given module?
  boolean differentFrom(Module that);
  // Return the corresponding java.lang.Class object.
  Class asClass();
}
```

Listing 3.11: Class `keris.reflect.Module` (Part 1: Reflective module representation).

module instance as an object with erased type. With the help of a reflection API it is then possible to introspect the module instance or to change its internal state or its configuration.

3.4.6.2 Library Support

We now present a reflection API that is based on the notion of module contexts. The library uses functionality of the standard JAVA reflection library to access the generated methods illustrated in Listing 3.10.

The library basically consists of two classes: the class `Module` and the class `Context`. Both are written in plain JAVA so that the library can also be used to integrate KERIS components into JAVA programs.

```
public final class Module {
  // Is this a module instance?
  static boolean isImpl(Object impl);
  // Configures the module instance by resolving context dependencies and by
  // creating submodule instances.
  static void configure(Object impl);
  // Initializes the given module instance by executing its module initializers.
  static void init(Object impl);
  // Invokes the given module instances main method with the given arguments.
  static void invoke(Object impl, String[] args);
  // Returns the configuration of a given module instance as a module context.
  static Context getContext(Object impl);
  // Reconfigures a given module implementation with a given module context.
  static void setContext(Object impl, Context context);
  // Propagate the context of a module to all submodules.
  static void propagate(Object impl);
  // Reconfigures the given module instance and propagates the new setting to
  // all submodules. This method can be used to hot swap submodules.
  static void propagateContext(Object impl, Context context);
  // Returns a module representation for a module of the given name.
  static Module forName(String moduleName);
  // Returns a module representation for a module specified by a Class instance.
  static Module forClass(Class clazz);
  // Returns a module representation corresponding to the given module instance.
  static Module forImpl(Object o);
}
```

Listing 3.12: Class `keris.reflect.Module` (Part2: Creating module representations and handling module instances).

Module class. Instances of class `keris.reflect.Module` represent modules and module interfaces in a running Keris application similar to instances of class `java.lang.Class` which represent regular JAVA classes.

Like `java.lang.Class`, class `Module` has no public constructors. Unlike the instances of class `java.lang.Class`, `Module` instances do not get automatically created by the KERIS runtime system. KERIS strongly separates the reflective model from the runtime system, so that it is possible to use several reflective models simultaneously, side by side. Meta-module representations in form of `Module` instances get created explicitly, with the help of three static methods: `forName`, `forClass`, and `forImpl`. See Listing 3.12 for details.

There are several methods for retrieving information about the static structure of a module. In particular, it is possible to relate different modules; e.g. there are methods that check if a module is a specialization or a refinement of another module, or if a module is required or aggregated by another module. An overview over the functionality is presented in Figure 3.11.

In addition, functionality for querying and modifying dynamic aspects of concrete module instances is available (see Listing 3.12). The library enforces the execution of the various module initialization stages (configuration, initializa-

tion). It allows programmers to retrieve the current runtime configuration of an already initialized module instance. The runtime configuration of a module instance is expressed in terms of a `Context` object which maps `Module` objects to concrete module instances. Method `getContext` can be used to retrieve the module context implicitly defined by a Keris module instance. Such implicitly defined contexts contain all the required module instances and all the submodule instances. After retrieving such a context, it can be modified, e.g. by replacing an existing module instance with another compatible one, and it can be written back to the module instance with the help of method `setContext`. A call to method `propagate` will then re-link the whole module instance including all its submodules, so that all module instances now consistently refer to module instances of the new context.

Hot swapping. The procedure described in the previous paragraph can be used to *hot swap*, i.e. dynamically replace, module instances in Keris. One has to keep in mind though that hot swapping modules in the described fashion only guarantees the structural integrity of the system: If hot swapping succeeds, the system is still properly linked (all context dependencies are resolved), and all modules referring to the old module instance now refer to the new instance. If, at the time of the hot swap, functions of the old module are being executed, this execution of old code will be continued even after the hot swap was finished. Such old modules refer to other old modules so that most inconsistencies are avoided which could arise if new and old code would be mixed arbitrarily.[13] Also existing objects created by old modules still refer to the "old world" for consistency reasons. When old objects are processed by new functions, such functions may fail with an exception, if the old object does not support a newly introduced interface. Therefore, new versions of modules can only be hot swapped safely with old versions if interfaces of class fields are kept unchanged, or if all new functions handle old objects specially. Therefore, the presented mechanism only works properly if hot swapping is anticipated in the development of new versions. Like Keris, most platforms with explicit support for hot swapping, e.g. the Erlang runtime environment [14], require the anticipation of hot swaps and a manual synchronization of the actual hot swap action.

The use of module contexts for linking allows one to hot swap several modules at the same time. This feature is mainly required when recursively dependent modules have to be upgraded. Such an upgrade can only be effected consistently if several modules are swapped simultaneously.

[13] The programmer has to synchronize a hot swap action manually, otherwise he risks that the system is only partially swapped, allowing any mixing of old and new code. Old and new code can also be mixed if recursively dependent modules are not hot swapped simultaneously. It this the responsibility of the reflective library to rule out this case.

```
public final class Context {
  // Creates a new empty context.
  Context();
  // Returns an array of all modules provided by this context.
  Module[] modules();
  // For a given module, return its implementation in this context.
  Object implementationOf(Module that);
  // Check if the given module is implemented in this context.
  boolean isImplemented(Module that);
  // Is the given module compatible to the context so that any possible
  // reference to this module is unambiguous?
  boolean compatibleTo(Module that);
  // Check if the given array of modules is compatible to the context so that
  // any possible reference to any of these modules is unambiguous.
  boolean compatibleTo(Module[] thatmod);
  // Check if the given context is compatible to this context so that any
  // possible reference to a members of the given context is unambiguous
  // in this context.
  boolean compatibleTo(Context that);
  // Adds an implementation for a given module to this context.
  void include(Module mod, Object impl);
  // Adds all module implementations of a given context to this context.
  void include(Context that);
  // Links a new module instance for a given module into this context.
  Object link(Module mod);
  // Links a set of new module instances (that may recursively depend on
  // each other) into this context.
  Object[] link(Module[] mods);
  // Updates an outdated version of a given module with an instance of the
  // new version.
  Object update(Module mod);
  // Updates the outdated versions of the given modules with instances of
  // the new module versions.
  Object[] update(Module[] mods);
}
```

Listing 3.13: Class `keris.reflect.Context`.

Context class. Instances of the class `Context` represent linking contexts. Linking contexts are sets of interconnected module instances. A `Context` object maps `Module` objects to corresponding module instances.

It is possible to instantiate empty contexts and link in modules dynamically, or one can use method `getContext` of class `Module` to retrieve the module context implicitly defined by a KERIS module instance.

Once a context object is created, it is possible to manipulate this context. The library guarantees that this happens safely, allowing only well-formed context objects to be created. A context object is well-formed, if all dependencies of the context members are consistently resolved. If, for instance, a new module in-

stance is to be added to a context using method `link`, then it needs to be checked if the context can satisfy all the requirements of the new module instance. Otherwise, method `link` will fail. Since recursively dependent modules have to be included simultaneously, there is a second version of method `link` which adds several new module instances at the same time.

Unlike method `link`, method `include` does not create a new module instance. It rather takes an existing module instance, and integrates it into the context. This method can yield contexts that are linked in ways that are not captured by the static type system of KERIS. Therefore, this method has to be used with caution. Finally there is a method `update` which makes it possible for programmers to replace an existing module instance with a compatible version dynamically.

Besides the functionality for modifying context objects by introducing new or updating old module instances, class `Context` also offers numerous functions for relating two context objects. A summary of this functionality is given in Listing 3.13.

3.4.7 Module Execution

Executable modules. Modules without context dependencies are executable if they define a main method with the signature `void main(String[] args)`. To turn such modules into executable JAVA classes, we have to generate a static main method which instantiates the module and calls the corresponding main method of the newly created module instance. The following code illustrates this.

KERIS

```
module M {
  void main(String[] args) {
    System.out.println("Hi!");
  }
}
```

JAVA

```
class M {
  M(Object context) { ... }
  ...
  void $main(String[] args) {
    System.out.println("Hi!");
  }
  static void main(String[] args) {
    new M(null).$access().$main(args);
  }
}
```

The problem with this solution is that we get a name clash between the generated static main method and the hand-written non-static main method. The KERIS compiler resolves this clash by renaming the hand-written method to `$main`. In fact, it consistently renames every function with name `main` to `$main` if it is defined directly inside of a module or a module interface. This renaming does not cause legacy problems, because modules and module interfaces do not exist in JAVA; it is therefore not possible to introduce any incompatibilities.

Monitoring module initialization. Modules are instantiated and initialized lazily. As explained in Section 3.2.3, this can cause problems with recursively dependent modules. Our KERIS implementation allows one to monitor the module initialization process by emitting log messages on demand. When the JAVA virtual machine is invoked with the option `-Dkeris.verbose`, the runtime system will generate a log which contains all necessary information for debugging possible initialization conflicts. We omit a detailed explanation of how the code gets instrumented to produce such log messages. To illustrate this facility, we just show a small fragment of the module initialization log obtained from a run of the KERIS compiler (which itself is written in KERIS). From the log it can be clearly seen when a module gets created, configured, and initialized.

```
> java -Dkeris.verbose org.zenger.keco.MAIN
# create MAIN
# configure MAIN
#   create MAIN.K_OPTIONS
#   create MAIN.K_COMPILER
...
# initialize MAIN.K_COMPILER
# configure MAIN.COMPILER.REPORT
# initialize MAIN.COMPILER.REPORT
# configure MAIN.COMPILER.PREDEF
# initialize MAIN.COMPILER.PREDEF
#   configure MAIN.K_COMPILER.K_DEFS
#   initialize MAIN.K_COMPILER.K_DEFS
#     configure MAIN.K_COMPILER.K_CLASSREADER
#       create MAIN.K_COMPILER.K_CLASSREADER.K_ATTRIBREADER
#       create MAIN.K_COMPILER.CLASSREADER.CLASSIN
#       create MAIN.K_COMPILER.CLASSREADER.POOL
#     initialize MAIN.K_COMPILER.K_CLASSREADER
#       configure MAIN.K_COMPILER.K_CLASSREADER.K_ATTRIBREADER
#       initialize MAIN.K_COMPILER.K_CLASSREADER.K_ATTRIBREADER
#       configure MAIN.K_COMPILER.CLASSREADER.CLASSIN
#       initialize MAIN.K_COMPILER.CLASSREADER.CLASSIN
#       configure MAIN.K_COMPILER.CLASSREADER.POOL
#       initialize MAIN.K_COMPILER.CLASSREADER.POOL
#   configure MAIN.COMPILER.NAMES
#   initialize MAIN.COMPILER.NAMES
#     configure MAIN.COMPILER.CONVERSIONS
#     initialize MAIN.COMPILER.CONVERSIONS
#   configure MAIN.K_COMPILER.K_TYPES
#   initialize MAIN.K_COMPILER.K_TYPES
#   configure MAIN.COMPILER.SCOPES
#   initialize MAIN.COMPILER.SCOPES
#   configure MAIN.COMPILER.MANGLER
#   initialize MAIN.COMPILER.MANGLER
#   configure MAIN.COMPILER.CONSTS
#   initialize MAIN.COMPILER.CONSTS
#   configure MAIN.K_COMPILER.K_AST
#   initialize MAIN.K_COMPILER.K_AST
```

```
# configure MAIN.COMPILER.OPERATORS
# initialize MAIN.COMPILER.OPERATORS
# configure MAIN.COMPILER.UNITS
# initialize MAIN.COMPILER.UNITS
```

3.4.8 KeCo: The Keris Compiler

We implemented a compiler prototype for Keris which compiles Keris programs following the presented implementation scheme. The compiler reads Keris source code and produces standard Java classfiles for classes as well as modules. Since Keris is designed to be a conservative extension of Java which fully interoperates with regular Java classes, the Keris compiler can also be used as a drop-in replacement for *javac*.

The Keris compiler is written in Keris itself. For bootstrapping the system we first developed a compiler for a subset of Keris which extends our extensible Java compiler JaCo [214, 215]. JaCo itself is designed to support unanticipated extensions without the need for source code modifications. It is written in a slightly extended Java dialect which supports extensible algebraic datatypes [214]. The type checker of this initial Keris compiler was incomplete, and the code generator created only unoptimized code. Nevertheless this temporary compiler was robust enough to be used in the re-implementation of the compiler in the programming language Keris itself. The current version of the compiler, called KeCo, implements the full language, compiles itself, and generates optimized code. An overview over the design and implementation of KeCo is presented in Section 4.3.

3.5 Benchmarks

Extensibility does not come "for free." It often pays off already at the *design stage* to invest in considerations about later changes (*design for change*) even though at this time, it is probably completely unclear what concrete changes that might be. KERIS helps at the *implementation level* to build systems that can easily be extended later, especially in cases where the future changes cannot be anticipated. But the use of KERIS also imposes costs both in terms of code size and runtime efficiency mostly because of additional indirections that are being introduced by the KERIS compiler for extensibility purposes.

This section will neither analyze the costs for designing nor writing extensible software. It will focus on the runtime costs imposed by the current implementation of the KERIS compiler which deploys the compilation scheme presented in the previous section. We proceed by first analyzing some micro benchmarks which are supposed to reveal the runtime overhead associated with specific language constructs, like method calls, class instantiations, type tests, and type casts. Then we will briefly discuss the performance of a real-world program which was first written in a slightly extended JAVA dialect and then ported to KERIS.

3.5.1 Micro Benchmarks

Framework. All the micro benchmarks are based on the framework shown in Figure 3.10. The framework consists of a single main function which executes a specific instruction n times. Since we are not interested in the overhead of the framework itself as well as the overhead imposed by the JAVA runtime environment, we measured the runtime of the framework without placing a statement in the body of the loop. This overhead is presented in the table of Figure 3.10 for $n = 1000000$ and $n = 2000000$. In all the micro benchmarks of this section we subtracted this overhead from the overall time so that the numbers only reflect the costs of the statement in the body of the loop.

All benchmarks were made on an *Apple PowerBook G4 (400 MHz, 512MB RAM)* running Apple's *Java 2 Runtime Environment 1.4.1* on *MacOS X 10.2.6*. We typically executed the benchmark programs six times and took only the run which exhibited the best performance. Since the behavior of JAVA's just-in-time

```
module Overhead {
  void main(String[] args) {
    int i = 0;
    while (i++ < n);
  }
}
```

n	JIT	VM
1000000	114	1752
2000000	213	3515

Figure 3.10: Overhead of the benchmark framework for the just-in-time compiler (JIT) and the bytecode interpreter (VM) (measured in ms).

compiler (JIT) is often quite different from the runtime behavior of the bytecode interpreter (VM), we benchmarked both systems independently.

Function calls. The code for this benchmark is shown together with the benchmark results in Figure 3.11. The main function calls a function foo of a submodule A one million times. With all optimizations switched off, the KERIS compiler generates code that will call the module initialization method $access of submodule A every time foo is invoked. In the optimized case, the $access method will only be called once, during the execution of the first loop iteration.

The runtime numbers given in Figure 3.11 were obtained by timing several runs of the program and by subtracting the overhead displayed in Figure 3.10, yielding the pure function dispatch costs for one million invocations of function foo. The numbers show that the calls to $access slow-down the actual function dispatch time by a factor of more than 4 in the JIT case, and 2 in the VM case. From the code optimization perspective, this is equivalent to a speedup of the optimized program of 77% (JIT), respectively 54% (VM) relative to the unoptimized program.

Of course, these numbers refer to the pure dispatch costs and will never show up in such an extreme manner for real programs where function dispatch costs constitute only a relatively small fraction of the overall runtime.

Class field instantiations. We now target instantiations of classes and class fields. The corresponding specialization of the framework is shown in Figure 3.12. It creates one million instances of class field C defined in submodule A. This program was again timed for the optimizing and the non-optimizing compiler. Furthermore, a small variation of the program was created in which instead of class field C an empty top-level JAVA class is instantiated. Also for this benchmark, the $access method for submodule A will be called in every iteration of the loop if all optimizations are switched off. The benchmark results in Figure 3.12 show that due to the higher runtime costs for instantiations, the rel-

```
module Bench1 {
  module A;
  void main(String[] args) {
    int i = 0;
    while (i++ < 10000000)
      A.foo();
  }
}
module A {
  void foo() {}
}
```

	JIT	VM
Unoptimized	172	7625
Optimized	39	3513
Gain	133	4112
	≅ 77%	≅ 54%

Figure 3.11: Function call costs in the optimized and unoptimized case (in ms).

```
module Bench2 {
  module A;
  void main(String[] args) {
    int i = 0;
    while (i++ < 10000000)
      new A.C();
  }
}
module A {
  interface I { I(); }
  class C implements I = {}
}
```

	JIT	VM
Unoptimized	1388	16910
Optimized	1259	12698
Class	1001	7357
Overhead		
Unoptimized	387	9553
	≅ 28%	≅ 56%
Optimized	258	5341
	≅ 20%	≅ 42%

Figure 3.12: Costs of class and class field instantiations (in ms).

ative difference between the runtime of the optimized and the unoptimized case is significantly smaller than in the previous benchmark.

But here we are more interested in the overhead of class field instantiations with respect to plain class instantiations. In addition to possible $access calls we also have to consider that class fields are created indirectly through factory methods. In the JIT case, the instantiation of class fields is affected with an overhead of 20% compared to the instantiation time of plain classes, if the code is generated by the optimizing compiler. In the VM case it is 42%. For the compiler which leaves in all $access calls, this percentage is even higher: 28% for the JIT, and 56% for the VM. Again, these numbers refer to the plain instantiation time which makes up only a very small fraction of every JAVA or KERIS program.

Method calls. Dispatching method invocations on class field objects is more expensive than dispatching methods of regular classes for two reasons:

1. Class field types get erased in the translation to JAVA requiring that every method call is preceded by a type cast to an appropriate interface type.
2. Especially for the JIT, dispatching methods of interfaces can be slower than dispatching methods of classes.

To analyze the different method dispatch times, we created several variations of the program shown in Figure 3.13. The first version corresponds to the program printed in Figure 3.13, the second version was obtained by eliminating the generated type cast by hand, in the third version the dispatch on a class field method got replaced by a dispatch on a class method, and the last version finally invokes a method through an interface type. In the remainder of this paragraph we restrict our attention to the JIT numbers.

From the runtime numbers of the first two versions we can see that the type cast introduced by the KERIS compiler accounts for almost 60% of the dispatch time of the unoptimized program. Unfortunately, the bytecode verifier of the

```
module Bench3a {
  module A;
  void main(String[] args) {
    A.C c = new A.C();
    int i = 0;
    while (i++ < 20000000)
      c.foo();
  }
}
module A {
  interface I {
    I(); void foo();
  }
  class C implements I = {
    public void foo() {}
  }
}
```

	JIT	VM
Unoptimized	918	7503
Optimized	392	6030
Class	16	5630
Interface	378	6010
Overhead (unopt.)		
~ Class	902	1873
	≅ 98%	≅ 25%
~ Interface	540	1493
	≅ 59%	≅ 20%

Figure 3.13: Dispatch costs for methods of classes, interfaces, and class fields (in ms).

Java Virtual Machine allows the elimination only for some cases (like our benchmark). In general, such type casts cannot be eliminated. Thus, the numbers of the optimized program are fictitious and will be disregarded from now on.

The dispatch on class methods must obviously be highly optimized by the JIT. With a runtime of 16ms for two million calls, it is even quite likely that the JIT detected the subsequent calls to a single empty method and removed the whole loop. It is surprising to see that method invocations through interface types obviously do not benefit from a similar optimization — they are almost 24 times more expensive. Invocations through class field types are 57 times more expensive than through class types, and 2.4 times slower than through interface types. In addition to these numbers, Figure 3.13 also shows the overhead of method dispatches via class field types in relation to dispatches via class or interface types.

To assess the performance of method invocations in (more realistic) situations where the JIT cannot predict the receiver, we conducted a second, more complicated experiment. The code for this experiment is presented in Figure 3.14 together with the corresponding benchmark results. To avoid trivial optimizations of the JIT, we create an array which refers to 64 objects instantiated from four different classes. In every iteration of the `while` loop a new receiver is chosen from this array. Additionally, our framework ensures that subsequent receivers are instantiated from different classes.

Like before, we created variants of the benchmark program to allow a comparison of the method dispatch times for regular class and interface types with the corresponding performance of class fields. Again, the numbers listed in the statistics of Figure 3.14 only refer to the total runtime of the two million method calls in the `while` loop and do not take the overhead created by the other statements into account.

```
module Bench3b {
  module A;
  void main(String[] args) {
    A.C[] cs = new A.C[64];
    for (int j = 0; j < 64; j+=4) {
      cs[j]   = new A.C1();
      cs[j+1] = new A.C2();
      cs[j+2] = new A.C3();
      cs[j+3] = new A.C4();
    }
    int i = 0;
    while (i++ < 20000000) {
      A.C c = cs[i & 0x3f];
      cs[(i+64) & 0x3f] = c;
      c.foo();
    }
  }
}
module A {
  interface I { I(); void foo(); }
  abstract class C implements I;
  class C1 extends C implements I=
  { public void foo() {}}
  class C2 extends C implements I=...
  class C3 extends C implements I=...
  class C4 extends C implements I=...
}
```

	JIT	VM
Unoptimized	1872	7752
Optimized	1403	6199
Class	889	5739
Interface	1367	6155
Overhead (unopt.)		
~ Class	983	2013
	≅ 53%	≅ 26%
~ Interface	505	1597
	≅ 27%	≅ 21%

Figure 3.14: More complicated experiment to measure method dispatch costs (in ms).

As the benchmark results of this experiment show, the runtime behavior of our more complicated test case is identical to the original test when executed on the JVM. The numbers for the execution by the JIT are though quite different, in particular for the dispatch on class types. According to this experiment, method dispatch on interfaces is slower by a factor of 1.5 compared to classes. Method dispatch on class fields is slower by a factor of 2.1 compared to classes, and 1.37 compared to interfaces. Thus, in this experiment, the additional type cast which is required for dispatching on class field types accounts for only 27% of the overall dispatch time of the unoptimized program compared to the 60% in the first experiment.

We assume that the actual method dispatch slow-down in real-world programs is better characterized by the second experiment. Unfortunately, every language that compiles types by erasure on the Java Virtual Machine is affected with this slow-down. It is not possible to carry out any significant optimizations. Since in typical object-oriented programs, method calls happen extraordinary frequently, additional dispatch costs can indeed have an influence on the overall performance of a real-world program.

```
module Bench4 {
  module A;
  Object o = new A.C();
  void main(String[] args) {
    boolean c; int i = 0;
    while (i++ < 10000000)
      c = o instanceof A.C;
  }
}
module A {
  interface I { I(); }
  class C implements I = {}
}
```

	JIT	VM
Mod. member	5189	58347
Class	136	1864
Interface	216	1866
Overhead		
~ Class	5053	56483
	≅ 97%	≅ 97%
~ Interface	4973	56481
	≅ 96%	≅ 97%

Figure 3.15: Costs for type tests against JAVA classes, JAVA interfaces, and class abstractions nested in KERIS modules (in ms).

Type tests. Type tests are already inefficient in JAVA and they are significantly more inefficient in KERIS even though the compilation scheme does not use slow tree walking procedures for performing a type inclusion test.

To determine the overhead of type tests regarding (dependent) module member types, again several versions of the benchmark program shown in Figure 3.15 were created and compared. One version tests for a class, interface, or class field type defined within a module, another version tests for a top-level class type, and the third version tests for a top-level interface type. We only discuss the numbers of the JIT case.

According to the benchmark results in Figure 3.15, testing for interface types is almost 1.6 times slower than testing for class types. A test for a module member type is 38 times slower than a test for a class type and 24 times slower than a test for an interface type. These numbers suggest an overhead of 97% which is spent in the type test method generated by the KERIS compiler.

Similar to our investigations for method dispatches, we conducted a second experiment which makes optimizations more difficult by varying the object involved in the type test. This change of the experiment did not affect the results obtained for the JVM. For the JIT it had a small effect on the time measured for testing class types: it is now equal to the time measured for testing interface types (which itself did not change significantly compared to the original experiment).

Please note that the results of this benchmark can only be used to draw a conclusion for flat class hierarchies. Depending on the type test implementation within the Java Virtual Machine, deep class hierarchies or the use of multiple interface inheritance could degrade the JVM runtime performance for tests on class and interface types. Since the type test implementation for module member types uses the built-in JAVA type tests, this will also affect module member types, but the relative overhead will decrease. Thus, the presented benchmark can be seen as a worst case scenario.

```
module Bench5 {
  module A;
  Object o = new A.C();
  void main(String[] args) {
    A.C c;
    int i = 0;
    while (i++ < 10000000)
      c=(A.C)o;
  }
}
module A {
  interface I { I(); }
  class C implements I = {}
}
```

	JIT	VM
Mod. member	6333	67910
Class	112	1869
Interface	243	1878
Overhead		
~ Class	6221	66041
	≅ 98%	≅ 97%
~ Interface	6090	66032
	≅ 96%	≅ 97%

Figure 3.16: Costs for type casts to JAVA classes, JAVA interfaces, and class abstractions nested in KERIS modules (in ms).

Type casts. Since type casts to module member types are implemented using the type test procedure benchmarked in the previous paragraph, we can expect a similar slow down for casts. Therefore it is not surprising that according to the benchmark results in Figure 3.16, casts to class field types are 57 times slower than casts to class types and 26 times slower than casts to interface types. Casts to interface types are almost 2.2 times slower than casts to class types.

Similar to the treatment of method invocations and type tests, we developed a second benchmark where the object which is cast to a class, interface, or class field type, changes from iteration to iteration. And again we made the observation that nothing changes for the JVM, and for the JIT, only the performance for class types degrades (compared to the simple benchmark illustrated in Figure 3.16). In the second benchmark, the time measured for casts to class types corresponds exactly to the time measured for casts to interface types.

Note that again, the results only apply to "flat types." For deeper class hierarchies or multiple interface inheritance, the built-in cast mechanism of JAVA could perform worse improving the relative difference to the cast mechanism for module member types. But generally, the extensive use of type casts and type tests will yield a noticeable slow-down of KERIS programs.

Discussion. The micro benchmarks are only useful for measuring the overhead of specific language mechanisms of KERIS in relation to JAVA. They indicate reasons for slow-downs of extensible KERIS programs compared to similar, but more static JAVA programs. The micro benchmarks cannot be used to make reliable predictions about the actual performance of bigger real-world applications. We will compare the performance of a bigger KERIS program with its JAVA counterpart in the next section.

3.5.2 Real-World Application

Chapter 4 discusses the topic of extensible compilers and presents two implementations of the extensible JAVA compiler JACO. One implementation is written in a JAVA dialect using an object-oriented architectural design pattern supporting extensibility. The second implementation, we call it JACO2, is a port of the first implementation to KERIS. It makes intensive use of extensible modules and class fields to keep the system open for future extensions. The details of the two systems will be discussed in Chapter 4. Here, we will compare the runtime performance of the two systems empirically.

Since it is always difficult to compare two programs written in different languages using different abstractions, we will first present a comparison of the two programs based on a selection of code metrics. This comparison will help understanding the final benchmark results better.

Code comparison. Since JACO2 is simply a port of JACO to KERIS, both systems are very similar. They are both composed out of the same components, the algorithms and data structures are the same, and data and control flow are almost identical. They differ in the way compiler components were written and interlinked to compose the full system. While JACO uses the architectural design pattern *Context/Component* to configure the system in an extensible fashion, JACO2 makes extensive use of the module system of KERIS.

Figure 3.17 presents some statistical data about the size of the two programs. The number of files does not say much about the size of a program. In JAVA it is common to define a single class in a single file, while in KERIS it is more natural to have one module per file where a module contains several classes, interfaces, etc. According to the number of source code lines in Figure 3.17, JACO is almost 24% bigger than JACO2. "Lines of code" is an intuitive measurement for the size of a program, but the number of tokens, i.e. the number of terminal symbols, is a more reliable figure for measuring objectively the size of source code. This number is not subject to a particular programming style. With a total number of 152590 tokens, JACO's source code is 7% bigger than JACO2's code which consists

	source files	source lines	tokens	generated classes	generated methods	modules	classes	methods	$access calls	removed $access
JaCo	96	30269	152590	216	1757	—	—	—	—	—
JaCo2	66	24143	142774	398	4588	63	132	1495	2165	746

Figure 3.17: Quantitative comparison of JACO and JACO2

	1 file		*47 files*		*326 files*	
	JIT	VM	JIT	VM	JIT	VM
JaCo	4371	5128	6316	9505	13822	41817
JaCo2	6379	5852	9335	11182	19125	47384
JaCo2 (unopt.)	6640	5981	9734	11476	20064	48347

Figure 3.18: Comparison of JACO and JACO2 applied to input of different size (in ms).

of 142774 tokens.

These numbers indicate that the use of the module system of KERIS can help to reduce the size of the code that otherwise has to be manually written to define and glue software components. In the given case, JACO's whole *Context/Component* logic (see Section 4.2.1 and 4.2.3) disappears for JACO2 which uses native modules instead of the design pattern-based approach of JACO. The saving in code size appears even higher if one also considers that the use of class fields in KERIS encourages coding against interfaces instead of using classes directly. This can lead to extra code required for the definition of additional interfaces and the introduction of helper methods like "getter" and "setter" methods.

For binaries, Figure 3.17 shows a different picture. Here, JACO gets compiled to 216 classfiles while the smaller source code of JACO2 yields almost 400 classes. This is an increase of 84% compared to JACO. Of course, most of these 400 classes are generated automatically by the KERIS compiler and are not hand-written. JACO2 consists of 63 hand-written modules that contain 132 class, interface, and class field definitions. Since every module implicitly defines 4 classes, these 63 modules expand to 252 classfiles. The rest of the 146 classes is either hand-written or generated automatically from anonymous class field implementations. The extraordinary high number of 4588 methods (compared to 1495 hand-written methods) can also be attributed to the module translation. As mentioned in 3.4.3, the scheme for creating access methods to submodules can yield quite a lot of automatically generated methods.

The last two columns in the table of Figure 3.17 judge the optimization of `$access` calls by the KERIS compiler. From totally 2165 `$access` calls, the KERIS compiler safely removes 746 calls, which corresponds to almost 35% of the total number of calls.

Benchmark. The runtime of the two compilers was measured by compiling the *Jakarta Bytecode Engineering Library* (BCEL) version 5.0 developed by Markus Dahm. This library is fully written in JAVA. It consists of 326 files in which 360 classes with 2980 methods are defined. The library comprises 54401 lines of code with a total number of 151274 lexical tokens.

Both JACO and JACO2 were applied to the whole library, to the subpackage `org.apache.bcel.classfile`, and to a single file `Utility.java` of the same sub-

	1 file	*47 files*	*326 files*
$access calls	180035	547018	3217335
class field instantiations	4997	13768	66068
casts to class field types	31612	104638	552175
class field type tests	392	25134	149016

Figure 3.19: Statistics about the frequency of specific language construct invocations.

package. The aim of having 3 compiler runs with different input size was to find out the relationship between the execution time and the performance overhead of JACO2.

Figure 3.18 presents the results of the different compiler runs. For completeness, it also includes the results for an unoptimized version of JACO2. In all three cases, the `$access` call optimization improves the total runtime by about 5%. For the whole library this is almost an improvement of one second which would otherwise have been spent uselessly in almost empty `$access` methods. These numbers allow an estimation of around 10% of the overall time being spent in `$access` method calls that have not been removed by the optimizer of the KERIS compiler.

We now turn to a comparison of the runtime costs for JACO and for the optimized version of JACO2. For a single file of 1356 lines of JAVA code and 5553 tokens, the virtual machine executes JACO2 faster than the just-in-time compiler (JIT) does. This is an indication that the JIT is doing a lot of work which does not pay off in the end; i.e. it compiles many files to native code, but does not execute the generated code after that. The fact that the JACO2 run is almost two seconds slower than the corresponding JACO run might also indicate that the JIT is doing less superfluous work for JACO. The probably most likely reason for the bad performance of JACO2 is the number of classes that are involved. JACO2 consists of almost the double number of classes — it has 182 classes more than JACO, which all have to be loaded and processed by the JIT.

With more files to compile, the relative overhead of JACO2 gets, as expected, smaller. For the package `org.apache.bcel.classfile` which consists of 47 files with 10344 lines of code and 27215 tokens, JACO2 is slower by a factor of almost 1.5 in the JIT case, and 1.2 in the VM case. For the whole library, the overhead shrinks to a factor of 1.4 for the JIT, and 1.1 for the VM.

Figure 3.19 gives some more insights about the performance degradation of JACO2. It presents some statistics about the frequency of some language construct invocations. According to this figure, there must have been a quite high runtime overhead caused by type casts and type tests; together there are more than 700000 type casts and tests for the compiler run with 326 files. In Section 3.5.1 we determined that the execution of one million casts takes 6.2 seconds longer for class fields than for JAVA classes. For type tests this number was around 5 seconds (JIT case). If one assumes that in this benchmark the numbers

are similar, we get an overall overhead of 4188ms time being spent in casts and type tests for the compiler run with 326 files. Now if one considers that according to Figure 3.18, the JACO2 compiler run took 5303ms longer than the execution of JACO, it gets clear now that this overhead originates predominantly in the low performance of KERIS' type test facility.

Conclusion. Overall we can draw the following conclusions from this experiment:

1. In our experiment, JACO2 has an empirical runtime overhead of approximately 27% compared to JACO.
2. In the experiment, the biggest costs for JACO2 can be ascribed to the low performance of the runtime type test and type cast facility of KERIS.
3. For shorter program runtimes (e.g. small programs), relatively high runtime penalties are caused by the significantly higher number of classfiles to load and to compile. The decrease of the runtime overhead of JACO2 with growing input indicates this in particular.
4. The remaining costs can be attributed to the invocation of `$access` methods which are used to implement lazy module initialization.

An increase in the efficiency of JACO2 could be obtained by implementing type tests and type casts more efficiently. It is unclear if it is possible to devise a significantly better runtime type support for KERIS. Most type casts are necessary in JACO2 because of the extensive use of JAVA's collection classes. With generic types most explicit type casts should disappear.

The runtime overhead in general could be reduced by representing module contexts generically with, for instance, hash tables. This would save 3 classes per module being loaded dynamically, but it would introduce many more type casts in the code. The current implementation scheme was implemented with an efficient inner class handling on the classfile level in mind. At the time the compilation scheme was designed, such an optimized handling of nested classes was in discussion by Sun Microsystems, but it seems to have been rejected by now.

Giving up lazy module initializations would be another measure slightly enhancing the runtime performance of KERIS programs in general. Unfortunately, it is difficult to define a satisfying strict module initialization scheme which does not provoke a disproportionally high access rate to members of uninitialized modules. A strict module initialization scheme seems to work well only if recursively dependent module initializers are abandoned. Checking this statically is possible for "simple" module initializers (which do not call external or overridable internal functions, instantiate objects, etc.). Mutually dependent modules could be retained as long as the module initializers are not involved in the recursion.

3.6 Discussion

The previous sections of this chapter presented the design of the programming language Keris, its applications, and its implementation. To conclude the chapter we briefly review the design of the module system of Keris and relate it to module systems of other programming languages. Before presenting details of this discussion we give a quick introduction to module systems and clarify module system related terminology.

3.6.1 Module Systems

3.6.1.1 Motivation

Many programming languages support programming in the large through a module system. While the nature of module systems can differ quite significantly between different programming languages, all module systems basically provide at least some of the following functionality [37, 100]:

1. *Code organization:*
   Modules provide a way of breaking up code into manageable pieces that can be designed, implemented, and understood in isolation.

2. *Name space management:*
   Since modules open new name spaces they help to structure names and help to avoid name clashes when modules are combined.

3. *Abstraction control:*
   Modules provide important abstraction barriers, grouping together declarations that fit logically, and hiding information about implementation details from clients. The smaller the dependency of a module on implementation details of other modules is, the easier it is to replace modules with different implementations or new versions of former implementations.

4. *Separate compilation:*
   Support for separate compilation aids the programmer in developing systems incrementally, and makes debugging of large programs practical. With separate compilation it is possible to create reusable code in libraries in such a way that the source code need not be available to clients.

5. *Code reuse:*
   The ability to use a module in different applications with possibly different cooperating modules helps reusing code. Reusing a module multiple times within the same application requires modules to be *generic* in the context dependencies.

3.6.1.2 Design Issues

Since different module systems often achieve the goals above with quite different language mechanisms, we now review typical design issues, and discuss trade-offs in the design of module systems. The following coverage is kept on a relatively abstract level, since in general, it is very difficult to clearly talk about more than one module system at a time.

Language level. Module systems are small domain specific languages dedicated to modularization. They are usually built on top of a *core programming language*. One aim in module system design is to keep the module language level as independent as possible from the core language level [12, 110]. The purpose of this aim is to allow the module language to be used with different core languages — a goal which has not been achieved yet in practice.

Module systems where modules are regular values of the core language are called *first-class*. In most module systems, modules are *second-class* in the sense that they are only values on the level of the module language.

Information hiding. Modules are program units that *encapsulate* a set of logically or physically related program entities. Typically, the programmer has to specify explicitly which of these entities are *exported* such that they are accessible to external clients (e.g. other modules). This is either done by tagging the definitions with access modifiers (e.g. like in OBERON-2 [146]), or the programmer has to write a separate *module interface* which specifies the *external view* on the entities to export. Some module systems allow to specify multiple module interfaces per module. This feature is, for instance, supported by the module systems of SML [123] and MODULA-3 [44]. It is mostly used to grant different access privileges to different clients.

Similarly, some module systems require that every interface has exactly one module implementation, while others allow the definition of multiple implementations for a single module interface.

In some module systems (e.g. MODULA-3, LOOM [37]), modules have to declare explicitly what interfaces they implement, in other module systems this is implicit, for example through a naming scheme (e.g. MODULA-2 [206]). There are finally module systems where a module automatically implements all module interfaces with structurally matching signatures (e.g. in SML). This purely structural approach has the advantage that all possible views of the module do not have to be anticipated, but this approach is also often criticized for falsely associating a module with an interface that coincidentally defines entities with matching signatures but different semantics.

Name spaces. In all common module systems, modules open a new name space such that names defined by the module cannot clash with names of enti-

ties defined in other modules. Access to entities exported by a module is typically possible via the *dot notation* [45] — that is, $m.f$ referring to the operation f provided by the module abstraction m. Some module systems allow to either import the whole exported scope of an external module into the scope of another module (e.g. Java), or selectively allow to make only some module members of an external module accessible without explicit qualification with a reference to the external module abstraction (e.g. Modula-2, Modula-3, Java). Since such scope embeddings can often lead to name clashes, some module systems also provide facilities for renaming abstractions, or for introducing local aliases (e.g. Oberon-2, C# [86]).

Dependencies. Modules typically depend on services provided by other modules. In most module systems this is expressed as a dependency on other external modules. In *unit*-style module systems [71, 64] modules directly depend on abstractions provided by other modules; i.e. modules import entities defined in other modules, they do not directly refer to other modules.

In most module systems, dependencies on external modules or entities of external modules have to be stated *explicitly*. An exception to this is, for instance, the package system of Java where dependencies are *implicit* and automatically inferred by the compiler.

A module is usually free to depend on any other module. Exceptions are module systems, like SML's, which require systems to be built strictly incrementally, ruling out recursive dependencies between modules. This can impede modular programming by forcing mutually dependent components to be consolidated into a single module, which partially undermines the very idea of modular organization [54].

Genericity. In classical module systems for imperative languages, modules depend on other modules specifically; i.e. they reference *specific*, fixed external modules. While in such *first-order* module systems, references to other modules are hard-wired, module systems of functional languages typically allow to abstract over context dependencies. Here, references to the external world are *generic* [163]. Module systems with generic modules are also called *higher-order* module systems.

Composition. Before modules with generic context dependencies can be deployed, they have to be *linked* or *composed* with other modules until all context dependencies are resolved. Because of such higher-order modules it is important to distinguish between generic modules which abstract over their dependencies and concrete *module instances* where the generic dependencies are replaced with concrete ones.

In ML, modules with generic dependencies are expressed as functions mapping module instances to module instances. Such functions are called *functors*,

the concrete module instances are called *structures*. Modules are composed by functor applications.

In unit-style module systems, modules abstract over external entities like values, functions, and classes. Modules are composed by "wiring" exported entities explicitly with imported ones so that all context dependencies are satisfied. As opposed to functors, unit-style modules support recursive dependencies. Component-oriented programming languages typically provide linguistic abstractions for software components in a unit-style fashion [182, 187, 5].

Aggregation. To cope with the complexity of large software systems, it is not sufficient to simply divide them into smaller pieces because the pieces themselves will either be too numerous or too large [25]. A *hierarchical* modular structure is the natural solution where modules aggregate *submodules* and thus smaller modules are composed to form bigger ones.

Type abstraction. Strongly typed programming languages with module systems have to provide some sort of modular type abstraction mechanism. Depending on this mechanism, module systems can be classified as either *opaque* or *transparent*.

Opaque systems totally hide information about the identity of type components, requiring that all operations on the type must be defined within the scope of the module where the type is defined. Generally, opaque types behave as unique types which have no relationships to other types. They are useful for programming abstract datatypes where all specifics of the implementation are hidden from clients. This makes it simple to guarantee invariants and to replace the implementation with a compatible one, but the loss of type identities also precludes some interesting uses of higher-order module features, e.g. building new ADTs on top of old ones [118].

As opposed to this, transparent systems completely reveal type identities by inspecting module implementations. This subverts data abstraction and prevents separate compilation [118]. On the other hand, transparent types have an important potential: they allow sharing of type information across module boundaries [160]. An example for a fully transparent module system can be found in SML'90, the first release of SML.

Between the two extremes, there are module systems supporting the notion of *partial abstraction*. Partial type abstraction is the idea of allowing some information to escape the abstraction barrier, while keeping the essential implementation abstract. Types which allow such a *partial revelation* are also called *translucent types* [84, 118]. They combine the benefits of the two extreme approaches without adopting the disadvantages. OCAML's *manifest types* [109] are, for example, translucent types where the module interface fully specifies what information about the type is revealed. MODULA-3 goes even a step further; it features a module system which supports the notion of *incremental revelation*. Here,

information about the identity of the type can be revealed step by step even in clients of the module that defines the type.

Generativity. Module systems with higher-order modules support multiple instantiations of a module. In the presence of abstract types the question arises if the types provided by two different instances are the same or distinct.

In SML, functors are *generative*, meaning that the types in structures generated by functor applications are generally distinct; i.e. every runtime instance of a module abstraction gets "new" member types. On the contrary, functors in OCaml [111] are *applicative*, meaning that equal arguments in a functor application yield module instances with equal types.

Modula-3 offers one of the few module systems for imperative languages that has support for generativity. But here, generic modules are *templates* that get expanded by a macro expansion mechanism at link-time. Similar to the template mechanism of C++ [189], Modula-3 does not typecheck such "module templates."

Coherence. Type equality is also a relevant issue when different modules interact. When two interacting modules both refer to a third module, it is important to know whether both refer to the same module instance or at least whether both share the type components that are relevant for the interaction. Otherwise data exchange between the two modules involving types of the third module would not be statically safe. Module systems deal with such *coherence* issues quite differently.

In simple module systems without higher-order modules, every module is conceptually a *singleton* with a distinct name (e.g. in Modula-2, Component Pascal [207]). In such a setting, modules referring to an external module with the same name will also refer at runtime to the same module instance — this is the essence of specific module references. Thus, two types are equal if their fully qualified names are equal.

Pierce distinguishes two mechanisms dealing with coherence in module systems featuring generic module references: *sharing by parameterization* and *sharing by specification* [163]. Sharing by parameterization comes in two flavors: *parameterization on external modules*, and *parameterization on external types*. In the first approach, modules and module interfaces are parameterized with all external modules. Effectively, this approach, which is also called *fully functorized style*, implies that modules are "maximally parameterized" relating all modules of the dependency closure. Ultimately, this means that arbitrary patterns of sharing have to be anticipated [163]. Technically, coherence is controlled in this approach by a compile-time notion of equivalence of module instances. Only types of module instances that can statically be identified are considered to be equivalent. This form of parameterization is supported by SML'90.

Module systems that abstract directly over types of external modules address sharing implicitly through the choice of the module parameters. While with deeper dependency hierarchies, interfaces parameterized on modules scale badly, *parameterization on external types* does not suffer from this problem, as Pierce points out in [163]. MZSCHEME's *units* rely on such a parameterization scheme, as well as component abstractions of many component-oriented programming languages.

The idea of the sharing by specification approach is to either augment modules with submodules which refer to corresponding external modules, or to augment modules with external types. Together with a facility for specifying sharing of submodules, or sharing of types of different modules, it is possible to address the coherence problem in a post-hoc fashion. Following these ideas, the programming language SML'97 explicitly supports *sharing constraints* to identify types and thus allows modules to interoperate flexibly with data of a shared type.

Separate compilation. An important feature of many module systems is the support for separate compilation. This is achieved by ensuring that all interactions among modules are mediated by interfaces which capture all information required to compile clients without revealing implementation details. If some essential information is missing in the interface of a module, then clients that depend on this missing information have to refer directly to the corresponding module implementation, possibly compromising separate compilation if the dependency between the client and the module implementation is mutual recursive. Transparent types make separate compilation impossible for the same reason. They reveal their concrete implementation fully through the module implementation so that a clear separation between module interface and implementation gets impossible.

Module systems that do not require the specification of explicit module interfaces and instead read the interfaces off the module implementation support separate compilation in general only, if they do not allow mutual dependencies.

Separate compilation is essential to enable the *binary* reuse and the binary distribution of software components.

Reuse. Modules with generic context dependencies can be *instantiated multiple times* within one software system. This form of reuse, which is sometimes also called "as is"-reuse, is probably the most common form of reuse.

Some module systems provide facilities for reusing an existing module in the definition of a new one. *Module inclusion*, for instance, is a mechanism which inserts all definitions of an existing module or module interface into another module or module interface. Support for this can, for example, be found in the module system of OCAML. In combination with a structural type system, it allows programmers to easily create extensions of modules which add new module members. Modifications of existing members can be achieved with combinations of

mixin modules [31, 28]. Systems of mixin modules support cross-module recursion and overriding. In the context of ML, mixin modules can elegantly model recursive dependencies [88, 57].

Mixin module combinations can be *shallow* or *deep*. While shallow mixin composition mechanisms compose module members only on the top-level, deep module composition facilities recurse deeply into the definition of module members possibly fusing module member abstractions like submodules, functions, algebraic datatypes [57], and classes [94]. Such module systems support, to some extent, *aspect-oriented programming* [103].

3.6.2 Module Systems and Object-Oriented Languages

Classes as alternative modularization constructs. Pure object-oriented languages like Smalltalk [80] and Eiffel [133] typically identify classes with modules. This is unfortunate since the primary role of classes is to define types and to serve as extensible generators of objects. As described in the last section, modules are typically associated with quite different purposes like name space management, provision of abstraction barriers, and abstractions targeted towards separate compilation.

To compensate for this, newer object-oriented languages have class abstractions that share many features with module systems. One of these features is class nesting which allows for structuring the name space of classes and for grouping closely related classes. Another feature are access modifiers which give control over the visibility of class members, and thus help to set up abstraction barriers. Often, classes are also associated with compilation units.

Shortcomings of classes as modularization constructs. However, support for modularity is in general very rudimentary and sometimes pretty awkward [37]. Imagine packaging a collection of mathematical functions in a class. For using these functions, one must first create an object of that class and then call the functions by sending messages to the object, even though the methods are not in any way specialized to the object. Furthermore, this object has to be threaded through all parts of the system in which mathematical functions are required (unless one wants to permanently re-create the object — an approach which would be even more awkward). This task can be quite challenging especially since this often requires some form of anticipation. Therefore, most object-oriented languages have ad-hoc mechanisms like static methods or class methods to address these shortcomings. Every access to a static class member introduces a hard link. Thus, every module system supporting generic dependencies is already more expressive with respect to reuse and substitutability, allowing to instantiate a program unit with possibly different implementations of the mathematical library. Furthermore, module systems allow a much more natural way for accessing such intrinsically static features in comparison with artificial class

construct extensions.

Class abstractions are also very limited when it comes to express relationships between different classes. In module systems, context dependencies are typically explicit so that a programmer can easily see on what other components a module depends. In class-based object-oriented languages on the other hand, such relationships are implicit. The user of a class has no clue what other cooperating classes are involved (e.g. when packaging a library or a subsystem) unless he inspects code by hand or uses specific code inspection tools. While class nesting provides a means to encapsulate tightly coupled classes, it is difficult to split up such a class set into smaller subsets such that each of these subsets can be compiled separately, but access to private class members of classes in other subsets is still allowed. This is mainly because in most object-oriented languages access to class members can only be controlled in a very coarse grained way through a fixed number of predefined access levels like `public`, `protected`, and `private`. To some extend, ad-hoc features like *friends* [189] can help to overcome some limitations, but the use of such completely arbitrary constructs can result in what Szyperski calls "spaghetti scoping" [192].

The most conspicuous difference between class systems of object-oriented languages and module systems in general is the lack of class or type abstraction mechanisms in object-oriented languages. This is a show-stopper for developing extensible object-oriented software in a statically type-safe way, since references to classes are always static, making it impossible to use compatible classes or class extensions instead without hacking source code destructively. Furthermore, types cannot be adapted so that extensions refer to more refined types through which they can access extended features. While some design patterns can help to avoid hard links to classes, type refinements can only be enforced via type casts that in principle bypass the static type system. Only recently, strongly-typed object-oriented programming languages appeared that alleviate the shortcomings by supporting powerful type abstraction mechanisms. Examples are GBETA [58], SCALA [153, 151], and FAMILYJ [156, 208]. Only in such languages, classes provide a serious alternative to classical modules.

Modules and classes as complementary abstractions. While there is a trend to incorporate more and more features of modules into classes, the programming language MOBY [66] tries to provide both modules and classes, but without duplicating features. The design philosophy is to keep language features independent, but complementary [65]. MOBY is a statically typed ML-like language that supports class-based object-oriented programming with simple class abstractions that only support those mechanisms that are intrinsic to classes, namely object creation and inheritance. It relies on the module system to support namespace management and visibility control [68, 67]. The resulting design is indeed quite expressive, but the combination of some language features, like the coexistance of structural and nominal subtyping, might be confusing for some programmers.

3.6.3 Keris

In the following we review the design of KERIS in the framework laid out in the previous section.

Language level. The module system of KERIS is built on top of a core language which more or less corresponds to JAVA. Modules are second-class; i.e. they are not regular values of the core language which can be computed or passed around with core language constructs. In support for reflection, it is possible to package up a module as an object and use operations of a reflection API to inspect or modify the configuration of a module dynamically within the core language.

Information hiding. Modules in KERIS encapsulate submodules, class abstractions (classes, interfaces, class fields), variables, and functions. With the help of the access modifiers `public` and `private`, a programmer can specify explicitly what module members are exported and which ones are hidden from clients.

KERIS also allows programmers to define module interfaces which describe the signature of exported module members like submodules, functions, and class fields without giving a concrete implementation. Module interfaces typically also contain regular JAVA interface definitions. In KERIS, modules have to specify explicitly what module interfaces they implement. A module may implement several module interfaces and a module interface may be implemented by several modules.

Name spaces. Modules in KERIS either open a new name space, or they extend an existing name space when modules are refined or specialized. Members of modules are dereferenced via the dot notation. The `::` operator is used to access submodules. It is possible to import the name space of an external module or a submodule via the star-import statement. One can also selectively import only specific submodule references.

Dependencies and composition. In KERIS, modules are explicitly parameterized with external modules. These context dependencies are generic, allowing modules to be instantiated multiple times with different cooperating module instances. When such higher-order modules are deployed as submodules, they get automatically linked with matching modules of their deployment context; i.e. they are linked to other submodules or modules that are required by their host module. In other words, the module wiring is inferred rather than manually specified by a programmer. This "wiring inference" is based on two principles:

1. Module instances are generally identified by the name of their corresponding module definition, and
2. Every module context defines an unambiguous set of module instances (requiring that every generic module is instantiated only once per context).

This approach combines the simplicity of classical module systems for imperative languages with the advantages of modern, component-oriented formalisms. In particular, modules reference other modules via their declared name. This is similar to module systems with specific context dependencies where references to other modules are interpreted as "the module with this name and interface (whose precise identity will be known at link-time)." On the other hand, modules are reusable, generic software components that can be linked with different cooperating modules without the need for resolving context dependencies by hand. Like in simple module systems with specific context dependencies, modules can depend on each other mutual-recursively.

Aggregation. KERIS makes it easy to implement hierarchical modular structures with its built-in support for submodule aggregation. Thus, bigger compound modules are built from smaller compound or atomic modules incrementally. As explained before, the aggregation mechanism is combined with the instantiation and composition mechanism of submodules.

Class abstractions. KERIS strictly distinguishes class implementations from interface descriptions. Both are immutable building blocks for class fields which associate the two in an extensible way. A class field introduces a new nominal type together with an implementation for that type. Every class field declaration mentions all interfaces implemented by possible implementations. It also mentions other class fields which define supertypes. Types defined by class fields are translucent in the sense that implemented interfaces and supertypes can be incrementally revealed to clients. The class field mechanism of KERIS disables a complete revelation of the concrete type implementation. Such a manifestation of a type implementation would ultimately rule out further extensions.

Extensibility is a distinguished feature of class fields in KERIS. In module refinements and specializations it is possible to covariantly override the set of implemented interfaces and the set of supertypes. It is also possible to fully exchange the implementation of a class field in a non-invasive fashion. Extensible class abstractions like class fields are essential for evolving object-oriented software. Due to the central role of class fields in KERIS, there is also support for representing class field types at runtime. These runtime type representations are required to implement type casts and type tests correctly.

Generativity and coherence. Modules in KERIS are generative so that conceptually every instantiation of a module gets a new set of member types. Type coherence is managed by a dependent type system where types depend on concrete module instances. Two types $P.T$ and $Q.T$ are considered to be equal if the two module paths P and Q refer to the same module instance. For such a kind of reasoning, the type system has to be equipped with a compile-time notion of module instance equivalence.

Separate compilation. In general, modules can be compiled separately if solely module interfaces mediate between different modules. Otherwise the compiler will directly read the interface off a module definition. As long as systems are built incrementally without recursive dependencies between module implementations, separate compilation is still possible in the presence of such inferred module interface descriptions. Only module implementations with mutual-recursive dependencies have to be compiled jointly.

Extensibility. Modules are extensible in KERIS so that functionality can be modified or added. KERIS has support for two different forms of extensibility on the module level:

1. A new version of an already existing module can be created for updating or adding module members. New versions of modules subsume old versions in the sense that one can replace an instance of an old version with an instance of a new version. This form of reuse is called *refinement*.
2. A physically and logically new module can be derived from an existing parent module by inheritance. The reuse of existing modules as "prototypes" for new modules, which customize the prototype for a specific application, is called *specialization*.

Both forms of reuse are based on inheritance. By allowing submodule definitions to be overridden, it is not only possible to extend atomic modules but also compound modules representing fully-linked subsystems. When a submodule is overridden, one does not have to re-link the whole system again. The wiring inference mechanism of KERIS integrates new versions of submodules easily in a plug-and-play fashion.

Since context dependencies of modules can also be refined or specialized, it is straightforward to consistently extend sets of cooperating modules. This is an important requirement for evolving large systems consisting of many interconnected components safely and modularly.

Since the extensibility mechanism is based on inheritance, modules are refined and specialized in a non-invasive way, in which old versions persist when new versions are derived. Furthermore, new and old versions of modules can coexist in different contexts even within the same system.

Chapter 4

Case Study: Extensible Compilers

In this chapter we study the design and the implementation of extensible compilers to illustrate how non-trivial real-world applications can be made extensible and to show what problems programmers are typically facing when developing extensible applications. We first present a general discussion of compilers and extensibility related issues. Then we focus on a particular design of an extensible compiler architecture. We describe the implementation of two extensible JAVA compilers which are both built according to this design. The first implementation is based on a general object-oriented architectural design pattern and uses mainstream object-oriented language features in its implementation. The second compiler is written in KERIS, making intensive use of extensible modules and the refinement and specialization mechanism of KERIS. The chapter concludes with a qualitative and quantitative comparison of the two systems.

4.1 Introduction

Traditionally, compilers are developed for a fixed programming language. Consequently, extensibility and reusability of compiler components are often considered to be unimportant properties. In practice this assumption does not hold. People constantly experiment with new language features. They extend programming languages and build compilers for them. Writing a compiler for such an extended language is usually done in an ad-hoc fashion: the new language features are hacked into a copy of an existing compiler. By doing this, the implementation of the new features and the original implementation get mixed. The extended compiler evolves into an independent system that has to be maintained separately.

To avoid such a destructive reuse of source code, compilers have to be extensible so that they can be specialized easily for integrating language extensions [170, 215, 149]. This chapter discusses two implementation techniques for extensible compilers. In both approaches, extended compilers reuse components of their predecessors, and define new or extended components without touching any predecessor code. All extended compilers derived from an existing base compiler share the components of this base compiler. This approach provides a basis for easily maintaining such a family of systems.

Before discussing details of the two different compiler implementations, we look at the traditional organization of compilers and motivate problematic issues implementors of extensible compilers are typically facing. We identify basically two problems:

1. Abstract syntax trees and compiler phases operating on these trees have to be extensible, and
2. Compilers have to be re-configurable so that compiler components can be updated and new compiler components can be integrated.

In this section we explain a solution for handling the first problem. We will use the same approach in both of the two compiler implementations. The second problem will be solved differently and is therefore discussed separately for the two compilers in the following two sections 4.2 and 4.3.

4.1.1 Extensibility Problem

Traditionally, the compilation process is decomposed into a number of subsequent phases, where each phase is transforming the program from one internal representation to another one. These internal representations are implemented as abstract syntax trees. Compiler phases are operations that traverse the trees. Figure 4.1 gives an overview over the general structure of a compiler including typical components.

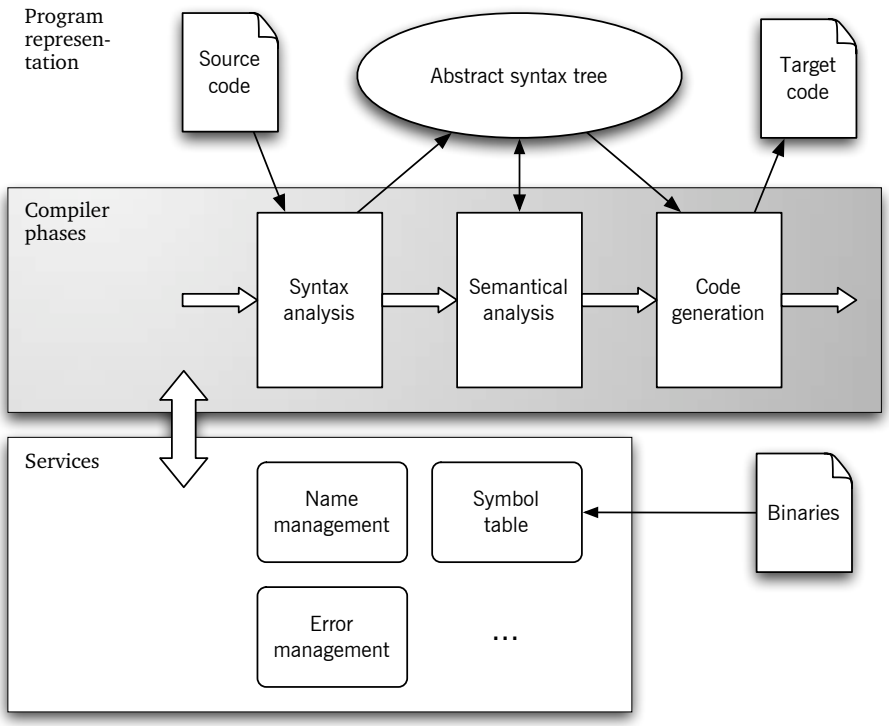


Figure 4.1: Structure of a multi-pass compiler.

An extension or modification of the compiler's source language often requires both, extensibility of the datatype modeling the abstract syntax and the set of phases operating on this type. Furthermore it is often necessary to adapt existing phases. This well-known problem of extending data and operations simultaneously is called the *expression problem* or the *extensibility problem* [50, 51, 64, 70, 78, 106, 203].

Unfortunately, neither a functional nor an object-oriented approach solves this problem in a satisfactory way. With an object-oriented language such a datatype would be implemented as a set of classes sharing a common interface. We call these classes *variants* of the datatype, the methods of these classes implement the operations. While extending the datatype is simply done by creating new variant classes supporting the common interface, adding new operations is tedious. New operations require that either all existing variant classes are subclassed, or that they get modified in a destructive fashion.

In a functional language, the variants of a datatype are typically implemented with an algebraic type. Ordinary algebraic datatypes cannot be extended, so it is not possible to add new variants. On the other hand, writing new operations is simple, since operations are simply functions over this type.

Each of the two approaches can encode the other. In one direction, object-oriented languages can model the functional approach using the *Visitor* design pattern [74]. In the other direction, objects can be represented in functional languages as closures taking an algebraic type of messages as parameter. How-

ever, each of these encodings exchanges the strengths and weaknesses of one approach with the strengths and the weaknesses of the other; neither encoding gains simultaneous extensibility of both data and operations.

4.1.2 Related Work

Several attempts to solve this problem are published. MULTIJAVA's *open classes* tackle the shortcomings of the object-oriented approach in a pragmatic way [50]. Open classes allow the programmer to add new methods to existing classes without modifying existing source code and without breaking encapsulation properties. This approach provides a clean solution to the extensibility problem, but in practice, it still suffers from a few drawbacks. Whereas a new operation is typically defined in a single compilation unit, modifying an existing operation can only be accomplished by subclassing the affected variants and overriding the corresponding methods. This leads to an inconsistent distribution of code, making it almost impossible to group related operations and to separate unrelated ones. Furthermore, extending or modifying an operation always entails extensions of the datatype. This restricts and complicates reuse. For instance, accessing an extended operation in one context and using the original operation in another one cannot be implemented in a straightforward way.

For functional programming languages, various proposals were put forward to support extensibility of algebraic datatypes. Among them, the most prominent ones are Garrigue's *polymorphic variants* [77] and the *extensible types* of the ML2000 proposal [13]. In [214], both approaches are compared with the one presented in the next section. Several papers discuss the extensibility of algebraic types in the context of building extensible interpreters in functional languages. Existing approaches like [112] and [63] allow a restricted form of extensibility: algebraic types are extensible, but the final datatype has to be closed before being used. Furthermore, extensions of datatypes always require updates of all existing functions to support the new variants. On the other hand, these approaches support the combination of orthogonal extensions. Basically the same holds for *mixin modules* proposed by Duggan and Sourelis [57].

The literature also describes several modifications of the Visitor design pattern which focus on extensibility. Krishnamurthi, Felleisen, and Friedman introduce the composite design pattern *extensible visitor* [106]. Their programming protocol keeps visitors open for later extensions. One drawback of their solution is that whenever a new variant is added, all existing visitors have to be subclassed in order to support this new variant. Otherwise a runtime error will appear as soon as an old visitor is applied to a new variant. Palsberg and Jay's *generic visitors* are more flexible to use and to extend with respect to this problem [157]. Since generic visitors rely on reflective capabilities of the underlying runtime environment, this approach lacks static type safety and is subject to substantial runtime penalties. Kühne's *translator* pattern relies on generic functions performing

a double-dispatch on the given operation and datatype variant [108]. As with the solution of Krishnamurthi, Felleisen, and Friedman, datatype extensions always entail adaptations of existing operations accordingly. Therefore Kühne proposes to not use the translator design pattern in cases where datatypes are extended frequently.

4.1.3 Extensible Compiler Phases with Algebraic Datatypes

Defaults. The fact that extra code is necessary to adapt an operation to new variants can be very annoying in practice. We made the observation that an operation often defines a specific behavior only for some variants, whereas all other variants are subsumed by a default treatment [214]. Such an operation could be reused without modifications for an extended type, if all new variants are properly treated by the existing default behavior. The experience with our extensible Java compilers shows that for extended compilers, the majority of the existing operations can be reused "as is" for extended types, without the need for adapting them to new variants [214].

If it would be possible to specify a default case for every function operating on an extensible type, a function would have to be adapted only in those situations, where new variants require a specific treatment. This technique would improve "as is" code reuse significantly.

Extensible algebraic datatypes. This section presents a solution to the extensibility problem based on the notion of *extensible algebraic datatypes with defaults*. These datatypes are described in the context of a JAVA-like, object-oriented language. The syntax is similar to PIZZA [154], but unlike PIZZA's algebraic types, it is possible to derive extended types from existing algebraic types by defining additional variants. This approach allows one to solve the extensibility problem in a rather functional fashion; i.e. in a way that strictly separates the definition of datatypes and operations on these types. Extensions on the operation side are completely orthogonal to extensions of the datatype. It is possible to apply existing operations to new variants, since operations for extensible algebraic types define *default cases* which handle all future extensions. In addition to adding new variants and operations, it is also possible to extend existing variants of datatypes, or to modify existing operations by subclassing and overriding. Extensibility is achieved without the need for modifying or recompiling the original program code or existing clients.

Example. We explain the implementation of extensible compiler phases with algebraic datatypes by introducing various fragments of a compiler for a small source language, only consisting of variables, lambda abstractions, and lambda applications. We use the syntax introduced by Pizza [154] and implement abstract syntax trees with the following algebraic type definition:

```
class Tree {
  case Variable(String name);
  case Lambda(Variable x, Tree body);
  case Apply(Tree fn, Tree arg);
}
```

We now define a type checking phase for the small source language in a separate class TypeChecker. Pattern matching is used to distinguish the different variants of the Tree type in the process method of class TypeChecker. In PIZZA, the switch statement is used to pattern match on objects of an algebraic type.

```
class TypeChecker {
  Type process(Tree tree, Env env) {
    switch (tree) {
      case Variable(String n):
        return env.lookup(n).type;
      case Lambda(Variable x, Tree body):
        ...
      case Apply(Tree fn, Tree arg):
        Type funtype = process(fn, env); ...
      default:
        throw new Error();
    }
  }
  ...
}
```

Adding operations. By using this approach, it is straightforward to add new operations (phases) to the compiler simply by defining new methods in possibly new classes. But it is also easy to modify an existing operation by overriding the corresponding method in a subclass.

```
class NewTypeChecker extends TypeChecker {
  Type process(Tree tree, Env env) {
    switch (tree) {
      case Lambda(Variable x, Tree body):
        ...
      default:
        return super.process(tree, env);
    }
  }
}
```

Class NewTypeChecker modifies the treatment of the Lambda variant and reuses the former definition for the other variants of the Tree type by delegating the call to the super method.

Adding data variants. As we saw, extending the set of operations and modifying existing operations does not pose serious problems. The only missing piece for solving the extensibility problem now consists in the extension of the Tree datatype with new variants. Pizza's algebraic types cannot be extended in that way. *Extensible algebraic datatypes with defaults* [214] on the other hand help to overcome exactly this problem. These datatypes allow to define a new algebraic datatype by adding new variants to an existing type. Here is the definition of an extended Tree datatype, which adds two new variants Zero and Succ:

```
class ExtendedTree extends Tree {
  case Zero;
  case Succ(Tree expr);
}
```

This definition introduces a new algebraic datatype ExtendedTerm consisting of five variants, Variable, Apply, Lambda, Zero, and Succ. One can think of an extensible algebraic datatype as an algebraic type with an implicit default case. Extending an extensible algebraic type means refining this default case with new variants. For the example above, the new type ExtendedTerm inherits all variants from Term and defines two additional ones. With our refinement notion, these two new variants are subsumed by the implicit default case of Term. As shown in Appendix C, this notion turns ExtendedTerm into a subtype of Term. Such a subtype relationship is crucial for code reuse, since it makes it possible to apply all functions for the original type to terms containing nodes from the extended type. Since existing functions perform a pattern matching only over the original variants, an extended variant is handled by the default clause of the switch statement.

Updating operations. For the current typechecking method, the default clause simply throws an Error exception. To handle the new variants correctly, we have to adapt the type checking phase accordingly. This is done by subclassing the original type checking component and by overriding the existing process method:

```
class ExtendedTypeChecker extends TypeChecker {
  Type process(Tree tree, Env env) {
    switch (tree) {
      case Zero:
        return IntType();
      case Succ(Tree expr):
        checkInt(process(expr, env));
        return IntType();
      default:
        return super.process(tree, env);
    }
  }
}
```

Benefits. The code fragments of this section demonstrate the expressiveness of extensible algebraic datatypes in the context of an object-oriented language like JAVA. As opposed to almost all approaches mentioned in Section 4.1.1 on page 150, extensible algebraic datatypes with defaults allow datatypes and operations to be extended in a completely independent way. An extension in one dimension does not enforce any adaptations of the other one. Since in a pattern matching statement, new variants are simply subsumed by the default clause, existing operations can be reused "as is" for extended datatypes. The presented approach supports a modular organization of datatypes and operations with an orthogonal extensibility mechanism. Extended compiler phases are derived from existing ones simply by subclassing. Only the differences have to be implemented in subclasses. All other functionality is reused from the original system, which itself is not touched at all. Roudier and Ichisugi refer to this form of software development as *programming by difference* [173]. Another advantage is that an operation for an algebraic datatype is defined locally in a single place. The conventional object-oriented approach would distribute a function definition over several classes, making it difficult to understand the operation as a whole.

4.2 JaCo: Design Pattern-Based Extensibility

This section discusses the extensible Java compiler JaCo. JaCo is written in an enhanced version of Java supporting extensible algebraic types with defaults. It uses the technique presented in the previous section to implement extensible types and components offering extensible functions operating on these types.

The approach from the previous section does not include a mechanism for gluing a certain combination of components and datatypes together to build extensible subsystems which are finally combined to a concrete compiler. In JaCo we achieve this by using the object-oriented architectural design pattern *Context/-Component* [210, 215]. This design pattern is specifically targeted towards building extensible, hierarchically composed systems. The architecture of JaCo combines the use of algebraic types with this object-oriented pattern yielding a compiler which can be flexibly extended without modifying source code and without anticipating all possible extension scenarios.

We first describe the Context/Component pattern, then we apply it to compilers, and finally we describe how the pattern is used in JaCo.

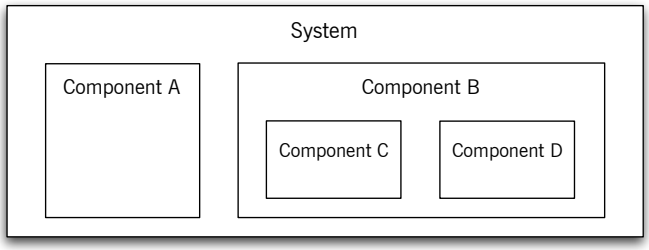
4.2.1 Architectural Pattern: Context/Component

Architectural patterns. Several design patterns for structuring a system are described in the literature. The pattern *Whole-Part* [41] builds complex systems by combining subsystems with simpler functionality. A *whole*-object aggregates a number of simpler objects called *parts* and uses their functionality to provide its own service. *Composite* [74] is a variant of *Whole-Part* with emphasis on uniform interfaces of simple and compound objects. A *Facade* [74] helps to provide a unified interface for a subsystem consisting of several interfaces, so that this subsystem can be used more easily. All these design patterns only target the structural decomposition of a system. They do not consider the fact that designs often require that the concrete implementation of some components or subsystems is not known at compile-time. For this reason, design patterns like *AbstractFactory* and *Builder* [74] have to be used in addition. They allow to configure instantiations at runtime, but they are also suitable for configuring a system statically so that it can be extended easily in future.

The Context/Component pattern. The architectural design pattern *Context/-Component* is supposed to facilitate the implementation of extensible hierarchically composed systems [210]. It separates the composition of a system and its subsystems from the implementation of the components. This principle makes it possible to freely extend or reuse subsystems. Among all design patterns mentioned before, Context/Component is the only pattern that offers a uniform way to compose, to extend, to modify, and to reuse components and subsystems in a non-invasive fashion while still being easy to implement manually.

4.2.1.1 Idea

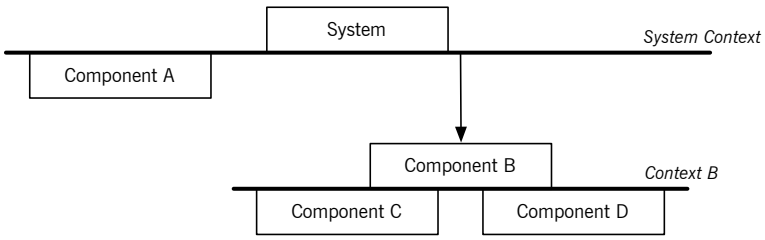
We suppose to implement a hierarchically structured component system. The following figure shows a system consisting of two components `A` and `B`. Component `B` represents a more complex subsystem which aggregates two local subcomponents `C` and `D`.



The main idea of the Context/Component pattern is to separate the configuration of a system from the implementation of its components and to make both artifacts independently extensible. We call the configuration of systems *contexts*. Formally, a context aggregates all components of a system or subsystem. Every component is embedded in exactly one context. It refers to this context in order to access the other components of the system. For this purpose, the context object offers a *factory method* [74] for every of its components. These methods specify the instantiation protocol of the different components. Typically, either a new instance of a component is created for every factory method call, or the component is a *singleton* [74] with respect to the context. In this case, the component is instantiated only once during the first call of the factory method.

Components that represent more complex subsystems, like component `B` from the example above, have an own local configuration. In other words, they are embedded in their own context which specifies all their local subcomponents. Thus, contexts have a nested structure. Every context might have subcontexts for more complex subsystems. The context in which an embedded subcontext is nested, is called the *enclosing context*. Components defined in a nested context can access the components of their own context and all the components defined in enclosing contexts. On the other hand, it is not possible for a component to access components defined in subcontexts directly.

Following [210], we illustrate the structure of a system in terms of the Context/Component pattern with a graphical notation. For the scenario mentioned at the top of this page, we get the following picture:



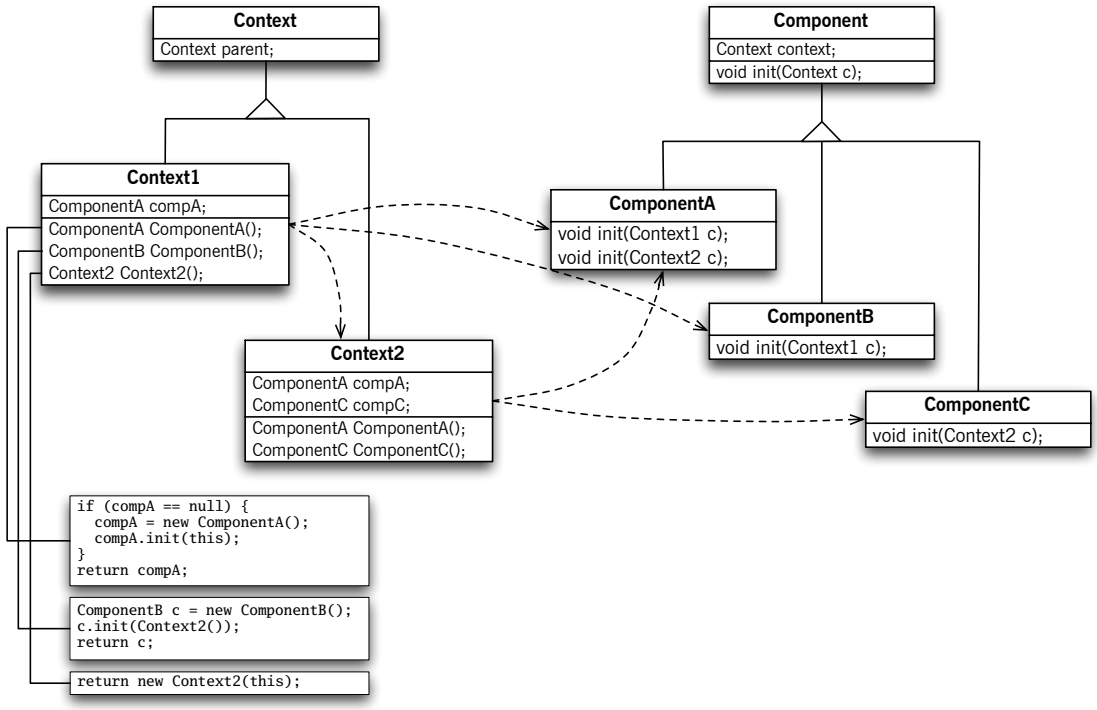


Figure 4.2: Structure of the architectural pattern *Context/Component*.

Contexts are represented by lines. Singleton components embedded in a context correspond to boxes located directly beneath the line. A context's non-singleton components are depicted as lifted boxes with an arrow pointing to them. More complex components refer to subcomponents defined in local contexts, which are drawn as lines directly beneath the component's box.

4.2.1.2 Structure

The structure of the architectural pattern is shown in Figure 4.2. The pattern has four different participants:

Context `Context` is the superclass of all contexts. It simply defines a generic reference to the enclosing context.

Component The abstract superclass of all components defines a method `init` which is called immediately after component creation to initialize the component. The context in which the component is embedded is passed as an argument to `init`. `init` typically gathers references to other components that are accessed within the component. Thus, what we called the configuration and initialization stage in Section 3.4.1, is combined in this pattern in a single, strict initialization step.

Concrete Context A concrete context like `Context1` and `Context2` in Figure 4.2 defines a particular context of a system. It provides factory methods for all

embedded components. These methods specify whether a component is a singleton relative to the context, and whether a component is initialized in an own nested context, defining local subcomponents. Furthermore, a `ConcreteContext` provides factory methods for creating local subcontexts.

Concrete Component A concrete component like `ComponentA` in Figure 4.2 implements a specific, instantiatable component of a system. It defines a customized `init` method which is called from the corresponding context immediately after object creation. The context is passed as an argument, enabling the `init` method to import references to external components which are accessed within the component. It is only possible to import components from the own or an enclosing context.

Overloading the `init` method enables a flexible embedding of components in different concrete context classes. The `init` methods act as adaptors to the different contexts in which a component can be embedded.

An important design decision in the pattern above is the separation of component creation and component initialization. This separation is important to break cycles in the dependency-graph of the components. Let's look at the scenario of Figure 4.2. By using the symbolic notation we get the following diagram:

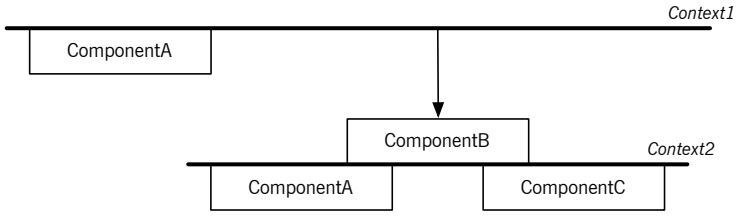


Figure 4.2 also shows the factory method implementations of the concrete context class `Context1`. For singletons like `ComponentA` it is important that the object is first created and then initialized in a second step. Otherwise, the instantiation of mutually dependent components would cause an endless loop in which alternately new components are created infinitely.

4.2.1.3 Consequences

The Context/Component pattern is a composite architectural design pattern. Contexts are combinations of *AbstractFactories* [74] and *ObjectServers*. They support hierarchical organizations of complex systems while offering a uniform and extensible configuration mechanism. Since the components of a system are defined explicitly and centrally within a context class, the context hierarchy can also be seen as a formal specification of a system architecture.

The Context/Component pattern decouples the system composition from the implementation of the components. This approach enables a flexible reuse of

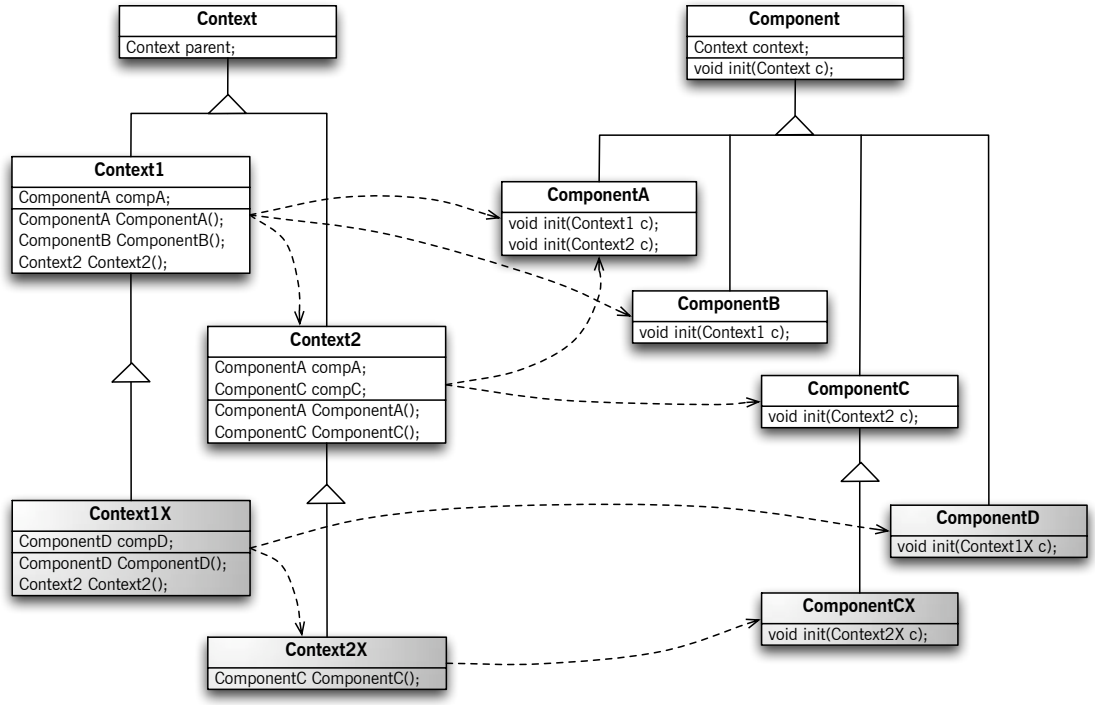
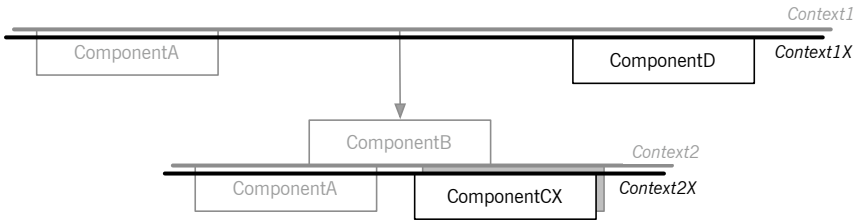


Figure 4.3: Extending a system by subclassing.

components in different, unrelated contexts. Only an adaptor in form of a new `init` method is necessary to embed a component in a new context. This principle is illustrated in Figure 4.2 where, for instance, component `ComponentA` can be deployed in two completely unrelated contexts, `Context1` and `Context2`.

With the Context/Component pattern it is possible to exchange existing components and add new components to a system without modifying any source code of existing component or context classes. Extended systems evolve out of existing ones simply by subclassing. Figure 4.3 explains the principle by illustrating how to extend the system of Figure 4.2. More precisely, Figure 4.3 illustrates a scenario in which a new top-level component `ComponentD` is added to the original system. This is done by extending the top-level context `Context1`. Furthermore, component `ComponentC` gets replaced by a new component `ComponentCX` in the local context `Context2` of component `ComponentB`. Figure 4.3 highlights new context and component classes in gray. In the symbolic notation, the scenario looks like this:



In this notation, new contexts and components are drawn in black, whereas old

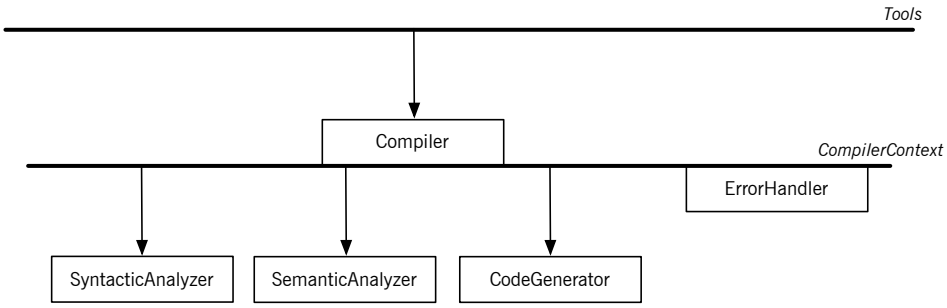


Figure 4.4: A simple compiler architecture.

contexts and components "shine through" in gray. Such contexts and components are reused "as is;" i.e. in their original version. Extended components are displayed on top of the original components (which can still be seen as a shadow) if extended components replace the original ones.

As this example shows, extending or modifying a system does not entail any source code modifications of existing classes. By extending a system, one does not destroy the original version. Both the original and the extended systems can be deployed separately or even together within one application.

4.2.2 Application to Extensible Compilers

We now combine the techniques developed in Section 4.1.3 and 4.2.1 to build extensible compilers. Our software architecture for extensible compilers is based on the classical design of a *multi-pass compiler* [159]. A multi-pass compiler decomposes the compilation process into a number of subsequent phases. Conceptually, each of these phases is transforming the program representation until target code is emitted. Today, most compilers use a central data structure, the *abstract syntax tree*, for the internal program representation. This syntax tree is initially generated by the parser and modified continuously in the following phases. From the software architecture's point of view, this design can be classified as a *Repository* [184]. See Figure 4.1 on page 151 for an illustration.

We now apply the Context/Component design pattern. Figure 4.4 shows the structure of a simple compiler. The compiler is modeled as a component of the top-level `Tools` context. The compiler is a composite component, consisting of several subcomponents that are defined in the local `CompilerContext`. Except for the `ErrorHandler` component, these subcomponents model the different compilation phases. By declaring `ErrorHandler` to be a singleton component with respect to its context, we ensure that every compiler phase accesses the same `ErrorHandler` object.

The implementation of this structure as an instance of the Context/Component pattern is straightforward. We only show some interesting code fragments, starting with the `Tools` class:

```
class Tools extends Context {
  Compiler Compiler() {
    Compiler c = new Compiler();
    c.init(CompilerContext());
    return c;
  }
  CompilerContext CompilerContext(){ return new CompilerContext(this);}
}
```

The Tools class defines the factory method for the Compiler component and the nested CompilerContext. We follow the naming convention of giving factory methods the name of the method's return type. The CompilerContext class contains the actual configuration of the compiler. It defines the different compiler phases and a global ErrorHandler component.

```
class CompilerContext extends Context {
  CompilerContext(Tools encl) { super(encl); }
  SyntacticAnalyzer SyntacticAnalyzer() {
    SyntacticAnalyzer c = new SyntacticAnalyzer();
    c.init(this);
    return c;
  }
  SemanticAnalyzer SemanticAnalyzer() { ... }
  CodeGenerator CodeGenerator() { ...}
  ErrorHandler err;
  ErrorHandler ErrorHandler() {
    if (err == null) { err = new ErrorHandler(); err.init(this); }
    return err;
  }
}
```

In our compiler, data like abstract syntax trees is represented with extensible algebraic types. Most compiler phase implementations are similar to the one of SemanticAnalyzer shown in the following listing. They define a method operating on the abstract syntax tree for performing the actual operation of the compiler phase. Pattern matching is used to distinguish the different Tree nodes.

```
class SemanticAnalyzer extends Component {
  ErrorHandler ehandler;
  void init(CompilerContext cc) {
    ehandler = cc.ErrorHandler();
  }
  void analyze(Tree tree) {
    switch (tree) {
      case Variable(String name): ...
      ...
    }
  }
}
```

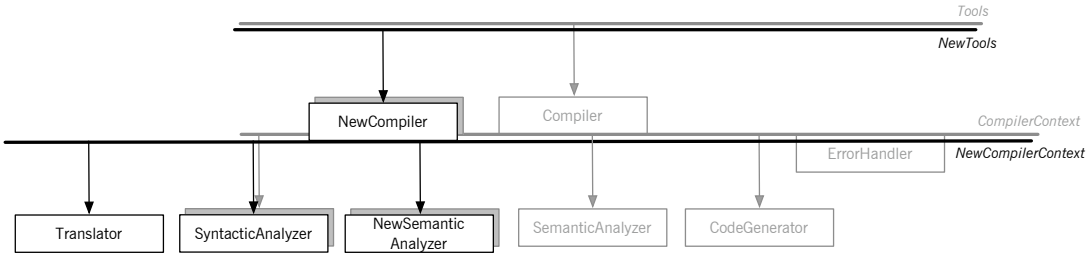


Figure 4.5: An extended compiler architecture.

Finally, we present the implementation of the main compiler component. In method `init`, this component retrieves all the compiler phases from its deployment context. Method `compile` is used to execute the phases sequentially.

```
class Compiler extends Component {
  SyntacticAnalyzer syntactic;
  SemanticAnalyzer semantic;
  CodeGenerator codegen;
  void init(CompilerContext cc) {
    syntactic = cc.SyntacticAnalyzer();
    semantic = cc.SemanticAnalyzer();
    codegen = cc.CodeGenerator();
  }
  void compile(String file) {
    Tree tree = syntactic.parse(file);
    semantic.analyze(tree);
    codegen.generate(tree);
  }
}
```

Figure 4.5 depicts a possible extension of our small exemplary compiler. We assume that new syntactical constructs, like the `Zero` and `Succ` terms on page 155, were added to the source language. Therefore, both a new syntactical and a new semantical analysis are needed. Furthermore, a new compilation phase `Translate` is introduced, which transforms syntax trees of the extended language into trees of the original source language. Since it is still possible to use the original semantical analysis, we can check a program transformed by the `Translate` phase again before applying the code generator adopted from the old compiler. This second semantical analysis might also be imposed by the translator, which might not preserve attributes like typings or scopes, determined by the first semantical analysis. The code generator typically relies on a proper attribution of the structure tree and therefore requires a second semantical analysis after the syntax tree transformation if the transformation does not preserve the full tree attribution. Here is an implementation of the corresponding compiler context hierarchy extension:

```
class NewTools extends Tools {
  NewCompiler NewCompiler() {
    NewCompiler c = new NewCompiler();
    c.init(NewCompilerContext());
    return c;
  }
  NewCompilerContext NewCompilerContext() {
    return new NewCompilerContext(this);
  }
}
```

In the NewTools context, the existing Compiler factory method is not overridden, making it possible to call both, the new and the old compiler from that context. The NewCompilerContext class provides an extended syntactical analysis and includes two new compiler phases, NewSemanticAnalyzer and Translator.

```
class NewCompilerContext extends CompilerContext {
  NewCompilerContext(NewTools encl) { super(encl); }
  SyntacticAnalyzer SyntacticAnalyzer() {
    NewSyntacticAnalyzer c = new NewSyntacticAnalyzer();
    c.init(this);
    return c;
  }
  NewSemanticAnalyzer NewSemanticAnalyzer() {
    NewSemanticAnalyzer c = new NewSemanticAnalyzer();
    c.init(this);
    return c;
  }
  Translator Translator() { ... }
}
```

The NewSemanticAnalyzer phases extends the already existing semantic analyzer. The extended compiler uses both, the former semantic analysis which gets inherited to NewCompilerContext and the new extended phase. A possible implementation of the new semantic analyzer is shown in the following program. The new semantic analyzer class refines the analyze function by overriding the existing analyze method with a method which handles the new syntactical constructs and which delegates all other cases to the former implementation.

```
class NewSemanticAnalyzer extends SemanticAnalyzer {
  void analyze(Tree tree) {
    switch (tree) {
      case Zero: ...
      case Succ(Tree tree): ...
      default: super.analyze(tree);
    }
  }
}
```

With this new semantic analysis and a translator component which is not described in more detail here, we can finally implement the new main component of the compiler:

```
class NewCompiler extends Compiler {
  NewSemanticAnalyzer newsemantic;
  Translator trans;
  void init(NewCompilerContext cc) {
    super.init(cc);
    newsemantic = cc.NewSemanticAnalyzer();
    trans = cc.Translator();
  }
  void compile(String file) {
    Tree tree = syntactic.parse(file);
    newsemantic.analyze(tree);
    tree = trans.translate(tree);
    semantic.analyze(tree);
    codegen.generate(tree);
  }
}
```

This example shows that components and configurations (contexts) can be extended and reused quite flexibly in the given framework. This is due to a strict separation of datatype definitions, component implementations, and the configuration of systems. Extensible algebraic datatypes with defaults provide a mechanism for separating datatype definitions from components encapsulating operations, whereas the Context/Component pattern promotes a separation of components from system configurations.

4.2.3 Architecture of JaCo

JACO is a multi-pass JAVA compiler. It's architecture and implementation is described in detail in [210]. For this case study, we only present the relevant facts so that a comparison with its counterpart, implemented in the programming language KERIS, is possible.

Data representation. A compiler has to deal with a wide range of programming language specific data, which has to be represented in a suitable way. Examples are the abstract syntax of programs, types, symbols, scopes, or compile-time constants. Since JACO is intended to be extensible, all data representations have to be extensible as well.

In JACO, data with different variants is expressed with extensible algebraic datatypes. Functions operating on such data are organized in extensible "modules," one for every algebraic type. Concrete instances of all datatypes are created solely via factories, never directly by instantiating a class.

Separating datatype definitions from function definitions does not only allow programmers to extend both more flexibly, it also allows them to have different implementations of the functionality in different contexts of the compiler without changing the identity of the data. Imagine a representation for types and a subtype operation which checks if a type is a subtype of another given type. Depending on the compiler pass, this subtype operation might have to behave differently. In compilers, or generally in systems where it is impractical to update all objects just for updating their functionality, if would here be necessary to parameterize the subtype operation with the current compiler pass so that this operation could be equipped with context-sensitive behavior.

Phase decomposition. JaCo's compiler phases are organized hierarchically in accordance with the following principle:

1. A compiler run is implemented by a compiler phase.
2. A compiler phase is either *atomic* or *compound*.
3. An *atomic phase* traverses the structure tree, modifies it, or produces other side-effects.
4. A *compound phase* consists of a sequence of other compiler phases.

A typical multi-pass compiler simply consists of a sequence of atomic phases. Compound phases on the other hand introduce abstraction layers that help to understand a system more easily in a top-down manner. Furthermore, compound phases easily allow to reuse subsystems consisting of several phases more safely. For instance, if a semantic analysis phase is needed more than once, it can simply be instantiated multiple times. Without compound phases, we would have to instantiate all atomic phases that belong to the semantic analysis and organize them in the same sequential order to avoid inconsistencies. This is a tedious and error-prone procedure which also duplicates maintenance. Changes in the semantic analysis have to be carried out in several places consistently. Similarly, compound phases make it more easy to modify subsystems of the compiler locally in a safe way.

Figure 4.6 presents a *structural decomposition* of JaCo's compilation process into a hierarchy of compiler phases. This figure also indicates in what order the different phases are executed at runtime. As Figure 4.7 shows, it is straightforward to implement such a hierarchy of compiler phases with the Context/Component pattern. Here, every compound phase has a local component context which defines all sub-phases. For historic reasons, compound phases are not implemented as singletons. The original intention was to allow a phase to be used multiple times in the same context based on a runtime decision; but none of JaCo's compiler extensions ever made use of this feature.

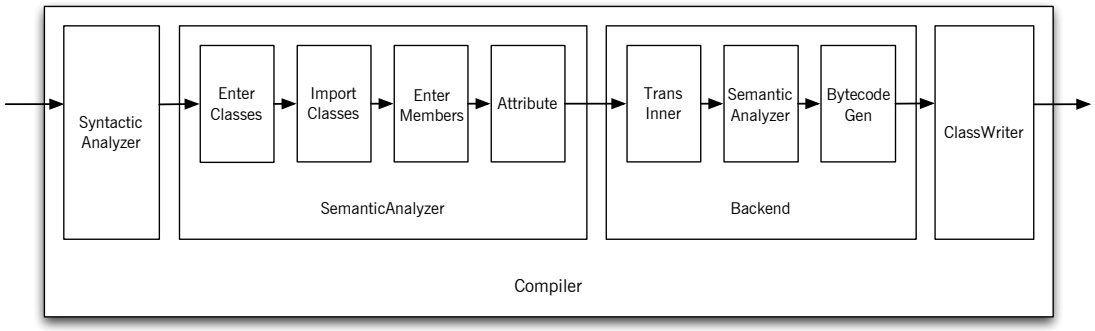


Figure 4.6: JACO's hierarchy of compiler phases.

Services. As shown in Figure 4.1 on page 151, compilers also include various components that do not implement a compiler phase directly, but rather provide services used across compiler phases. JACO provides such components for the following tasks:

1. Data input and output (e.g. class loading, pretty printing, user interactions like error output, etc.),
2. Administration of global data structures (e.g. name table management, scope management, classfile management, etc.),
3. Function libraries for a specific datatype (e.g. operations on types, constants, or symbols),
4. Descriptions of source and target language aspects (e.g. specification of predefined operators, modifiers, types, bytecodes, etc.).

This is not a mutually exclusive classification of JACO's components. Sometimes it is not possible to assign a component to exactly one category. For instance, the function library for symbols is combined with the administration of the global symbol table. Retrospectively, this was a major design flaw, since it sometimes restricts reuse significantly. It is, for example, not possible to have a compiler with two different function libraries for symbols (where one extends the other one), since this would yield two global symbol tables. This problem also prevented a clean implementation of the KERIS compiler as an extension of JACO. Since this compiler translates KERIS programs (typed with dependent types) to plain Java (typed with purely nominal types), it is important to have, at least, two different versions of the type operations: one corresponding to JAVA's typing rules, and an extended one respecting KERIS dependent type system. Since type operations and predefined global types were jointly defined within the same component, it was not possible to create two instances of the type operations, since those would inconsistently refer to the two sets of predefined global types.

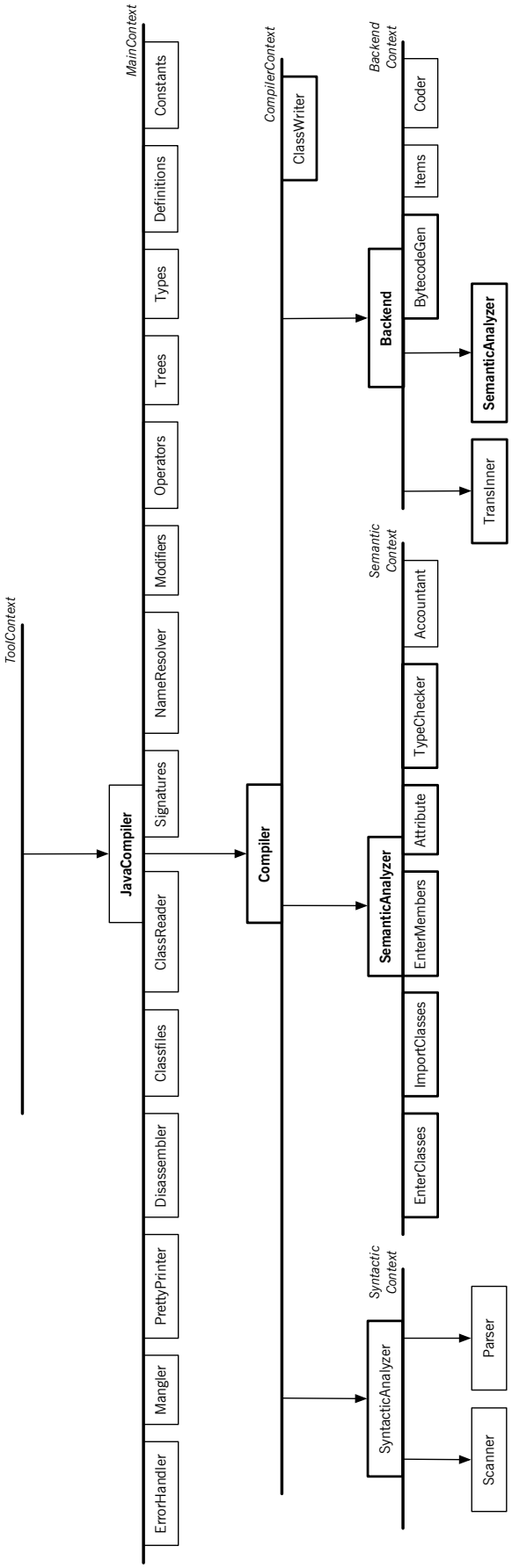


Figure 4.7: Decomposition of JaCo into components and contexts.

Configuration. A decomposition of JACO into all the various components can be found in Figure 4.7. In this figure, the architecture of JACO is shown as an instantiation of the Context/Component pattern. Compiler phases are emphasized with boxes in boldface. The most interesting aspect in the configuration of JACO is the reuse of the semantic analysis phase. It is used in the `CompilerContext` as a top-level phase and within the `BackendContext` as a local phase for attributing the syntax tree produced by the `TransInner` tree translation phase which eliminates inner classes.

Except for the parser, all components are written by hand. The parser is generated by JCUP from an LALR(1) grammar. JCUP is a YACC-style parser generator derived from JAVACUP [91]. A brief description of all compiler components can be found in [210].

4.2.4 Extending JaCo

Overview. In general, JACO is extended by performing at least some of the following four steps:

1. *Extending data structures:*
   New data variants are included by extending algebraic types, existing data variants are updated by subclassing. To integrate new or extended datatypes into the compiler, the corresponding factory classes have to be extended.

2. *Updating existing components:*
   Existing components are subclassed to modify existing functionality by overriding, and to provide new functionality in form of new fields and methods.

3. *Creating new components*

4. *Configuring the extended compiler:*
   An extended context hierarchy has to be built on top of the existing one which integrates new and modified components and sub-contexts.

The general idea is to create a new compiler by only implementing the differences in form of subclasses and by reusing all other compiler components and configurations "as is." Implementing an extension in this fashion (i.e. purely based on inheritance) guarantees that the old compiler is not affected by the changes and therefore can be used as before. Furthermore, due to late binding, the new and the old compiler share most binary components. Physically, the new compiler consists only of classfiles that describe the differences compared to the old version. This form of program development is often called *programming by difference* [173, 94].

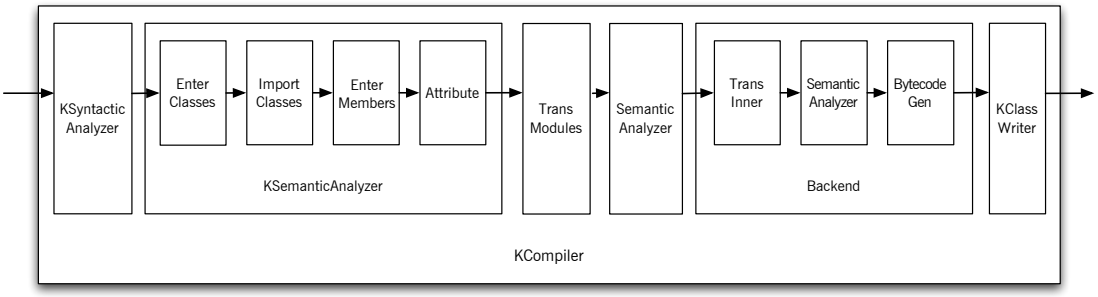


Figure 4.8: KerisC's hierarchy of compiler phases

A Keris compiler based on JaCo. As an example for a typical JaCo extension, we now look at KerisC, our first compiler prototype for Keris. KerisC works like most compilers for programming languages that extend Java: It implements a customized parser (generated from a modified LALR grammar) which generates extended abstract syntax trees. These trees are analyzed by a customized semantical analysis phase. This phase is followed by a translation phase which transforms extended abstract syntax trees to regular Java syntax trees. Some compilers also transform tree attributes so that after the transformation the tree is still fully attributed (with type and symbol information). For simplicity, extensions of JaCo typically do not do this. Instead, the syntax tree is type checked again after the tree transformation phase, restoring all tree attributes. After this second semantical analysis, the standard Java backend generates bytecode for all classes. In the Java backend, only the component which writes out classfiles has to be modified so that the original types and scopes (which are probably lost after the transformation phase) are saved in a special attribute inside of the classfiles.

An overview over the phases of KerisC is given by Figure 4.8. Another view on the architecture of the system gives the Context/Component diagram in Figure 4.9. This diagram shows that for implementing KerisC, almost no new compiler components have to be developed from scratch. Mostly old components have to be adapted and new extended contexts have to be created that instantiate new components instead of old ones.

Most remarkable in Figure 4.9 is the way in which the original semantic analysis is reused. It appears three times in the compiler; two times, the original semantic analysis for Java is reused "as is," one time, an extension of the original analyzer is deployed which implements the type checking procedure for Keris. The compiler prototype discussed in this section, performs type checking in the framework set up by the original Java type checker; i.e. the semantic analysis consists of four consecutive phases: `EnterClasses`, `ImportClasses`, `EnterMembers`, and `Attribute`. Without using advanced analysis techniques like lazy type checking, it is not possible to check Keris programs properly in these four phases. This is why KerisC only compiles a subset of Keris. This subset sufficed to bootstrap a second compiler which, this time, implements the full language. The second compiler uses many more type-checking phases (see Section 4.3 for details).

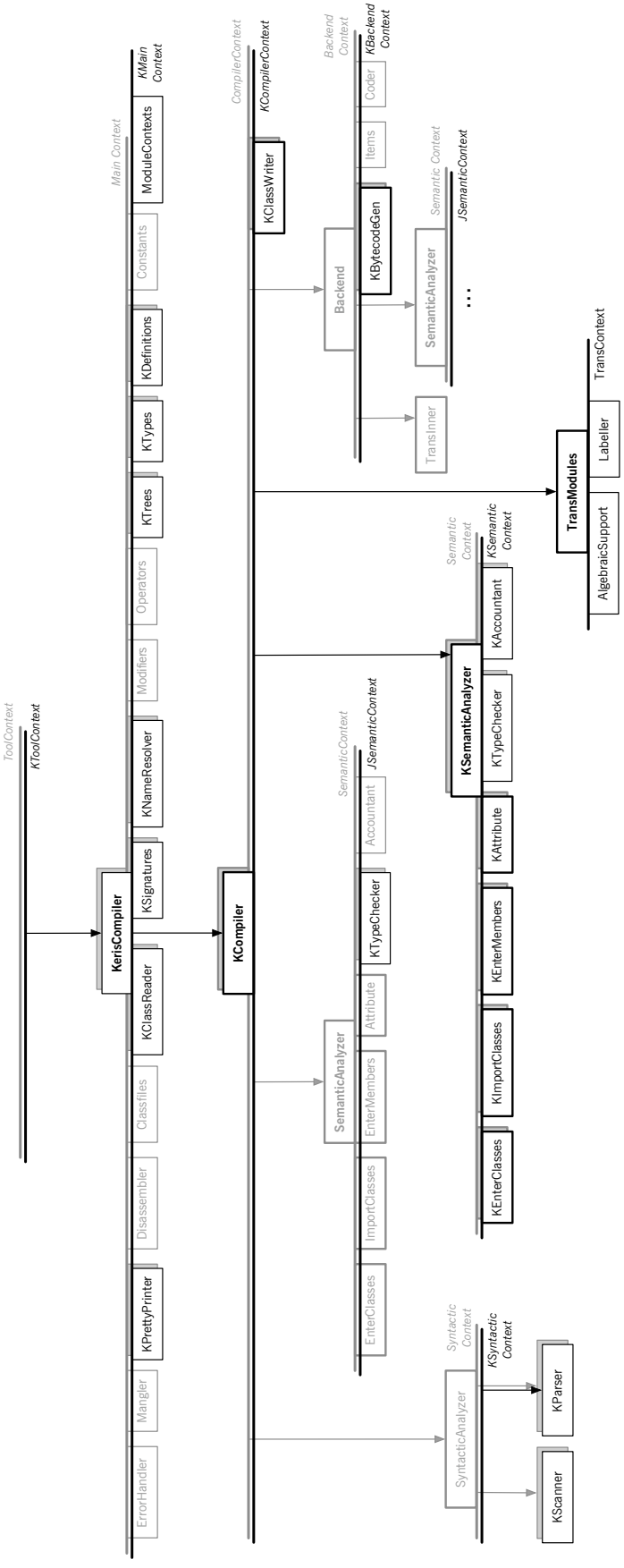


Figure 4.9: Decomposition of KERISC into components and contexts. Contexts and components reused from JACO "shine through" in gray.

4.2.5 Experience

Applications. Throughout the last four years, JaCo was successfully used in various projects. Apart from the initial extension with algebraic types [210, 214, 215], several other language extensions have been implemented and are still being maintained by different people [48, 62, 177, 216, 98, 161]. Among the implementations is a compiler for Java with *synchronous active objects*, proposed by Petitpierre and implemented by Petitpierre and Cavin [161]. Itzstein uses JaCo to implement a compiler for Java with *join* synchronization [98]. Another extension introduces Büchi and Weck's compound types together with type aliases [38]. Operator overloading was added to the Java programming language by Saidji [177], in the style advocated by Gosling [81]. Furthermore, Eugster, Guerraoui, and Damm implemented a domain specific language extension supporting *publish/subscribe* primitives on top of JaCo [62]. A rather exotic extension of JaCo is an implementation of a small language based on *join calculus* [73]. It replaces the syntactic analyzer pass with a full compiler for join calculus that generates a Java syntax tree as output. This tree is then fed into the remaining Java compiler to generate Java bytecodes. In this extension, JaCo is used as a backend for a compiler of a language, which has nothing in common with Java.

Benefits. JaCo turned out to be a valuable tool for rapidly implementing prototype compilers for language extensions. For the implementation of the extensions mentioned before, it was not necessary to modify the code of the base compiler a single time. JaCo's architecture and implementation was open enough to support a broad range of extensions involving new or modified linguistic constructs, type system extensions, and backend modifications. Changes of the base compiler were all related to modifications in the specification of the Java programming language or to bugs found in the compiler implementation. These changes can usually be elaborated in such a way that binary compatibility of Java classfiles is not broken. As a consequence, all compilers which are derived from the base compiler benefit immediately from the changes, since they inherit them. Because of Java's dynamic loading and late binding mechanisms, it is not even necessary to recompile derived compilers.

The implementation of KerisC on the other hand revealed that the semantic analysis of JaCo is difficult to extend and reuse if the type system of the extended language differs significantly from the relatively simple type system of Java. As Section 4.3.2 and 4.3.3 will discuss, these restrictions are mainly due to architectural "mistakes" in the type-checking component. This experience also teaches that developing extensible components requires good knowledge of the domain and a good understanding of future evolution scenarios. A well designed framework can facilitate the development of extensible components, but alone, it can never guarantee that concrete component implementations are extensible by default.

Shortcomings. As the experience with JACO shows, the Context/Component design pattern provides a relatively simple implementation technique that allows one to built hierarchically structured systems that can be extended flexibly. In practice though, many programmers found it quite time consuming to set up an initial context hierarchy layer for extended compilers by hand. Therefore, the build script of JACO provides an option for automating the generation of new empty compiler extensions. This meta-programming tool helped to speed up the process of getting a first compiler extension up and running significantly.

While the separation of components from configurations offers, in principle, advanced reuse capabilities and allows one to reason about the composition of a system independently from the concrete implementation, some programmers had difficulties to use a compound component together with the corresponding context consistently in a large system. The two artifacts are only weakly coupled through the `init` method, and a consistent use of the two is not enforced by the programming protocol (this might possibly not be desired for some cases).

Another minor nuisance is the problem of *change propagation*. If one wants to replace a component in a "deeply nested" context, all enclosing contexts have to be extended as well to finally incorporate the new context which defines the new component. This is a general problem of hierarchical systems which are expressed as nested layers and which are extended by subclassing.

Both extensible algebraic types and object types defined by regular classes can be specialized. Nevertheless, it is not possible to specialize components and types consistently in a type-safe way. Methods that operate on a specific type T can be specialized by overriding, but it is not possible to narrow parameter type T in clients to a more specialized type, since JAVA's type system requires methods to override other methods *invariantly*. JAVA imposes this restriction since a covariant change of a parameter type would be unsound in general [46]. Therefore, all extensions of JACO use type casts to circumvent the restrictions. The downside of this approach is that the compiler cannot check anymore if specialized components are only used in conjunction with objects of specialized types. This is checked dynamically in all places where components make use of specialized functionality. Overall, this is a serious shortcoming — but all software developed in JAVA-like languages is subject to it.[1] Therefore, work on the programming language KERIS had a strong focus on mechanisms that enforce that component abstractions and types evolve consistently and that abstractions can be modified in a covariant manner.

[1]The designer of the programming language EIFFEL [133] considered the covariant refinement of method parameters to be of such an importance for the development of extensible software, that he integrated this feature into the language, even though this mechanism was known to be unsound. This pragmatic design decision is unfortunate since it hides potential problems from programmers. In JAVA-like languages with invariant method overriding, explicit type casts make the programmer aware of possible inconsistencies that may arise when a method parameter has to be narrowed to a specialized type. Languages like SCALA [151], GBETA [58], FAMILYJ [208], or KERIS show how to solve the problem in a statically type-safe way via dependent object types.

4.3 JaCo2: Extensibility with Extensible Modules

JaCo2 is a re-implementation of the extensible Java compiler JaCo in Keris. Where JaCo's implementation makes extensive use of design patterns and uses ad-hoc workarounds to refine components covariantly, JaCo2's implementation exploits the linguistic constructs of Keris. We will show how these constructs help to safely develop a robust, extensible application.

4.3.1 Architecture of JaCo2

System composition. The architecture of JaCo2 closely corresponds to the one of JaCo. Basically every component of JaCo is mapped to a Keris module. Compound components are mapped to modules that aggregate submodules. An overview over the architecture of JaCo2 is given in Figure 4.10. As introduced in Section 3.2, boxes represent modules and nesting is used to express submodule aggregation.

Where the experience showed flaws in the design of JaCo, this was corrected in JaCo2. Therefore, there are some slight variations in the structure of the system compared to the original JaCo implementation. For example, module TYPEOP was derived from the JaCo component Types by factoring out compiler-phase-dependent type operations. This separates the definition of type representations and type operations and enables the use of different operations in different contexts with the same type representation. For instance type operations differ in different versions of the semantic analyzer. In JaCo this issue was addressed with a hack. A compiler-global variable in the Types module was used to switch between different versions of the type operations.

Another variation in the design of JaCo2 is caused by the conversion of former singleton objects into modules. For instance, all the submodules of the global MAIN module and the submodules of the CLASSREADER and CLASSWRITER modules had been singletons originally (and no components in the sense of the Context/Component design pattern).

Module reuse. Figure 4.10 does not fully reveal how modules are reused already in the base compiler. Modules that are being reused "as is" in different contexts appear multiple times with the same name in the figure. Examples are the modules SEMANTIC_ANALYZER, TYPEOP, and BASICRESOLVER. BASICRESOLVER is the parent module of RESOLVER. Such refinements and specializations of base compiler modules are also not visible in Figure 4.10. For instance CLASSPATH is a specialization of the PATH module, which is used to represent source paths in the compiler. Furthermore, all modules representing compiler phases are specializations of a generic PROCESSOR module which itself refines module DEBUGGABLE. Module DEBUGGABLE implements support for debugging a compiler component in a generic way.

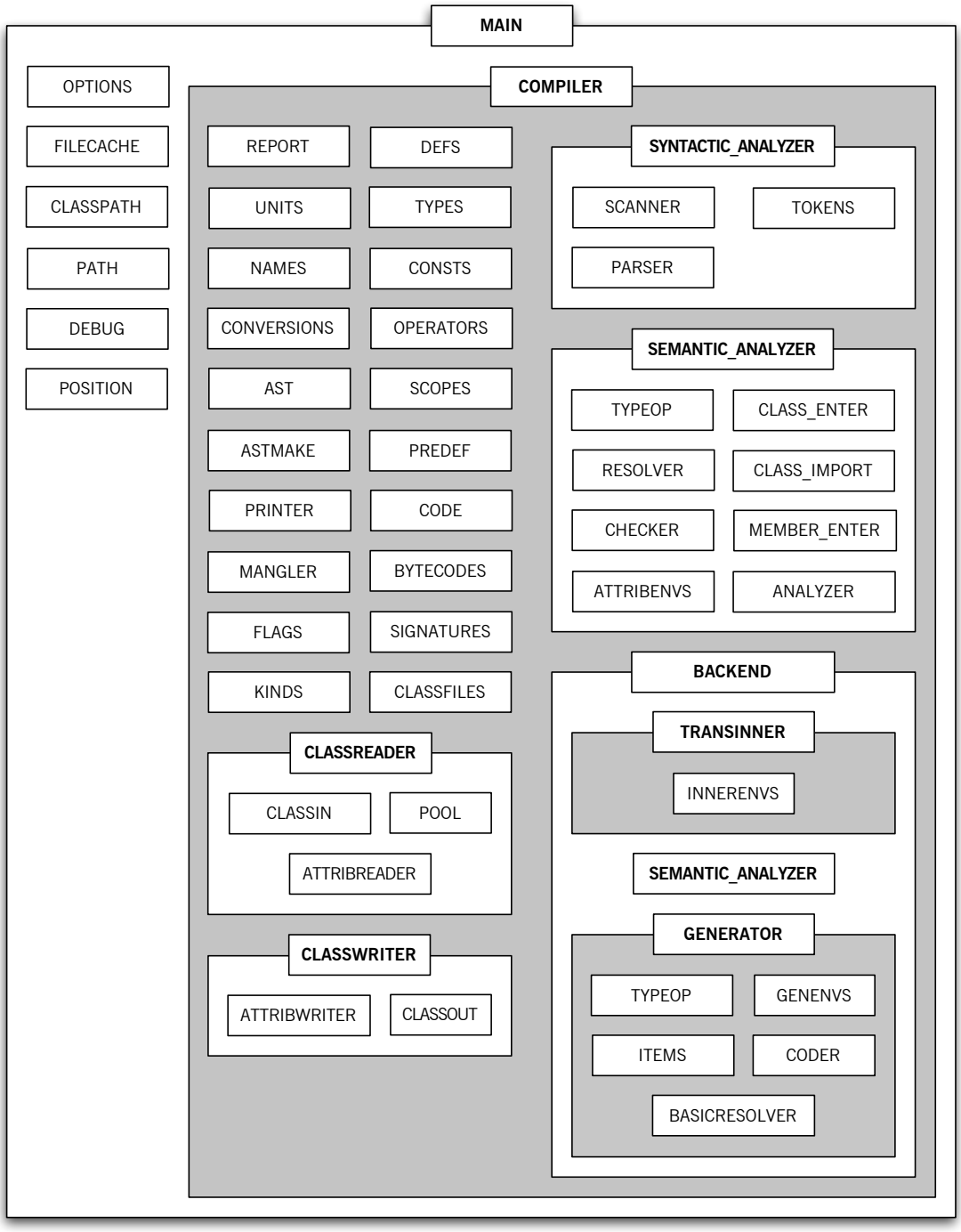


Figure 4.10: Hierarchical composition of JaCo2.

Covariant module specialization. As an example for the simultaneous refinement of cooperating modules in combination with the refinement of class fields, we now focus on the implementation of module TRANSLATOR. This module plays the role of a generic template for syntax tree translation phases in the compiler. In it's original form, it just implements the identity translation. Here is a code fragment of the module:

```
abstract module TRANSLATOR specializes PROCESSOR requires ... {
  module ENVS;
  ...
  Tree transDecl(Tree tree, Env env) { ... }
  Tree transStat(Tree tree, Env env) { ... }
  Tree transExpr(Tree tree, Env env) { ... }
  Tree transType(Tree tree, Env env) { ... }
  ...
}
```

For the various syntactical categories, module TRANSLATOR provides a translation method which, in its original form, just returns a copy of the syntax tree. Specializations of this module override these methods and handle some tree nodes specially. Since the translation of trees is, in general, context dependent, the tree translation methods also receive an environment which encapsulates information about the context of a tree node. These environments are represented with the help of submodule ENVS which defines the class field Env.

```
module ENVS requires ... {
  class Env implements IEnv = CEnv;
  interface IEnv {
    IEnv(Tree tree, Env next);
    Env next();
    Tree tree();
    MethodDecl enclMethod();
    ClassDecl enclClass();
    CompilationUnit topLevel();
  }
  class CEnv implements IEnv {
    ...
  }
}
```

In specializations of TRANSLATOR it is possible to override submodule ENVS with a specialized version in order to collect other, translation specific context information. The specialized translator module can safely access the new context data without the use of type casts. In the base compiler, only module TRANSINNER specializes TRANSLATOR. It provides a refined submodule INNERENVS which is used to collect free variables. This information is used for "lambda lifting" [162] anonymous inner classes — one of the main tasks performed by module TRANSINNER.

4.3.2 Extending JaCo2

Module refinements and specializations are the primary extensibility mechanisms in KERIS. These mechanisms are already exploited in the base version of the extensible JAVA compiler JACO2. Extensions of JACO2 are most easily derived in a top-down manner. Whenever a second implementation of an already existing module is needed, the existing module is specialized, otherwise modules evolve through refinements.

Figure 4.11 illustrates how the implementation of JACO2 was used to develop KECO, an (extensible) compiler for KERIS. New modules are displayed as black boxes; module refinements that override a previous version overlay the former module in the diagram. These overridden submodules are still partly visible as a shadow of the overriding submodules. Modules that are inherited and reused "as is" from the base compiler are drawn in gray as if they would "shine through" from the base implementation.

In the top-level module K_MAIN, only submodule K_OPTIONS was overridden. The full JAVA compiler is inherited "as is" to the KERIS compiler, so that it can be used if the command-line option "–java" is set. The actual KERIS compiler module K_COMPILER is a specialization of the JAVA compiler module, and as such, can coexist with it in the same context. The K_COMPILER module specifies many customizations:

- Source language description modules like K_AST, K_FLAGS, K_TYPES, and K_DEFS are updated to cover KERIS-specific information.
- Target code relevant description modules like K_SIGNATURES, K_CLASSFILES, K_CLASSREADER, and K_CLASSWRITER are customized to preserve KERIS-specific meta-data and enable separate compilation.
- New data structures for dealing with type and module collections got introduced by the new modules TYPECOLL and MODCOLL.
- All top-level compiler phases of the base compiler are overridden with versions that enable the translation and analysis of KERIS programs.

The semantical analysis underwent a major restructuring which was mainly necessary because the type system of KERIS has a quite different nature compared to the relatively simple type system of JAVA. It now consists of 10 subsequent subphases from which only one could be reused as it was. Besides the 3 subphases that were inherited and refined, 6 new phases had to be introduced. An overview over the evaluation order of the various semantical analysis phases is given by Figure 4.12.

Please note that for the implementation of KECO it was necessary to develop two different extensions of JACO2's semantical analysis. Besides K_SEMANTIC_ANALYSIS there is also a module J_SEMANTIC_ANALYSIS which extends JACO2's version minimally by supporting covariant return types when

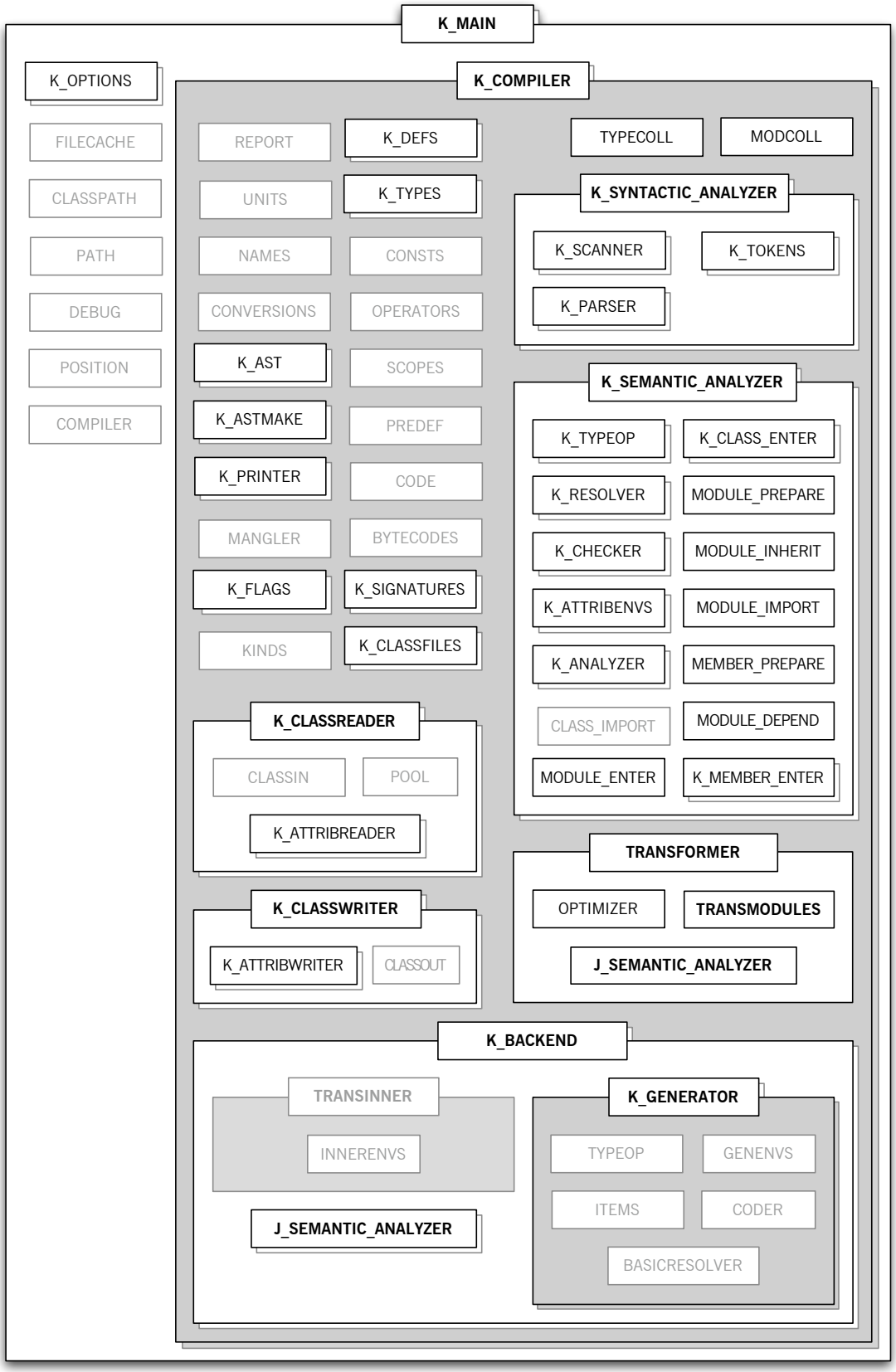


Figure 4.11: Hierarchical composition of KeCo, an extension of JaCo2.

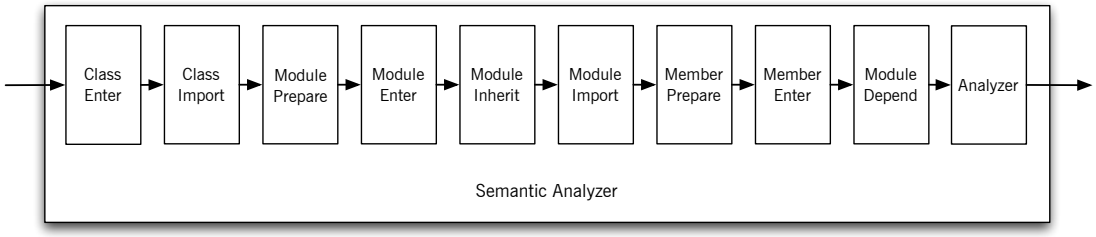


Figure 4.12: Compiler subphases of KeCo's semantical analysis.

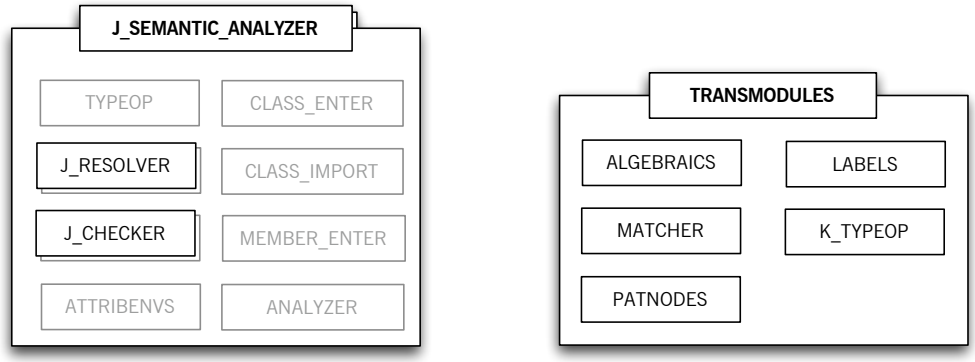


Figure 4.13: Configuration of J_SEMANTIC_ANALYZER and TRANSMODULES.

methods are overridden. Figure 4.13 shows that for implementing this extension, only the submodules RESOLVER and CHECKER have to be overridden in the original semantical analysis. The modified semantical analysis is used in both the TRANSFORMER module and the K_BACKEND module. The transformation module translates, with the help of submodule TRANSMODULES, KERIS code into pure JAVA code, still keeping covariant return types. The backend inserts bridge methods to bridge the gap between KERIS' covariant and JAVA's invariant method overriding [32]. Only for this reason it was necessary to extend the backend in KECO as well. The structure of the backend module is displayed in Figure 4.11, the structure of TRANSMODULES can be found in Figure 4.13.

4.3.3 Experience

It was straightforward to develop an implementation of JACO in KERIS based on the original compiler design. The re-implementation of the compiler in KERIS even revealed several design-flaws and inconsistencies in the original architecture which were corrected in the new compiler.

Revealed design flaws. Some design flaws were discovered through KERIS' strict mechanism for defining context dependencies explicitly opposed to the Context/Component approach where components mention the deployment context explicitly, but do not precisely state the accessed components. For instance

the problem with the former Types module, described in Section 4.3.1, was discovered this way. Here, some modules would not compile without referencing the types module, even though they did not access any functionality of this module directly. The sole reason for this access was to toggle a flag which controlled the concrete implementation of the type operations which were executed indirectly through other modules. This approach was necessary to allow different contexts of the compiler to refer to different type operations but to share the same type representation.

The problem was solved by breaking up the original Types component into two modules TYPES and TYPEOP, one defining the data structure for representing types, the other one defining operations on types. With this approach it was possible to have a global type representation, and allow at the same time to instantiate different versions of the TYPEOP module in different contexts. This change triggered further minor corrections in the structure of the compiler which all had to do with a clean separation between functionality shared by all compiler phases and context-sensitive functionality.

The change also revealed a quite serious inconsistency in JaCo. JaCo had global typechecking and name resolution components (where the latter depended on the typechecking component). While the typechecking component was only used by the semantic analysis, name resolution functionality was also needed by the backend — this was the initial motivation for integrating both components on a global level. Unfortunately, even though the two compiler phases can share implementations for both components in the base compiler, possible compiler extensions might require changes for semantic analysis purposes only. But in the original design, this would inevitably also affect the backend. Therefore, JaCo2 abolishes global typechecking and name resolution components and introduces two local and independent modules CHECKER and BASICRESOLVER instead. Since some advanced resolution functionality depends on functionality provided by CHECKER, a module RESOLVER was created that refines BASICRESOLVER just to introduce calls to the typechecking module. With this infrastructure it was easy to introduce name resolution and typechecking functionality in form of the two modules RESOLVER and CHECKER in the semantical analysis and to use a local BASICRESOLVER module (without the CHECKER module) in the backend.

Design improvements. Similar refactorings were carried out in other parts of JaCo2. All were made possible by the better reuse and integration capabilities of Keris' "plug-and-play" modules compared to the object-oriented components of JaCo that have to be glued together with quite heavy boilerplate code. Since this boilerplate code has to be written by hand, it is natural that developers try to minimize it as good as possible by avoiding new components from being integrated. In JaCo and extensions thereof this often lead to an exaggerated reuse of component instances which all had to be shifted artificially to a more global level in the component hierarchy just for making them more widely accessible.

The design philosophy of JAKO2 puts more emphasis on the reuse of *modules* instead of *module instances* and pushes modules into more deeply nested positions. This requires that often several instances of the same module are created in different contexts, but it eases extensibility and makes the system more robust with respect to extensions.

The implementation of JAKO2 follows the *Law of Demeter* [115, 117, 114] which requires that a module is only allowed to access its immediately required or aggregated modules, and not their submodules. Thus, JAKO2 and KECO do not make use of the `::` operator; all submodules are private and hidden from clients. In the context of extensible software, this is an important property, because then, programmers who extend a module know precisely the scope of the change and are therefore also aware of the impact. For instance, replacing a submodule with a different version would only have a direct consequence for the host module as well as all other submodules (which are all known at compile-time), but not directly to unknown clients.

Extensions. In general it turned out that JAKO2 can easily be extended through module refinements and specializations without writing glue or adaptation code. The module re-wiring mechanism of KERIS was very helpful, in particular when small changes had to be integrated into big subsystems. The implementation of KECO revealed basically two problems:

- In compilers, many components are recursively dependent, often not even directly but indirectly through other dependent modules. This makes it generally hard to specialize a particular module, because it would require that all modules that participate in this recursion need to be specialized as well.
- When changes have to be integrated into a deeply nested submodule, this change will be propagated to all enclosing modules. These modules also have to be refined or specialized to integrate the submodule that refers to the change.

The first problem is application specific and there is little that can be done to ease module specializations in general. A sound type system has to be conservative in the sense that it always assumes the worst case. For dependent modules this means that they mutually exchange objects of locally defined classes that refer to all modules in the dependency closure through inheritance. It would be possible to refine the type system of KERIS and annotate dependencies with privileges. This would avoid for many cases that all recursively-dependent modules have to be specialized simultaneously. On the other hand, it would complicate the definition and deployment of modules significantly and would lead to software that is much more difficult to extend. This is because one advantage of conservative type systems and their worst case assumption is that even if such an assumption does not hold for the original system, there might be a possible

extension where exactly this assumption is exploited. It is therefore a tradeoff between more expressive dependency annotations that ease specializations and more general dependency specifications that capture a wider range of possible extensions.

The second problem is related to the way systems are extended in Keris. Keris allows programmers only to change direct members of modules. This restriction enforces module encapsulation and follows the policy that foreigners are not allowed to change abstractions from the outside. While aspect-oriented programming gives up this restriction in favor of more flexible and expressive modularization mechanisms, it is essential for object-oriented programming. A programming language that allows programmers to change abstractions in deeply nested submodules would give up some encapsulation capabilities, but it would not suffer from the *change propagation* issue. It is therefore worth to consider some syntactic sugar for Keris that would allow programmers to change modules from the outside in a concise but non-invasive manner.

Despite these two issues, the implementation of KeCo showed that the extensibility mechanism of Keris scales extraordinary well. Both module refinements and specializations were used extensively and the separation between the two concepts was quite beneficial for the program understanding, but also for the sound evolution of the system.

4.4 Comparison

4.4.1 Design Patterns vs. Language Support

In the following we will analyze and compare the impact of the design pattern-based approach to extensibility with the second approach that uses the programming language KERIS as an implementation language for extensible systems.

Implementation overhead. The rigorous usage of the Context/Component pattern imposes extra implementation work on the programmer. The biggest part of this implementation overhead consists in the definition of factory methods in context classes to keep configurations of systems extensible, and in component classes to allow classes to evolve by making constructor calls overridable. The aim of such artificially introduced indirections is to eliminate hard links and to keep systems as open as possible. Unfortunately, JAVA-like languages make it very easy to introduce hard links in many ways and even when a special design pattern, like the Context/Component pattern, is used, there is no guarantee that a programmer does not accidentally forget to introduce or use such indirections. The extra implementation overhead may also tempt programmers to resort to ad-hoc mechanisms that circumvent the programming protocol for convenience but at the expense of extensibility.

Programs written in KERIS are not automatically extensible — it is still necessary to explicitly *design* a system for changes. However, by providing suitable linguistic constructs, KERIS facilitates the *implementation* of systems that can be evolved without much effort and without touching existing code or designs. KERIS promotes a modular decomposition of a system into smaller units while retaining extensibility on all levels of decomposition. Since KERIS' module and class abstractions are "open" by default (and can only be "closed" by explicitly stating this), extensibility comes without much implementation overhead and does not require programmers to observe non-trivial and verbose programming protocols. For instance, KERIS programs do not rely on explicit factory methods. Therefore, programmers cannot forget using factory methods and they cannot forget overriding such methods for integrating extensions.

Dependencies. KERIS makes it easy to explicitly specify context dependencies for modules. Modules can be deployed in *every* context that meet these requirements and the type system of KERIS guarantees that they are used only in those suitable contexts. In the architecture of JACO2 we can see that several components exploit this freedom. For instance, modules `SEMANTIC_ANALYZER`, `TYPEOP`, and `BASICRESOLVER` are used in more than one context. Similarly, modules derived from `TRANSLATOR`, like `TRANSMODULES` or `TRANSINNER` are deployed in quite different contexts.

The Context/Component pattern associates with every component a specific context. A component can only be deployed in this context or extensions thereof. For deploying it in other "compatible" contexts, it is necessary to explicitly adapt the component by providing an alternative `init` method. In JACO this problem shows up for the semantical analysis phase. This phase is deployed in two unrelated contexts: in the top-level compiler context and within the backend. We cannot write a generic `init` method that works for both contexts mostly due to JAVA's strict nominal typing discipline (which requires that we give the parameter of method `init` a single nominal type). Compound types [38] introduce some restricted form of structural typing. With such types we could easily specify a generic `init` method in a fashion similar to the way KERIS is handling module contexts. In JAVA-like languages without compound types it is extremely difficult to encode generic context dependencies safely in the style required by the `init` methods.

Consistency. For evolving large systems it is often necessary to extend several components consistently at the same time. Such a mutual extension of several components can be expressed easily in KERIS with a mutual refinement or specialization of context dependencies. This enforces that all extended modules are deployed at the same time and a mixture of old and new versions is impossible.

In the framework of the Context/Component pattern, `init` methods have to be overridden in subclasses to accommodate changes in the context dependencies. Due to JAVA's requirement of overriding methods invariantly, it is not possible to refine the parameter type to the new context type which describes the refined collaboration of possibly mutual dependent components. Thus, a consistent refinement of mutual dependent components is possible but inherently unsafe.

One concern of the design pattern approach was to separate component implementations from their configuration. This improves reusability and with the context hierarchy the programmer develops a separate formalism specifying the architecture of a system. In KERIS, component implementations and local component configurations are defined together in a single module implementation. For documenting a system configuration (without implementation code), one has to explicitly use module interfaces. In contrast to the solution based on design patterns, this approach does not provoke inconsistencies between the configuration of a component and the concrete component implementation. In particular, the conformance of a module implementation with its specification, given in form of a module interface, is checked mechanically.

Type safety. In the implementation of JACO, datatypes and components are defined separately, and only factory methods associate the two. Due to the lack of covariant abstraction facilities in JAVA, it is not possible to express that an ex-

tended component refers to an extended datatype. Consequently, extensions of components have to frequently down-cast values to the extended type; again this is unsafe and error-prone.

In KERIS, datatypes are defined within modules and whenever a module is extended, datatypes can be extended covariantly as well. This tight coupling between modules and datatypes rules out inconsistencies and allows more advanced static type checks than the design pattern-based approach.

4.4.2 Benchmarks

This section will discuss benchmarks for both JACO and JACO2. While Section 3.5 presented benchmarks concerning the runtime performance of the two systems, we are here mainly focusing on the program code. This will round up the presentation of the two extensible compilers, giving quantitative data about the implementation effort. Since it is difficult to compare the code of two systems written in different languages directly, we mainly focus on an independent evaluation of the two systems by looking at the base versions of the compilers and extensions thereof.

4.4.2.1 JaCo

Figure 4.14 presents the results of a code analysis of JACO and its extensions PICO, SJAVAC, CJAVAC, and KERISC. PICO [214, 210] adds extensible algebraic datatypes to JAVA, SJAVAC [161] implements linguistic support for synchronous active objects, CJAVAC [216] implements Büchi and Weck's proposal for compound types and type aliases [38], and KERISC is a preliminary version of the KERIS compiler which was used to bootstrap the full-blown KERIS compiler KECO [213].

The first part of the statistics in Figure 4.14 shows the number of source files, the total number of lines of code (including blank lines), the total number of lexical tokens, the total size of the source code in bytes, as well as the number of class and method definitions.

With approximately 32000 lines of code, JACO's source code size is comparable to the one of non-extensible compilers like the PIZZA compiler or JAVAC. Depending on the complexity of the added language feature, the size of the various extensions ranges from 28% to 75% of the base compiler's source code size.[2] The size of the generated code (i.e. classfiles) is in a similar range: extensions increase the code size of the base compiler between 25% and 62%.

These numbers give an idea about the total code size of typical compiler extensions of JACO. But they do not exhibit the administrative overhead which is

[2]The percentage refers to the number of lexical tokens, since the number of source code lines does not constitute a reliable measurement for the source code size; in particular, if the code is written by different people having different programming styles. Nevertheless, "lines of code" is a valuable data which gives an intuitive impression about the absolute size of the source code.

	JaCo	PiCo	SJavaC	CJavaC	KerisC
files	96	46	30	39	64
lines	30269	10459	6983	11167	18115
tokens	152590	61557	43380	49745	115294
bytes	954445	382150	260805	352198	714445
classes	216	66	40	66	97
methods	1757	379	170	402	759
code size [byte]	911975	339581	236683	302058	563758
context classes	6	7	5	5	8
lines	600	378	411	416	428
tokens	1657	1177	895	1136	1481
methods	50	35	26	34	44
factory classes	7	5	3	5	6
lines	1248	166	104	231	371
tokens	6028	837	539	968	2470
methods	154	19	15	18	49
components	44	24	15	19	30
lines	17007	4928	1661	4214	10051
tokens	89163	29366	11979	15750	67474
classes	68	29	17	27	39
methods	802	231	73	171	428

Figure 4.14: Implementation of JaCo and some of its extensions.

imposed by the design patterns and programming policies which keep JaCo and its derived compilers open for future extensions. For this purpose the statistics in Figure 4.14 distinguish between the source code of the contexts, the factory classes, and the component classes.

As the Context/Component scheme from Figure 4.7 on page 169 already revealed, the base compiler defines 6 different context classes defined by a total number of 600 lines of code. Extensions almost always extend all contexts and sometimes even define additional ones. The size of the context adaptation code of the extensions amounts to 54% of the original context code size for smaller extensions and goes up to 89% for more complex extensions.

For the compiler extensions presented in this section, factory classes which are used to instantiate data structures like abstract syntax trees, symbols or type representations, are often reused with only few adaptations. From the 7 factory classes defined by JaCo, SJavaC, for instance, adapts only 3. The other compilers mostly adapt 5 classes. But even here, on the average only 18% of the 154 factory methods defined in JaCo are overridden or newly defined.

The last part of the code statistics of Figure 4.14 is supposed to show how many components, in the sense of the Context/Component pattern, are typically

	JACO2	KECO
files	66	47
lines	24143	18882
tokens	142774	120807
bytes	770705	647287
modules	63	47
submodules	58	51
classes		
declared	132	44
module support	185	141
generated	398	237
methods		
declared	1495	653
generated	4588	3782
code size [byte]		
module support	363003	442246
total	1905683	1606152

Figure 4.15: Implementation of JACO2 and its extension KECO.

written for extensions. The number of newly defined or extended components ranges from 17 for SJAVAC up to 30 for KERISC, compared to JACO which consists of 44 components.

4.4.2.2 JaCo2

For JACO2 we have currently only one extension, which is KECO, the compiler for KERIS that is also used to compile JACO2. As we saw already in the previous paragraph, implementing KERIS on top of a JAVA compiler involves substantial changes and adaptations in both the frontend and the backend. As the statistics of Figure 4.15 show, KECO's source code is, with approximately 19000 lines, almost as big as the one of the base compiler JACO2, which roughly consists of 24000 lines of code. Based on the number of lexical tokens, the size of the compiler extension corresponds to 85% of the base compiler. Implementing other language features should typically involve much less efforts.

KECO consists of 47 module declarations which either define new modules or extend one of the 63 modules of JACO2. It is surprising to discover in Figure 4.15 that even though JACO2 consists of 63 modules, its code has only 58 submodule declarations corresponding to 58 modules being instantiated within JACO2. This discrepancy between the number of declared and deployed modules is due to the fact that some of the 63 modules are actually module interfaces. Furthermore, some modules are not instantiated at all; their only purpose is to factor out code

which is shared by several specializations.

The rest of the statistics summarizes the overhead introduced by the KERIS compiler. It contrasts the number of classes and methods declared in the source code with the number of classes and methods found in the generated classfiles. From 63 declared modules, and 132 class, interface, and class field definitions, almost 400 classfiles get generated for JACO2. This overhead is mainly introduced by the KERIS compiler to translate supporting code for modules and class fields. In this translation, the number of methods also increases by a factor of three. Overall, the generated classfiles occupy more than 1.8 mbytes from which 355 kbytes correspond to artificially generated code supporting the module system. The overhead generated for KECO is comparable to the one of JACO2. The 1.5 mbytes of generated classfiles complement the binaries of the base compiler yielding binaries with a total size of 3.3 mbytes.

While the size of JACO2's binary is disproportionally bigger than the the size of JACO, JACO2's source code is actually smaller. It is smaller by almost exactly the code size which is required for implementing the context hierarchies, the factory methods, and the logic for developing, extending, and deploying components in the Context/Component framework of JACO. Unfortunately, a similar comparison is not possible for KECO, since it implements the full language and is therefore a significantly bigger system compared to its predecessor KERISC which only supports the language subset that was required for bootstrapping the system.

4.4.3 Conclusion

The purpose of this case study is two-fold: (1) it shows that even relatively complex applications can be made extensible, (2) it explains the drawbacks of conventional object-oriented techniques and motivates how the features of KERIS can be used to implement extensible applications efficiently and safely without needing to anticipate concrete future changes.

KERIS provides a programming language infrastructure that promotes a software development process for building large-scale software systems that are statically evolvable when faced with unanticipated requirements. Without such support, unanticipated changes would force programmers to perform extensive invasive modifications of existing designs and implementations.

The case study also shows that independently of the language support, software has to be explicitly designed for being extensible. This typically imposes extra costs (including runtime penalties) and implies extra efforts in the implementation process. But designing a system for facilitating later changes is essential, since software is inherently subject to changes. Even for software systems with a fixed and well-defined task where it seems that extra efforts spent on changeability make no sense, it will pay off in the long run when the software evolves (possibly into a product line). An example for such a system is JACO, which was

implemented 6 years ago, and since then served as a simple and reliable platform for rapidly implementing JAVA language extensions. Other extensible JAVA compilers that were developed in the meantime include EPP [173] and POLYGLOT [149].

Chapter 5

Related Work and Conclusions

5.1 Related Work

5.1.1 Component-Oriented Programming Languages

Unlike multi-purpose object-oriented, or functional programming languages in which software components are implemented using ordinary classes or modules by observing special programming protocols or by deploying architectural design patterns, component-oriented programming languages provide explicit linguistic abstractions for software components. Such languages have a built-in notion of a software component and provide language constructs to manufacture and compose components. Supporting component programming on the programming language level has the advantage that the compliance with the component model can easily be enforced statically either by syntactic restrictions or by a static type system.

Most multi-purpose component-oriented programming languages are built on top of JAVA-like object-oriented languages. This section will discuss some of the most prominent examples like COMPONENTJ [182, 180], ACOEL [187], ARCHJAVA [5], and CELLS [171].

ComponentJ. Seco and Caires describe COMPONENTJ, a simple typed imperative core calculus for first-class components in the context of inheritance-free object-oriented programming [182]. COMPONENTJ completely avoids inheritance in favor of object composition. Components are closed black-boxes that can be statically or dynamically composed. The language is implemented as an extension of JAVA and can be compiled to both JVM classfiles [182] and .NET assemblies [180, 181].

ACOEL. ACOEL has a component model similar to COMPONENTJ [187]. Interaction points of ACOEL components are in- and out-ports. The language is class-based and supports a restricted form of inheritance. Like in COMPONENTJ, ports are connected explicitly. As opposed to COMPONENTJ, the type system of ACOEL does not ensure that all ports are connected. On the other hand, ACOEL supports a richer form of component subtyping which is based on user-defined constraints, specified in CORAL, a language for abstracting and specifying ACOEL components [186].

ArchJava. ARCHJAVA is an extension of JAVA that tries to unify the software architecture of a system with its implementation [5]. It introduces direct support for components, connections and ports. Components are implemented with extensible component classes. ARCHJAVA does not distinguish between required and provided ports. Instead, a port declares required and provided methods. Ports are again connected with explicit plug instructions. Like the previous two languages, ARCHJAVA's composition principle is based on the aggregation of subcomponents. A distinct feature of the ARCHJAVA type system is to guarantee *communication integrity* [145].

Jiazzi. JIAZZI [128] is a language for creating large-scale binary components in JAVA. In contrast to the approaches mentioned before, JIAZZI does not extend JAVA directly — it is rather layered on top of JAVA and can be used independently as a linking language. The design of JIAZZI is based on MZSCHEME's *units* [69]. JIAZZI's units are conceptually containers of compiled JAVA classes. JIAZZI supports well-defined connections of these units through externally specified sets of imported and exported classes.

Cells. The CELLS project [171] focuses at distributed programming and mobility. Its aim is to propose a component programming language that supports an integrated notion of both compile-time and runtime components. For this purpose, the CELLS infrastructure introduces *assemblies* [120] and *cells* [171]. An assembly is a declarative, stateless piece of code that facilitates code combination. It offers typed interfaces which can be used to link smaller assemblies into bigger compound assemblies. Assemblies may be loaded at runtime, yielding a cell. A cell is a dynamic, stateful component instance which interacts with other cells via explicit runtime interfaces. Unlike all previously mentioned projects, the CELLS language has not been implemented yet.

Comel. Ibrahim formalizes COM [172] by introducing a small programming language COMEL [93]. COMEL's components do not have named ports. Services are specified solely by type names. In the spirit of COM, COMEL emphasizes aggregation and does not support implementation inheritance. COMEL components have to be self-contained, not having any context dependencies. This is a

severe restriction that contradicts the aim to modularize software into small components that have to depend on their deployment context in order to be flexibly reusable.

5.1.2 Architecture Description Languages

Component-oriented programming languages feature concepts originating from *architecture description languages* (ADL) [130] like ACME [76], Aesop [75], Darwin [126], Rapide [121], Wright [8], SOFA/DCUP [165] etc. In general, architecture description languages are used to specify the architecture of software systems in a formal way. A software architecture describes the organization of a software system in terms of a collection of distinct components, connections between the components, and constraints on these interactions [159, 184, 195]. By using architecture description languages, the details of a design get explicit and more precise, enabling formal analysis techniques. Furthermore, such languages can help to understand the structure of a system, its implementation and use, as well as its reuse capabilities. A comparison of popular architecture description languages is presented in [130].

Extensibility on the component level is strongly related to *architectural evolution*, a research area concerned with the addition, removal, or replacement of components and connectors both statically and dynamically.

5.1.3 Software Composition Languages

Component-oriented programming languages are typically multi-purpose programming languages which provide linguistic facilities for implementing and composing software components. While these languages make it easy to compose components written in the same language, they have fundamental restrictions when components written in other languages have to be integrated. Languages that focus on software composition solely have typically better support for integrating components that conform to different component models and *architectural styles* [184] and that are developed using different technologies or languages. A specially-designed language is also better for explaining, highlighting, and exploring compositional issues as opposed to general-purpose programming issues [2].

The software composition language Piccola [2, 3] is based on the principle: *Applications = Components + Scripts*. In this formula, *components* are black-box abstractions with *plugs* (i.e. exported and imported services) whereas *scripts* specify how components are plugged together. Piccola models components and compositional abstractions by means of communicating concurrent agents. Interfaces of components as well as the contexts in which they get deployed are modeled by *forms*, a special notion of extensible, immutable records. Apart from components and scripts, *glue abstractions* are needed to bridge architectural styles

and adapt otherwise incompatible components [179]. Finally, *coordination abstractions* may be required to handle dependencies between concurrent and distributed components. The challenge in the design of PICCOLA was to identify a minimal set of features necessary and sufficient for specifying software compositions as scripts while supporting an open-ended set of architectural styles by allowing to define new, higher-level composition and coordination mechanisms in terms of lower-level ones. PICCOLA is based on a formal foundation, the $\pi\mathcal{L}$-calculus [122], a variant of the polyadic π-calculus [142] supporting forms.

There are a few industrial composition languages that are based on XML [33]. Among these languages is Sun's JAVABEAN PERSISTENCE [141] and IBM's BEAN MARKUP LANGUAGE [204]. Unlike PICCOLA which allows to integrate components developed using different technology, both systems are tailored specifically for JAVABEANS [190].

Languages that are mostly concerned with managing interactions and dependencies between concurrent and distributed components are called *coordination languages*. Classical coordination languages are, for instance, LINDA [47] and DARWIN [126].

5.1.4 Module Systems

Imperative module systems. Classical module systems like the one of MODULA-2 [206], MODULA-3 [44], OBERON-2 [146], and ADA 95 [196] can be used to model modular aspects of software components well, but they have severe restrictions concerning extensibility and reuse. These systems allow for type-safe separate compilation, but they hard-wire module dependencies by referring to other specific modules by name. This makes it impossible to plug in modules with different names but compatible specifications without performing a consistent renaming on the source code level.

The module systems of OBERON-2 and C# [86] allow to define local aliases for imported modules or classes. Here, one can easily replace an imported module with a compatible version just by modifying an alias definition. Such a modification would be destructive and would require a global recompilation, but it would not require to rename module references in the source code.

Functional module systems. Initially, functional programming languages introduced module systems that obey the principle of *external connections* [70], i.e. the separation of component definition and component connection. These module systems promote reuse, but they yield modules that are not extensible, since everything is hard-wired internally. Prominent module systems with external linking facilities are provided by SML [143], OCAML [111], and MZSCHEME [69].

In SML it is possible to parameterize modules with other external modules [123]. Such higher-order modules are called *functors*. SML functors are neither first-class nor higher-order. Consequently, they cannot be used to dynam-

ically manufacture new modules. Furthermore, they are not extensible, which makes it difficult to perform adaptations. An extension of SML with first-class modules was recently proposed by Russo [175, 176]. There are also proposals for adding inheritance and subtyping to SML-style module systems [144].

In contrast to functors in SML, higher-order modules in OCaml are applicative and they have good support for separate compilation. OCaml also offers several advanced module reuse mechanisms like module inclusion and mixin module composition [88].

OCaml's mixin modules are based on CMS [10, 12], a simple but expressive module calculus which can be instantiated over an arbitrary core calculus. The calculus supports various module composition mechanisms including mixin module composition with overriding. The work on mixin-based composition goes back to Bracha who observed that inheritance can be seen as a general mechanism for modular program composition [31, 29]. With his work on the programming language Jigsaw [28], he lifts the notion of class-based inheritance and overriding to the level of modules. The first proposal for mixin modules in ML goes back to Duggan and Sourelis [57].

As opposed to modules in ML, MzScheme's *units* [71] offer separate compilation of independent modules even with cyclic dependencies. With the unit system, MzScheme provides first-class module abstractions and linking facilities to compose modules hierarchically. Units are linked by explicitly connecting provided with required ports. A general problem of unit-style module systems is scalability of the wiring mechanism due to modules importing fine-grained entities like classes, functions, etc. and due to the need to wire the ports of modules explicitly. For this reason, MzScheme offers *signed units* that support bundles of variables, called signatures, which get linked all in one step [70].

Object-oriented module systems. Rüping analyzes the modularity of object-oriented systems during design and specification in [174]. He substantiates the need for modules in object-oriented languages as a means to encapsulate cooperating classes. The module refinement and specialization mechanisms of Keris implement his abstract notion of compatibility between modules. This notion enables the type-safe extension of systems where modules get substituted with compatible implementations.

Only recently, concrete proposals have been put forward to bundle class-based object-oriented programming languages with expressive module systems [64, 19, 11, 53].

Corwin, Bacon, Grove, and Murthy's module system MJ [53] adds advanced modularity and linking capabilities to Java. MJ replaces Java's *classpath* mechanism, which is often used as a low-level means for linking classes, with a pragmatic COM-inspired module system in which modularity constraints can be statically checked and dynamically enforced.

Ancona and Zucca propose another module system for JAVA-like programming languages [11]. While MJ is based on classical module systems with specific references to external modules, Ancona and Zucca's system provides more advanced abstraction and composition mechanisms. The proposal is quite expressive, even allowing to fully abstract over external classes. On the other hand, their module system is rather theoretical, leaving unclear if it is feasible to implement it in practice.

Ichisugi and Tanaka observe that extensibility on the module level greatly enhances the ability to extend and reuse object-oriented software [94]. The authors describe a practical module system for JAVA based on the notion of *difference-based modules*. Difference-based modules are solely linked by a form of multiple inheritance which also merges module members. Since modules cannot abstract over their context dependencies (which are hard-wired), this module system must be rather seen as a tool for *aspect-oriented programming* [103] than for developing context independent components.

Extensibility and reuse mechanisms. Most module systems provide module abstractions for creating static black-box software components. Only very few module systems allow programmers to create new versions or specializations of existing modules.

A module specialization mechanism similar to the one offered by KERIS is proposed by Radenski for a PASCAL dialect [168]. Radenski's *module embedding* proposal [167] enables the building of new modules from existing ones through inheritance and overriding of procedures and types. Since it is not possible to specialize imports, it is questionable to what extend this mechanism is useful in practice, where depending modules often have to be specialized simultaneously. Since embeddable modules are static abstractions which are not associated with a module type, it is neither possible to create multiple instances, nor replace a compatible module with an extended implementation.

Hultgren proposes a module system for SQUEAK [97] in which modules encapsulate changes relative to an existing base module (e.g. for adding new methods to existing classes) [92]. Therefore, the idea behind Hultgren's *delta modules* is comparable to the one behind Ichisugi and Tanaka's difference-based modules. A similar, but more elaborated mechanism for extending modules in SQUEAK was recently proposed by Bergel, Ducasse, and Wuyts. Their *classbox* module system [22] allows method additions and replacements while supporting the notion of *local rebinding* which makes changes made by a glassbox only visible to the glassbox itself and to other classboxes that import it. This feature is similar to the module refinement mechanism of KERIS which affects only other modules that refer to the refinement and not the former version. Thus, local rebinding and the refinement notion of KERIS prevent local changes from having a global impact.

5.1.5 Object-Oriented Programming

Instead of equipping object-oriented languages with module systems, some recent object-oriented programming languages try to achieve a similar goal by improving the expressiveness of class abstractions. Only such languages really substantiate Meyer's bold thesis that classes are the better modules [134]. As an example, we briefly discuss the class-based object-oriented multi-purpose programming language Scala [153, 151].

Scala. One of the original design goals of Scala was to provide powerful class abstraction and composition facilities that subsume the functionality of module systems. But instead of introducing many small ad-hoc fixes, Scala's class design is based on a smooth and coherent integration of *abstract types* as class members together with a mixin class composition scheme as a generalization of inheritance. In addition to these features, Scala also supports many advanced programming language features like traits, bounded parametric polymorphism, variance annotations, explicitly typed self references in classes, compound types, and regular expression pattern matching.

As first experience with Scala shows, the type system of the language is quite expressive (probably, much more expressive than most module systems), allowing to write reusable and extensible software in a safe, but not necessarily modular way. A drawback of abstractions that are designed to be as general as possible and type systems to be as expressive as possible is that they do not enforce a particular programming style with specific, desirable qualities; e.g. a programming style that requires to separate interfaces from implementations, that makes context dependencies explicit and generic, that defines extensible types, etc. It is possible to implement software components with these qualities, but in practice this is often technically also quite challenging. Here, the existence of an explicit module system has the advantage to guide the programmer in the implementation of reusable software components that conform to some standard set by the underlying module system. The conformance to such a standard makes it, for instance, easy for third parties to deploy the module in a new context.

GBeta and FamilyJ. Languages with similar properties, in particular, type systems and composition mechanisms, are gBeta [58] and FamilyJ [208] (a subset of the aspect-oriented language Caesar [137]). While gBeta supports some form of multiple-inheritance for its *patterns* [125], FamilyJ's main composition mechanism is object-based, and relies on *delegation* [156, 155]. Unlike Scala, which only supports extensibility for abstract types, gBeta and FamilyJ support *virtual classes* [124]; i.e. both languages allow to *furtherbind* [61] inner class definitions in subclasses. Furtherbindings enhance existing classes by adding new class members or by replacing existing members. Radenski's proposal [169] for a class overriding mechanism in Java is similar to furtherbindings in FamilyJ.

Abstract types vs. virtual classes vs. class fields. Class fields in KERIS offer a compromise between purely abstract types and fully virtual classes. Abstract types [153] are open in the sense that their bounds can be covariantly overridden in subclasses, but they first have to be closed explicitly before the class which defines the abstract type member can be instantiated. Another drawback is that abstract types do not refer to classes and therefore cannot be used for creating objects — explicit factory methods have to take over this task. Virtual classes on the other hand are open for extensions while at the same time being instantiatable. A serious disadvantage of virtual classes is their complicated scoping and name resolution rules [61], and ad-hoc means to prevent name clashes and accidental overrides (when virtual classes are subclassed) [208]. Class fields are extensible class abstractions which can be overridden, and which can be instantiated. The difference to virtual classes is that they do not support implementation inheritance. On the other hand, their notion of overriding is more powerful: class implementations can be fully exchanged with different, compatible versions. Virtual classes only allow furtherbindings which enhance an existing implementation, but which do not allow to fully replace an existing implementation.

5.1.6 Aspect-Oriented Programming

While people that are mostly interested in component technology emphasize explicit interfaces and context dependencies to make component abstractions black-box extensible and deployable, the aspect-oriented programming (AOP) [103] community is mainly interested in modularizing crosscutting concerns and focuses on source code-centric white-box code reuse and invasive source code-based software composition [15].

AOP techniques make it possible to modularize crosscutting aspects of a system, and therefore facilitate the separation of different concerns [198], promoting modularity, extensibility, and code reuse in general. A system consisting of various "aspect slices" is assembled by an *aspect weaver* which merges the fragments into a whole.

The most prominent AOP language is ASPECTJ [102]. Aspects in ASPECTJ were originally combined using a source code-based compile-time weaver. Only recently, a load-time-based weaving scheme was proposed to allow an on-demand composition of binary aspects at runtime.

Related to AOP is the notion of *collaboration-based designs* [89, 202]. This term describes a methodology for decomposing an object-oriented system into a set of classes and a set of *collaborations*. A collaboration represents a particular aspect of a system and consists of several participating classes. A class can be a member of several collaborations in which it typically plays different *roles*. Thus, a collaboration is said to *crosscut* the class structure.

Several programming languages support collaboration-based designs with explicit modularization constructs. The aspect-oriented programming language

Caesar [137, 138] offers the latest and most advanced approach. It promotes the notion of *on-demand re-modularization* with specific *collaboration interfaces* which facilitate the definition of *object roles* and the creation of wrappers to implement such roles [136]. Caesar's collaboration interfaces extend and generalize earlier work on *pluggable composite adaptors* [139] and *adaptive plug-and-play components* [135]. A long term aim of Caesar is to provide support for *fluid aspect-oriented programming*, a term coined by Kiczales [101]. It involves the ability to temporarily shift an implementation of a system to a different structure to do some piece of work with it before shifting it back to its original form.

Very similar to *collaboration interfaces* are *object teams* [87]. Object teams are containers for role classes which may be dynamically bound to base objects of a particular application. This approach is based on delegation, furtherbindings, and a dependent type system featuring types as object members.

Some approaches that focus on collaborations are based on mixins. A consistent refinement of a family of classes is possible with the notion of *mixin layers*, introduced by Smaragdakis and Batory [185]. Related to mixins is the concept of *delegation*. Integrated into a statically typed object-oriented language, delegation yields a powerful mechanism for object-based inheritance [105, 39]. Recently, Ostermann unified delegation with the mixin layer concept in his work on *delegation layers* [156].

Mixin layers where originally motivated by Batory's work on GenVoca. GenVoca is a design methodology for creating object-oriented system families and statically extensible software; i.e. software that is customizable via module additions and removals at compile-time [18]. GenVoca allows components to be refined step-wise where each refinement adds a particular feature to the component. Feature refinements are typically implemented using generative programming technology like templates [185], program generators [135], and program transformers.

Related to aspect-oriented programming is *subject-oriented programming* [85] and *adaptive programming* [114]. Subject-oriented programming (SOP) is a program composition technology that supports building object-oriented systems as compositions of subjects. A *subject* is a collection of class fragments whose class hierarchy models its domain in its own subjective way. Subject composition combines class hierarchies to produce new subjects that incorporate functionality from existing subjects. SOP targets various software development problems, including the creation of extensions and configurations of software, the customization and integration of systems and reusable components, as well as the decentralized development of classes. SOP is seen as a language-independent technology which is implemented typically at the level of integrated development environments like IBM's Visual Age.

Adaptive programming (AP) enables a form of AOP where some of the building blocks of software are expressible in terms of graphs and where the other building blocks refer to the graphs using traversal strategies [113]. In the AP

world, crosscutting concerns are expressed adaptively using strategies to embed small graphs into large graphs. A key feature of this embedding is that it is specified by abstracting over the details of the large graphs, making the graph embeddings *adaptive*.

An alternative approach to AOP offers the *composition filters* model [4, 23]. A composition filter is an entity that transforms messages to/from objects. Each filter enhances a class in a modular way; i.e. without necessarily modifying the definition of that class. In the AOP terminology, filters correspond to aspects.

5.2 Summary

Extensibility is a desirable property for software artifacts on all abstraction levels. It promotes reusability and facilitates software evolution. Nevertheless, *designing* an extensible system requires much more efforts than designing a static system with fixed functionality. Similarly, it is technically much more challenging to *implement* a system which is open for future extensions in comparison to closed systems which do not explicitly provide an extension or adaptation logic.

While extensibility is quite well investigated and well understood *in the small*, i.e. for a single class or the implementation of an algorithm or a small subsystem, it remains a challenge to implement extensible software *in the large*; i.e. on the level of software components. This observation is quite natural given the fact that software components are often seen as pure black-box abstractions. Since black-box software artifacts do not reveal implementation details, it is almost impossible to adapt or extend them if an appropriate extension logic has not been anticipated by the original developer of the software and if simple adaptation techniques for black-box components, like wrapping, fail.

This thesis focuses on extensibility on the level of software components and component systems. Its main objectives are:

- to point out why extensibility is important even for large software components,
- to show what problems programmers are typically facing when developing extensible software components,
- to propose concrete programming language abstractions that help programmers to implement extensible software components more efficiently and safely,
- to demonstrate with a case study that it is feasible to implement even large and relatively complex systems in an extensible fashion out of components that are themselves extensible, and
- to explain with the help of a case study how the proposed programming language abstractions support the process of building and extending an extensible application.

The first part of the thesis tries to explain the notion of software components and concepts related to component technology in a formal framework. It investigates what we mean by extensible software components and points out mechanisms for extending and adapting software components safely. This is done by discussing a typed component model that is designed to support the implementation and evolution of lightweight, extensible components in object-oriented programming languages. The model supports dynamic component manufacturing and composition in a type-safe way through a small set of component refinement primitives. In contrast to other approaches, the component model does not require that services of components are linked explicitly by plugging ports. Instead, components are composed using high-level composition operators. The component model is formalized as an extension of *Featherweight Java* to show how it integrates into the context of a statically typed object-oriented programming language.

For simplicity, the model assumes that services and service implementations are defined outside of a component in form of object types (interfaces) and classes. A real-world programming language for component programming has to take encapsulation more seriously and provide means to define interfaces and classes inside of a component. The consequences of this requirement are elaborated in the second part of this thesis, which discusses KERIS, an extension of JAVA. KERIS integrates a module system into JAVA which is designed to facilitate the development of extensible software components. The design acknowledges some fundamental principles outlined in the theoretical model:

- components are abstractions that provide and require services,
- components have to be distinguished from component instances, which embody concrete interlinked incarnations of components,
- components are not plugged together manually by linking ports; instead, the component wiring is inferred from the deployment context,
- components are extensible abstractions,
- component definition and extension is subject to a statical type system, and
- types of components are described with the help of nominal types.

Unlike the theoretical model, components, or modules as they are called in KERIS, are not first-class and can consequently not be manufactured and composed dynamically. There is also no support for mixin compositions of modules. Modules are solely composed by aggregation and extended by refinements or specializations. The extension mechanism of KERIS allows programmers to extend atomic modules as well as fully linked systems statically by replacing subsystems with compatible versions. In such a case, there is no need to re-link the full system. Furthermore, the old version is not destructively overridden with the new configuration; it persists and can still be deployed in different contexts.

Apart from the module abstraction, composition, and evolution features of KERIS, it is the notion of *class fields* which enables the development of extensible object-oriented software. The tradeoffs in comparison with abstract types and virtual classes were already discussed in Section 5.1.5. What makes class fields suitable for KERIS is that they strictly separate object interfaces from object implementations (classes), a key requirement for expressive module systems. They also help to separate the stable parts of a system, the interfaces and implementations, from the flexible parts, the configurations which define how the services specified in the interfaces are implemented at runtime. What class fields, abstract types, and virtual classes have in common is the nature of the type system which is required to deal with such abstractions. Types defined by classes that may be covariantly overridden in extensions of the enclosing class or module have to be bound to the enclosing instance; i.e. types are *dependent* on objects or module instances.

The third part of this thesis describes our experience with extensible compilers. This section explains that the language features offered by KERIS make it indeed quite simple to safely implement extensible, component-based software, even for relatively complex domains. On the other hand, the chapter also shows that design decisions have a significant impact on the reuse and extensibility capabilities of an application. If software is not designed with extensibility in mind, it will be hard to evolve in future, no matter what language it is written in. Such software typically has to be refactored first, before features like module refinement and specialization can be exploited appropriately. The significance of a language like KERIS is to provide an infrastructure in which it is easy to turn a design for an extensible component-based application into real software without resorting to complicated design pattern-based approaches which are difficult to design, tedious to set up and maintain, and whose usage cannot be enforced statically with a type system.

5.3 Future Work

KERIS, as it is presented in this thesis, focuses on the static evolution of systems. With the reflective infrastructure of KERIS, it is possible to link new modules into an existing context, or to hot swap modules dynamically, but this is inherently unsafe since this process is neither under the supervision of the static nor the dynamic type system. All safety guarantees are provided at runtime by the reflection library.

While reflective mechanisms are predominantly used for evolving systems today, little work has been done on languages and runtime environments in which a static type system can guarantee that dynamic component updates or component extensions are type-safe. Furthermore, safe hot swap mechanisms are required that have a well-defined semantics, in particular, if concurrency is in-

volved; as opposed to the ad-hoc approaches of KERIS and even ERLANG, which both require some degree of anticipation and support by the programmer of a module to avoid race conditions and type incompatibilities at runtime.

Apart from support for the dynamic evolution of component systems, it is worth considering to integrate more advanced module composition mechanisms into KERIS. Such mechanisms may include mixin module compositions or mechanisms that allow programmers to wire modules with explicit connector abstractions.

Appendix A

Type Soundness for Prototype-Based Components

In this section we present the full type soundness proof for our type system in Figure 2.12 with the weaker typing rules explained in Section 2.3.4. The presentation follows the style of the original type soundness proof of Featherweight Java [95]. The formalization of the type system is based on a fixed class table CT. For the subject reduction proof we have to assume that classes in CT are well-typed.

A.1 Subject Reduction

Lemma A.1.1 (Subtyping) The subtyping relation $<:$ is reflexitive and transitive; i.e. $T <: T$ and for $T <: U$ and $U <: V$, we also have $T <: V$.

Proof: For object types, the reflexitivity and transitivity are explicitly defined. For component and component instance types, these properties get inherited from the subset relation $\subseteq$. □

Lemma A.1.2 (Well-formed types) If all types in Γ are well-formed and $\Gamma \vdash e : T$ then T wf.

Proof: By a straightforward induction on a derivation of $\Gamma \vdash e : T$. Only component types are non-trivial due to the required disjointness of the provided and required services. Note that all typing rules that yield component types include this disjointness requirement explicitly. □

Lemma A.1.3 (Invariant method overriding) If $\text{mtype}(m, D) = \overline{T} \to T'$, then $\text{mtype}(m, C) = \overline{T} \to T'$ for all $C <: D$.

Proof: By induction on the derivation of $C <: D$. We suppose that $\text{mtype}(m, D) = \overline{T} \to T'$ and $C <: D$, and show that $\text{mtype}(m, C) = \overline{T} \to T'$.

Case 1: $C = D$
Trivial.

Case 2: $C <: D \qquad CT(C) = \texttt{class}\ C\ \texttt{extends}\ D\ \{\ldots\}$
We have to distinguish two cases, depending on whether m is overridden in C or not. If m is not defined in C, then we derive from the definition of mtype the required result $\text{mtype}(m, C) = \text{mtype}(m, D) = \overline{T} \to T$. For the case that m is defined in class C and thus overrides method m in class D, we look at the derivation of the method typing for method m:

$$\dfrac{\dfrac{\text{mtype}(m, D) = \overline{T} \to T' \ \text{ impl. } \ \overline{U} = \overline{T}, U' = T'}{\text{override}(m, D, \overline{U} \to U')} \quad \ldots}{U'\, m(\overline{U}\ \overline{x})\ \{\,\texttt{return}\ e;\ \}\ \text{ok in } C}$$

With the premise of the overrides clause we finally get the result $\text{mtype}(m, C) = \overline{T} \to T'$.

Case 3: $C <: D \qquad C <: E \qquad E <: D$
By the induction hypothesis, $\text{mtype}(m, E) = \overline{T} \to T'$. Another application of the induction hypothesis yields $\text{mtype}(m, C) = \overline{T} \to T'$. □

Lemma A.1.4 (Context permutation) If $\Gamma, x : U, y : V, \Gamma' \vdash e : T$ then $\Gamma, y : V, x : U, \Gamma' \vdash e : T$.

Proof: By a straightforward induction on the typing derivation $\Gamma, x : U, y : V, \Gamma' \vdash e : T$. Note that we assume that binders always introduce fresh names. In particular, $x \neq y$, $\{x, y\} \cap \text{dom}(\Gamma, \Gamma') = \emptyset$ and $\text{dom}(\Gamma) \cap \text{dom}(\Gamma') = \emptyset$. □

Lemma A.1.5 If $\Gamma, x : U \vdash e : T$ and $U' <: U$, then $\Gamma, x : U' \vdash e : T'$ for some $T' <: T$.

Proof: By induction on the derivation of $\Gamma, x : U \vdash e : T$.

Case T-Var: $e = y \qquad T = \Gamma(y)$
We have to consider two subcases, depending on whether y is the same as x. For $y = x$ we get $\Gamma, x : U' \vdash x : T'$ with $T' = U' <: U = T$. If $x \neq y$, then $\Gamma, x : U' \vdash y : T'$ with $T' = T$.

Case T-Fld: $e = e_0.f_i \qquad T = T_i \qquad \Gamma, x : U \vdash e_0 : C \qquad \text{fields}(C) = \overline{T}\ \overline{f}$
By the induction hypothesis, $\Gamma, x : U' \vdash e_0 : D$ for some $D <: C$. It can be shown easily that $\text{fields}(D) = \text{fields}(C), \overline{V}\ \overline{g}$. Therefore, we can apply rule (T-Fld) and get $\Gamma, x : U' \vdash e_0.f_i : T$.

Case T-Inv: $e = e_0.m(\overline{e'}) \qquad \Gamma, x : U \vdash e_0 : C \qquad \text{mtype}(m, C) = \overline{V} \to T \qquad \Gamma, x : U \vdash \overline{e'} : \overline{W} \qquad \overline{W} <: \overline{V}$

By the induction hypothesis:
$\Gamma, x : U' \vdash e_0 : D$ with $D <: C$
$\Gamma, x : U' \vdash \overline{e'} : \overline{W'}$ with $\overline{W'} <: \overline{W} <: \overline{V}$

With Lemma A.1.3, $\text{mtype}(m, D) = \overline{V} \to T'$ with $T' = T$. Now we apply rule (T-Inv) and get the needed result $\Gamma, x : U' \vdash e_0.m(\overline{e'}) : T$.

Case T-New: $e = \text{new } C(\overline{e'})$ $\quad \Gamma, x : U \vdash \overline{e'} : \overline{W}$
$T = C$ $\quad \overline{W} <: \overline{T'}$
$\text{fields}(C) = \overline{T'}\,\overline{f}$

By the induction hypothesis, $\Gamma, x : U' \vdash \overline{e'} : \overline{V}$ with $\overline{V} <: \overline{W} <: \overline{T'}$. With rule (T-New) we conclude that $\Gamma, x : U' \vdash \text{new } C(\overline{e'}) : C$.

Case T-Inst: $e = \text{new } e_0$ $\quad \Gamma, x : U \vdash e_0 : \varnothing \Rightarrow \overline{C}$
$T = [\overline{C}]$

By the induction hypothesis and the subtype relation, $\Gamma, x : U' \vdash e_0 : \varnothing \Rightarrow \overline{D}$ with $\overline{C} \subseteq \overline{D}$. With (T-Inst) we derive $\Gamma, x : U' \vdash \text{new } e_0 : [\overline{D}]$. The subtype relation for component instances completes the case with $T' = [\overline{D}] <: T$.

Case T-Serv: $e = e_0 :: C_i$ $\quad \Gamma, x : U \vdash e_0 : [\overline{C}]$
$T = C_i$

The induction hypothesis yields $\Gamma, x : U' \vdash e_0 : [\overline{D}]$ for some $\overline{D}$ with $[\overline{D}] <: [\overline{C}]$. That is, $\overline{C} \subseteq \overline{D}$ and therefore $C_i \in \overline{D}$. Now we apply (T-Serv) to get the required result $\Gamma, x : U' \vdash e_0 :: C_i : T$.

Case T-Com: Trivial.

Case T-Mix: $e = e_0 \texttt{ mixin } e_1$
$T = (\overline{C} \cup \overline{D}) \setminus (\overline{C'} \cup \overline{D'}) \Rightarrow \overline{C'} \cup \overline{D'}$
$\Gamma, x : U \vdash e_0 : \overline{C} \Rightarrow \overline{C'}$
$\Gamma, x : U \vdash e_1 : \overline{D} \Rightarrow \overline{D'}$

By the induction hypothesis:
$\Gamma, x : U' \vdash e_0 : \overline{E} \Rightarrow \overline{E'}$ with $\overline{E} \Rightarrow \overline{E'} <: \overline{C} \Rightarrow \overline{C'}$
$\Gamma, x : U' \vdash e_1 : \overline{F} \Rightarrow \overline{F'}$ with $\overline{F} \Rightarrow \overline{F'} <: \overline{D} \Rightarrow \overline{D'}$

Rule (T-Mix) yields $\Gamma, x : U' \vdash e_0 \texttt{ mixin } e_1 : T'$ with $T' = (\overline{E} \cup \overline{F}) \setminus (\overline{E'} \cup \overline{F'}) \Rightarrow \overline{E'} \cup \overline{F'}$. It remains to show that $T' <: T$. From the clauses derived by the induction hypothesis we conclude using the subtyping rules and Lemma A.1.1:

$\overline{E} \subseteq \overline{C}$ $\quad \overline{C'} \subseteq \overline{E'}$
$\overline{F} \subseteq \overline{D}$ $\quad \overline{D'} \subseteq \overline{F'}$

Simple set theory yields:

$(\overline{E} \cup \overline{F}) \setminus (\overline{E'} \cup \overline{F'}) \subseteq (\overline{C} \cup \overline{D}) \setminus (\overline{C'} \cup \overline{D'})$
$\overline{C'} \cup \overline{D'} \subseteq \overline{E'} \cup \overline{F'}$

With the subtyping rule for components we finally get $T' <: T$.

Case T-Req: $e = e_0 \texttt{ requires } C$ $\quad \Gamma, x : U \vdash e_0 : \overline{D} \Rightarrow \overline{D'}$
$T = \overline{D} \cup C \Rightarrow \overline{D'} \setminus C$

By the induction hypothesis, $\Gamma, x : U' \vdash e_0 : \overline{E} \Rightarrow \overline{E'}$ with $\overline{E} \Rightarrow \overline{E'} <: \overline{D} \Rightarrow \overline{D'}$. With rule (T-Req) we derive $\Gamma, x : U' \vdash e_0 \texttt{ requires } C : \overline{E} \cup C \Rightarrow \overline{E} \setminus C$. By the definition of $<:$ we get $\overline{E} \subseteq \overline{D}$ and $\overline{D'} \subseteq \overline{E'}$. We can now easily show that this implies $T' = (\overline{E} \cup C \Rightarrow \overline{E} \setminus C) <: T$.

Case T-Prv': $e = e_0$ provides C as x with d
$T = (\overline{D''} \cup \overline{D}) \setminus (\overline{D'} \cup C) \Rightarrow \overline{D'} \cup C$
$\Gamma, x : U \vdash e_0 : \overline{D} \Rightarrow \overline{D'}$
$\Gamma, x : U, y : [\overline{D''}] \vdash d : B$
$B <: C$

With the induction hypothesis we get $\Gamma, x : U' \vdash e_0 : \overline{E} \Rightarrow \overline{E'}$ with $\overline{E} \Rightarrow \overline{E'} <: \overline{D} \Rightarrow \overline{D'}$. By Lemma A.1.4, $\Gamma, y : [\overline{D''}], x : U \vdash d : B'$. This time the induction hypothesis yields $\Gamma, y : [\overline{D''}], x : U' \vdash d : B'$ with $B' <: B$. After another application of Lemma A.1.4 and by using the transitivity property of $<:$, we can now make use of rule (T-Prv'). We get $\Gamma, x : U' \vdash e : T'$ with $T' = (\overline{D''} \cup \overline{E}) \setminus (\overline{E'} \cup C) \Rightarrow \overline{E'} \cup C$. It remains to show that $T' <: T$. Since $\overline{E} \Rightarrow \overline{E'} <: \overline{D} \Rightarrow \overline{D'}$ we know from the definition of $<:$ that $\overline{E} \subseteq \overline{D}$ and $\overline{D'} \subseteq \overline{E'}$. Therefore, we also have $\overline{E} \cup \overline{D''} \subseteq \overline{D} \cup \overline{D''}$. Since we know that $\overline{D'} \subseteq \overline{E'}$, we finally get $(\overline{D''} \cup \overline{E}) \setminus (\overline{E'} \cup C) \subseteq (\overline{D''} \cup \overline{D}) \setminus (\overline{D'} \cup C)$. Now, it is easy to see that $T' <: T$.

Case T-Fwd': Similar to (T-Prv'). □

Lemma A.1.6 (Substitution preserves typing) If $\Gamma, \overline{x} : \overline{T} \vdash e : U$, and $\Gamma \vdash \overline{d} : \overline{V}$ where $\overline{V} <: \overline{T}$, then $\Gamma \vdash [\overline{d}/\overline{x}]e : W$ for some $W <: U$.

Proof: By induction on the derivation of $\Gamma, \overline{x} : \overline{T} \vdash e : U$. The proof is similar to the one of Lemma A.1.5. Instead of applying the induction hypothesis twice for cases (T-Prv') and (T-Fwd'), we now make use of Lemma A.1.5. □

Lemma A.1.7 (Weakening) If $\Gamma \vdash e : T$, $x \notin \mathrm{dom}(\Gamma)$, then $\Gamma, x : U \vdash e : T$.

Proof: By a straightforward induction on the derivation of $\Gamma \vdash e : T$. □

Lemma A.1.8 If $\mathrm{mtype}(m, C) = \overline{T} \to T'$, and $\mathrm{mbody}(m, C) = (\overline{x}, e)$, then for some D with $C <: D$, there exists some $U <: T'$ such that $\overline{x} : \overline{T}, \mathtt{this} : D \vdash e : U$.

Proof: By induction on the derivation of $\mathrm{mbody}(m, C)$. We assume that all classes are well-typed. So we can make use of (T-Meth) in the base case where m is defined in C. We immediately get $\overline{x} : \overline{T}, \mathtt{this} : D \vdash e : U$ for some $U <: T'$. The induction step is straightforward. □

Lemma A.1.9 If $\mathrm{service}(d, e, C) = d'$, with $\Gamma \vdash d : [\overline{E}]$, $\Gamma \vdash e : \overline{F} \Rightarrow \overline{F'}$, $C \in \overline{F'}$, and $\overline{F} \cup \overline{F'} \subseteq \overline{E}$, then $\Gamma \vdash d' : B$ for some $B <: C$.

Proof: By induction on a derivation of $\mathrm{service}(d, e, C)$ for a given d and C.

Base case 1: $e = e_0$ provides C as x with d_0 $\qquad d' = [d/x]\, d_0$
The last rule used for typing e is (T-Prv'):

$$\frac{\Gamma \vdash e_0 : \overline{D} \Rightarrow \overline{D'} \qquad \Gamma, x : [\overline{D''}] \vdash d_0 : B' \qquad B' <: C}{\Gamma \vdash e : F \Rightarrow F'}$$

with $F = (\overline{D} \cup \overline{D''}) \setminus (\overline{D'} \cup C)$ and $F' = \overline{D'} \cup C$. With $F \cup F' \subseteq \overline{E}$ we get $\overline{D''} \subseteq \overline{E}$ and therefore $[\overline{E}] <: [\overline{D''}]$. Now we can derive $\Gamma, x : [\overline{E}] \vdash d_0 : B''$ with $B'' <: B'$

by Lemma A.1.5. Lemma A.1.6 finally yields the required result $\Gamma \vdash d' : B$ where $B <: B'' <: B' <: C$.

Base case 2: $e = e_0$ `forwards` $\overline{D}$ `as` x `to` $d_0 \qquad d' = [d/x]\, d_0 :: C$
$C \in \overline{D}$

The proof is similar to the one of base case 1.

Induction step 1: $e = e_0$ `provides` D `as` x `with` $d_0 \qquad D \neq C$
The last rule used for typing e is (T-Prv'):

$$\frac{\Gamma \vdash e_0 : \overline{G} \Rightarrow \overline{G'} \qquad \Gamma, x : [\overline{G''}] \vdash d_0 : B' \qquad B' <: D}{\Gamma \vdash e : F \Rightarrow F'}$$

with $F = (\overline{G} \cup \overline{G''}) \setminus (\overline{G'} \cup D)$ and $F' = \overline{G'} \cup D$. Now we get $\overline{G} \cup \overline{G'} \subseteq \overline{G} \cup \overline{G'} \cup \overline{G''} = \overline{F} \cup \overline{F'} \subseteq \overline{E}$. Since $C \neq D$ and $C \in \overline{G'} \cup D$, we get $C \in \overline{G'}$. Now we apply the induction hypothesis and get service$(c, e_0, C) = d'$ with $\Gamma \vdash d' : B$ and $B <: C$.

Induction step 2: $e = e_0$ `forwards` $\overline{D}$ `as` x `to` $d_0 \qquad C \notin \overline{D}$
The proof is similar to the one of induction step 1. □

Theorem 2.3.1 (Subject reduction) If all types in Γ are well-formed, $\Gamma \vdash e : T$ and $e \longrightarrow e'$, then $\Gamma \vdash e' : T'$ for some $T' <: T$.

Proof: By induction on a derivation of $e \longrightarrow e'$ with a case analysis on the reduction rule used. We suppose that $\Gamma \vdash e : T$ and show for each case $\Gamma \vdash e' : T'$ with $T' <: T$.

Case R-Fld: $e = \text{new } C(\overline{d}).f_i \qquad e' = d_i \qquad \text{fields}(C) = \overline{U}\ \overline{f}$
With rule (T-Fld) and (T-New) we derive $\Gamma \vdash \overline{d} : \overline{V}$ with $\overline{V} <: \overline{U}$ and $T = U_i$. In particular, we have $\Gamma \vdash d_i : V_i$ with $T' = V_i <: U_i = T$.

Case R-Serv: $e = \text{new } e_0 :: C \qquad e' = \text{service}(\text{new } e, e, C)$
With (T-Serv) and (T-Inst) we derive
$\Gamma \vdash \text{new } e_0 : [\overline{D}]$ with $T = C = D_i$
$\Gamma \vdash e_0 : \varnothing \Rightarrow \overline{D}$
Lemma A.1.9 concludes this case with $\Gamma \vdash e' : T'$ for some $T' <: C = T$.

Case R-Inv: $e = \text{new } C(\overline{d}).m(\overline{d'}) \qquad \text{mbody}(m, C) = (\overline{x}, e_0)$
$e' = [\overline{d'}/\overline{x}, \text{new } C(\overline{d})/\texttt{this}]\, e_0$
Rule (T-Inv) requires
$\Gamma \vdash \text{new } C(\overline{d}) : C$
$\text{mtype}(m, C) = \overline{V} \to T$
$\Gamma \vdash \overline{d'} : \overline{W}$ where $\overline{W} <: \overline{V}$
With Lemma A.1.8 we get $\overline{x} : \overline{V}, \texttt{this} : D \vdash e_0 : W'$ for some $C <: D$ and $W' <: T$. According to Lemma A.1.7 this implies $\Gamma, \overline{x} : \overline{V}, \texttt{this} : D \vdash e_0 : W'$. With Lemma A.1.6 we get $\Gamma \vdash e' : T'$ with $T' <: W' <: T$.

Case R-Req: $e = e'$ `requires` C
From (T-Req) follows $T' = \overline{D} \Rightarrow \overline{D'}$ and $T = \overline{D} \cup C \Rightarrow \overline{D'} \setminus C$ for some $\overline{D}$ and $\overline{D'}$. It is now easy to show that $T' <: T$.

Case R-MixC: $e = e'$ `mixin component`
With (T-Mix) and (T-Com) we get immediately the required result $T = T' = \overline{C} \Rightarrow \overline{C'}$ for some $\overline{C}$ and $\overline{C'}$.

Case R-MixP: $e = e_0$ `mixin` $(e_1$ `provides` C `as` x `with` $d)$
$e' = (e_0$ `mixin` $e_1)$ `provides` C `as` x `with` d

We look at the derivation of $\Gamma \vdash e : T$:

$$\dfrac{\Gamma \vdash e_0 : \overline{D} \Rightarrow \overline{D'} \qquad \dfrac{\Gamma \vdash e_1 : \overline{E} \Rightarrow \overline{E'} \quad \Gamma, x : [\overline{E''}] \vdash d : B \quad B <: C}{\Gamma \vdash e_1 \texttt{ provides } C \texttt{ as } x \texttt{ with } d : (\overline{E} \cup \overline{E''}) \backslash (\overline{E'} \cup C) \Rightarrow \overline{E'} \cup C}}{\Gamma \vdash e : T}$$

where $T = (\overline{D} \cup \overline{E} \cup \overline{E''}) \backslash (\overline{D'} \cup \overline{E'} \cup C) \Rightarrow \overline{D'} \cup \overline{E'} \cup C$. Now we derive a type T' for expression e' and show that $T' = T$:

$$\dfrac{\dfrac{\Gamma \vdash e_0 : \overline{D} \Rightarrow \overline{D'} \quad \Gamma \vdash e_1 : \overline{E} \Rightarrow \overline{E'}}{\Gamma \vdash e_0 \texttt{ mixin } e_1 : ((\overline{D} \cup \overline{E}) \backslash (\overline{D'} \cup \overline{E'})) \Rightarrow \overline{D'} \cup \overline{E'}} \quad \Gamma, x : [\overline{E''}] \vdash d : B \quad B <: C}{\Gamma \vdash e' : T'}$$

where $T' = (((\overline{D} \cup \overline{E}) \backslash (\overline{D'} \cup \overline{E'})) \cup E'') \backslash (\overline{D'} \cup \overline{E'} \cup C) \Rightarrow \overline{D'} \cup \overline{E'} \cup C = T$.

Case R-MixF: The induction step is almost identical to case (R-MixP).

All the other cases are straightforward. □

A.2 Progress

Lemma A.2.1 (Object and component access) Suppose $\Gamma \vdash e : U$

1. If $e = \text{new } C(\overline{e'}).f_i$, then $\text{fields}(C) = \overline{T}\ \overline{f}$.
2. If $e = \text{new } C(\overline{e'}).m(\overline{d})$, then $\text{mbody}(m, C) = (\overline{x}, d')$ and $\#(\overline{x}) = \#(\overline{d})$.
3. If $e = \text{new } c :: C$, then $\text{service}(\text{new } c, c, C) = d$.

Proof:

1. This follows directly from (T-Fld).
2. The well-typedness of e yields $\text{mtype}(m, C) = \overline{T} \to T'$ with $\Gamma \vdash \overline{d} : \overline{V}$ and $\overline{V} <: \overline{T}$. Using this, it is easy to show that $\text{mbody}(m, C) = (\overline{x}, d')$ and $\#(\overline{x}) = \#(\overline{T}) = \#(\overline{V}) = \#(\overline{d})$.
3. By induction on the structure of c.

□

Theorem 2.3.2 (Progress) If $\vdash e : T$ then e is either a value or $e \longrightarrow e'$ for some e'.

Proof: By induction on the derivation of $\vdash e : T$. We only present the non-trivial cases where e is not a value and where congruence rules cannot be used.

Case T-Fld: $e = \text{new } C(\overline{v}).f_i \qquad T = T_i$
With Lemma A.2.1.1 we get $\text{fields}(C) = \overline{T}\,\overline{f}$. Now rule (R-Fld) yields $e' = v_i$.

Case T-Inv: $e = \text{new } C(\overline{v'}).m(\overline{v}) \qquad \vdash \overline{v} : \overline{U}$
$\vdash \text{new } C(\overline{v'}) : C \qquad \overline{U} <: \overline{T'}$
$\text{mtype}(m, C) = \overline{T'} \to T$

With Lemma A.2.1.2 we get $\text{mbody}(m, C) = (\overline{x}, \overline{d})$ and $\#(\overline{x}) = \#(\overline{v})$. With rule (R-Inv) we can now derive $e' = [\overline{v}/\overline{x}, \text{new } C(\overline{v'})/\texttt{this}]\, d$.

Case T-Serv: $e = \text{new } c :: T \qquad \vdash \text{new } c : [\overline{C}] \qquad C_i = T \in \overline{C}$
Lemma A.2.1.3 yields $\text{service}(\text{new } c, c, C_i) = d$. By looking at rule (T-Serv) we can choose $e' = d$.

Case T-Mix: $e = c_0 \texttt{ mixin } c_1 \qquad \vdash c_0 : \overline{C} \Rightarrow \overline{C'}$
$T = (\overline{C} \cup \overline{D}) \setminus (\overline{C'} \cup \overline{D'}) \Rightarrow \overline{C'} \cup \vdash c_1 : \overline{D} \Rightarrow \overline{D'}$
$\overline{D'}$

We have to distinguish three different subcases, depending on c_1 being either `component`, $c2$ `provides` C `as` x `with` d, or $c2$ `forwards` C `as` x `to` d. In all three cases, either rule (R-MixC), (R-MixP), or (R-MixF) immediately yields a corresponding e'.

Case T-Req: $e = e_0 \texttt{ requires } C$
A simple application of rule (R-Req) results in $e' = e_0$. □

Appendix B

Keris Grammar

This appendix presents a grammar for KERIS. The grammar extends the JAVA grammar from the *Java Language Specification* [82]. Only the new and modified rules are listed to keep the section short. The grammar has been mechanically checked to ensure that it is in LALR(1) form.

Top-Level Declaration

In addition to class and interface declarations, KERIS allows module and module interface definitions on the top-level.

TypeDeclaration ::= *ClassDeclaration*
| *InterfaceDeclaration*
| *ModuleDeclaration*
| *ModuleIntfDeclaration*
| ';'

Module Interface Declaration

Module interface declarations either introduce new module interfaces, or they refine or specialize existing interfaces. Module interfaces may contain constants, abstract methods, abstract or opaque class fields, class interfaces, and abstract submodule definitions.

ModuleIntfDeclaration ::= $Modifiers_{opt}$ 'module' 'interface' *Identifier* $SimpleRequires_{opt}$ *ModuleIntfBody*
| $Modifiers_{opt}$ 'module' 'interface' *Identifier* 'refines' *SimpleInterfaceTypeList* $SimpleRequires_{opt}$ *ModuleIntfBody*
| $Modifiers_{opt}$ 'module' 'interface' *Identifier* 'specializes' *SimpleInterfaceTypeList* $RequiresAs_{opt}$ *ModuleIntfBody*

SimpleRequires ::= 'requires' *SimpleInterfaceTypeList*

SimpleInterfaceTypeList ::= *SimpleQualifiedName*
| *SimpleInterfaceTypeList* ',' *SimpleQualifiedName*

ModuleIntfBody ::= '{' *ModuleIntfMemberDecls*$_{opt}$ '}'

RequiresAs ::= 'requires' *ModuleList*

ModuleList ::= *ModuleListElem*
| *ModuleList* ',' *ModuleListElem*

ModuleListElem ::= *SimpleQualifiedName*
| *SimpleQualifiedName* 'as' *SimpleQualifiedName*

ModuleIntfMemberDecls ::= *ModuleIntfMemberDecl*
| *ModuleIntfMemberDecls* *ModuleIntfMemberDecl*

ModuleIntfMemberDecl ::= *ConstantDeclaration*
| *AbstractMethodDeclaration*
| *ClassFieldHeader* ';'
| *InterfaceDeclaration*
| *AbstractSubmoduleDecl*
| *ModuleImportDeclaration*

ClassFieldHeader ::= *Modifiers*$_{opt}$ 'class' *Identifier* *Interfaces*$_{opt}$
| *Modifiers*$_{opt}$ 'class' *Identifier* *Super* *Interfaces*$_{opt}$
| *Modifiers*$_{opt}$ 'class' *Identifier* *Super* ',' *InterfaceTypeList* *Interfaces*$_{opt}$

AbstractSubmoduleDecl ::= *Modifiers*$_{opt}$ 'module' *SimpleQualifiedName* ';'
| *Modifiers*$_{opt}$ 'module' *SimpleQualifiedName* 'as' *SimpleInterfaceTypeList* ';'

ModuleImportDeclaration ::= 'import' *Name* ';'
| 'import' *Name* '.' '*' ';'

SimpleQualifiedName ::= *Identifier*
| *SimpleQualifiedName* '.' *Identifier*

Module Declaration

Module declarations either introduce new modules, or they refine or specialize existing modules. Modules may contain all possible class member declarations as well as submodule definitions, class fields, algebraic types, and module initialization blocks.

ModuleDeclaration ::= *Modifiers*$_{opt}$ 'module' *Identifier* *SimpleInterfaces*$_{opt}$ *SimpleRequires*$_{opt}$ *ModuleBody*
| *Modifiers*$_{opt}$ 'module' *Identifier* 'refines' *SimpleQualifiedName* *SimpleInterfaces*$_{opt}$ *SimpleRequires*$_{opt}$ *ModuleBody*
| *Modifiers*$_{opt}$ 'module' *Identifier* 'specializes' *SimpleQualifiedName* *SimpleInterfaces*$_{opt}$ *RequiresAs*$_{opt}$ *ModuleBody*

SimpleInterfaces ::= 'implements' *SimpleInterfaceTypeList*

ModuleBody ::= '{' ModuleBodyDeclarations$_{opt}$ '}'

ModuleBodyDeclarations ::= *ModuleBodyDeclaration*
| *ModuleBodyDeclarations* *ModuleBodyDeclaration*

ModuleBodyDeclaration ::= *SubmoduleDeclaration*
| *ModuleImportDeclaration*
| *ClassFieldDeclaration*
| *ClassMemberDeclaration*
| *AlgebraicDeclaration*
| *Block*

SubmoduleDeclaration ::= *AbstractSubmoduleDecl*
| $Modifiers_{opt}$ 'module' *SimpleQualifiedName* 'implements' *ModuleListElem* ';'

ClassFieldDeclaration ::= *ClassFieldHeader* ';'
| *ClassFieldHeader* '=' *ClassOrInterfaceType* ';'
| *ClassFieldHeader* '=' 'super' ';'
| *ClassFieldHeader* '=' *ClassBody*
| *ClassFieldHeader* '=' *ClassOrInterfaceType* *ClassBody*
| *ClassFieldHeader* '=' 'super' *ClassBody*

Interface Declaration

In addition to constants, abstract methods, classes, and interfaces, interface definitions may also specify class constructor signatures.

InterfaceMemberDeclaration ::= *ConstantDeclaration*
| *AbstractMethodDeclaration*
| *ClassDeclaration*
| *InterfaceDeclaration*
| *AbstractConstructorDecl*

AbstractConstructorDecl ::= $Modifiers_{opt}$ *ConstructorDeclarator* $Throws_{opt}$ ';'

Algebraic Datatype Declaration

In KERIS, algebraic types specify a set of constructors (also called *cases*). An algebraic type either extends a regular class, or another algebraic type.

AlgebraicDeclaration ::= $Modifiers_{opt}$ 'class' *Identifier* $Super_{opt}$ *AlgebraicCases* ';'

AlgebraicCases ::= *CaseDeclaration*
| *AlgebraicCases* ',' *CaseDeclaration*

CaseDeclaration ::= $Modifiers_{opt}$ 'case' *Identifier*
| $Modifiers_{opt}$ 'case' *Identifier* '(' $FormalParameterList_{opt}$ ')'

Types and Names

In KERIS, module qualification for submodule instances is expressed with the `::` notation.

ReferenceType	::=	*ClassOrInterfaceType*
	|	*ArrayType*
	|	*CompoundType*
CompoundType	::=	'[' *InterfaceTypeList* ']'
QualifiedName	::=	*Name* '.' *Identifier*
	|	*Name* '.' 'class'
	|	*Name* '::' *Identifier*

Appendix C

Principles of Extensible Algebraic Types

This appendix briefly reviews the type theoretic intuitions behind *extensible algebraic datatypes with defaults*. Traditionally, algebraic types are treated as sum types of variants. Classical sum types can be extended in a straightforward way by adding new variants. However, such an extension yields a subtype relation which is the reverse of the extension relation, i.e. extensions become supertypes of the original type. In the following we argue that the induced subtyping relation is not useful for writing extensible software. We show that adding *default cases* to algebraic types has the effect of reversing the original subtype relation, bringing it in sync with the extension relation.

Extensible Sums

Algebraic datatypes can be modeled as sums of variants. Each variant constitutes a new type, which is given by a tag and a tuple of component types. Consider, for instance, the following declaration:

```
class A {
  case A1(T1,1 x1,1, ..., T1,r1 x1,r1);
  case A2(T2,1 x2,1, ..., T2,r2 x2,r2);
}
```

This defines a sum type A consisting of two variant types A_1 and A_2, which have components $T_{1,1}\,x_{1,1}, \ldots, T_{1,r_1}\,x_{1,r_1}$ and $\mathtt{T}_{2,1}\,\mathtt{x}_{2,1}, \ldots, \mathtt{T}_{2,r_2}\,\mathtt{x}_{2,r_2}$, respectively. Let allcases(A) denote the set of all variants of the algebraic type A. For our example we get allcases$(A) = \{A_1, A_2\}$. To describe extensions of algebraic types, we introduce a partial order $\preceq$ between algebraic types. $B \preceq A$ holds if B extends A by adding new variants. A priori, the algebraic extension relation $\preceq$ is independent of the subtyping relation. In our setting $\preceq$ is defined explicitly by type declarations. For instance, the following code defines an algebraic datatype $B \preceq A$ that extends A with an additional variant B_1:

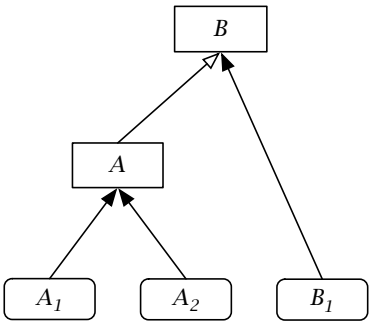


Figure C.1: Subtyping for extensible sums.

```
class B extends A {
  case B1(...);
}
```

The new type B is described by the set of its own variants $\text{owncases}(B) = \{B_1\}$ and the inherited variants $\text{allcases}(A)$. Thus, for the extended algebraic type B, we get $\text{allcases}(B) = \text{allcases}(A) \cup \text{owncases}(B) = \{A_1, A_2, B_1\}$.

The standard typing rules for sum types [46] turn A into a subtype of B if all variants of A are also variants of B. In our example, we have $\text{allcases}(A) \subseteq \text{allcases}(B)$, so the original type A is a subtype of the extended type B. Figure C.1 summarizes the relationships between types. In this figure, algebraic datatypes are depicted as boxes, variants are displayed as round boxes. Arrows highlight subtype relationships. More specifically, outlined arrows represent algebraic type extensions, whereas all other arrows connect variants with the algebraic types to which they belong.

We call this set-theoretic approach *extensible sums*. In general, this approach is characterized by the following equation:

$$\begin{aligned} \text{allcases}(Y) &= \text{inherited}(Y) \cup \text{owncases}(Y) \\ \text{where}\quad \text{owncases}(Y) &= \bigcup_i \{Y_i\} \\ \text{inherited}(Y) &= \bigcup_{Y \leq X, Y \neq X} \text{owncases}(X) \end{aligned}$$

Unfortunately, the subtype relation between extensible sum types is often the opposite of what one would like to have in practice. Imagine we have the following `Term` type:

```
class Term {
  case Number(int val);
  case Plus(Term left, Term right);
}
```

Adding a new variant `Ident` would yield a new algebraic type `ExtendedTerm`.

```
class ExtendedTerm extends Term {
  case Ident(String name);
}
```

Since `ExtendedTerm` is a supertype of `Term`, we cannot represent the sum of two identifiers with the `Plus` variant. This variant expects two `Terms` as its arguments, but the variant `Ident` is not included in the `Term` type. In other words, extensible sums do not support open recursion in the definition of a datatype. So the classical way of describing algebraic types by a fixed set of variants does not provide extensibility in the way we need it.

Extensible Algebraic Types with Defaults

To turn extended types into subtypes, we have to keep the set of variants open for every algebraic type. This is achieved by adding a default variant to every algebraic datatype, which subsumes all variants defined in future extensions of the type. The set of all variants of an extensible algebraic datatype is now given by the following equation.

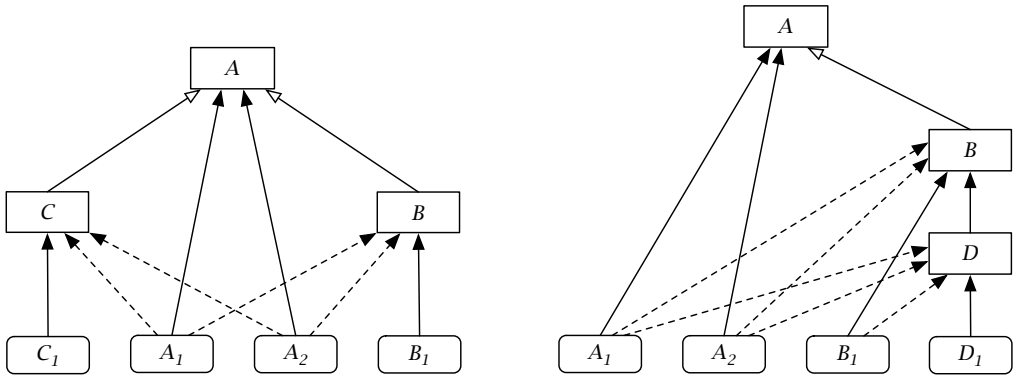
$$\begin{aligned} \text{allcases}(Y) &= \text{inherited}(Y) \cup \text{owncases}(Y) \cup \text{default}(Y) \\ \text{where}\quad \text{owncases}(Y) &= \bigcup_i \{Y_i\} \\ \text{inherited}(Y) &= \bigcup_{Y \le X, Y \neq X} \text{owncases}(X) \\ \text{default}(Y) &= \bigcup_{Z \le Y, Z \neq Y} \text{owncases}(Z) \end{aligned}$$

That is, every extensible algebraic type Y is defined by three disjoint variant sets $\text{owncases}(Y)$, $\text{inherited}(Y)$ and $\text{default}(Y)$. $\text{inherited}(Y)$ includes all inherited variants from the algebraic type Y is extending, $\text{owncases}(Y)$ denotes Y's new cases, and $\text{default}(Y)$ subsumes variants of future extensions.

With this understanding, the variant sets for types A and B from Section C now look like this: $\text{allcases}(A) = \{A_1, A_2\} \cup \text{default}(A)$, and $\text{allcases}(B) = \{A_1, A_2, B_1\} \cup \text{default}(B)$. Since $\text{default}(A)$ captures B_1 as well as $\text{default}(B)$, $\{B_1\} \cup \text{default}(B)$ is a subset of $\text{default}(A)$. Therefore $\text{allcases}(B) \subseteq \text{allcases}(A)$ and B is a subtype of A.

One might be tempted to believe now that one has even $\text{allcases}(A) = \text{allcases}(B)$. This would identify types A and B. But a closer look at the definition of default reveals that $\text{default}(B)$ only subsumes variants of extensions of B. Variants of any other extension of A are contained in $\text{default}(A)$, but not covered by $\text{default}(B)$. This is illustrated by the following algebraic class declaration:

```
class C extends A {
  case C1(...);
}
```



(a) Alternative extensions (b) Linear extensions

Figure C.2: Subtyping algebraic type extensions.

C is another extension of algebraic type A, which is completely orthogonal to B. Its case C_1 is not included in default(B), but it is an element of default(A). As a consequence, $\{B_1\} \cup$ default(B) is a proper subset of default(A), and therefore the extended type B is a proper subtype of A. C is a proper subtype of A for the same reasons, but the types B and C are incompatible.

The subtype relationships of our example are illustrated in Figure C.2a. Again, boxes represent algebraic datatypes, round boxes represent variants. Subtype relationships are depicted with arrows. Extending an algebraic type means creating a new type which is a subtype of the old one and which inherits all the variants of the old type. Furthermore, the new type may also define additional variants. Dashed arrows connect inherited variants with the algebraic type to which they get inherited.

With this approach, extended algebraic types get subtypes of the types they extend. Therefore existing functions can be applied to values of extended types. New variants are simply subsumed by the default clause of every pattern matching construct. Another interesting observation can be made when looking at two different extensions of a single algebraic type (like B and C in the example above). These types are incompatible; neither of them is a supertype of the other one. This separation of different extensions is a direct consequence of single-inheritance: An extensible algebraic type can only extend a single other algebraic datatype.

Extending the same type more than once yields extended algebraic types that share some variants, but that are incompatible to each other. Of course, it is also possible to extend an extension of an algebraic type further:

```
class D extends B {
  case D1(...);
}
```

Here, the algebraic type D extends B and defines an additional variant D_1. Figure C.2b shows the resulting subtype relations.

Figures

Listings

Bibliography

[1] M. Abadi and L. Cardelli. *A Theory of Objects*. Monographs in Computer Science. Springer Verlag, 1996.

[2] F. Achermann, M. Lumpe, J.-G. Schneider, and O. Nierstrasz. Piccola — A small composition language. In H. Bowman and J. Derrick, editors, *Formal Methods for Distributed Processing — A Survey of Object-Oriented Approaches*, pages 403–426. Cambridge University Press, 2001.

[3] F. Achermann and O. Nierstrasz. Applications = components + scripts — A tour of piccola. *Software Architectures and Component Technology*, pages 261–292, 2001.

[4] M. Aksit, L. Bergmans, and S. Vural. An object-oriented language-database integration model: The composition-filters approach. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 372–395, 1992.

[5] J. Aldrich, C. Chambers, and D. Notkin. Architectural reasoning in ArchJava. In *Proceedings of the 16th European Conference on Object-Oriented Programming*, Málaga, Spain, June 2002.

[6] J. Aldrich, C. Chambers, and D. Notkin. ArchJava: Connecting software architecture to implementation. In *International Conference on Software Engineering*, Orlando, Florida, USA, May 2002.

[7] E. Allen. Designing extensible applications. In *Diagnosing Java Code*. IBM developerWorks, 2001.

[8] R. Allen. *A Formal Approach to Software Architecture*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, May 1997.

[9] D. Ancona and E. Zucca. A theory of mixin modules: Basic and derived operators. *Mathematical Structures in Computer Science*, 8(4):401–446, 1998.

[10] D. Ancona and E. Zucca. A primitive calculus for module systems. In *Principles and Practice of Declarative Programming*, LNCS 1702. Springer-Verlag, 1999.

[11] D. Ancona and E. Zucca. True modules for Java-like languages. In *Proceedings of European Conference on Object-Oriented Programming*, LNCS 2072. Springer-Verlag, 2001.

[12] D. Ancona and E. Zucca. A calculus of module systems. *Journal of Functional Programming*, 12(2):91–132, 2002.

[13] A. Appel, L. Cardelli, K. Crary, K. Fisher, C. Gunter, R. Harper, X. Leroy, M. Lillibridge, D. B. MacQueen, J. Mitchell, G. Morrisett, J. H. Reppy, J. G. Riecke, Z. Shao, and C. A. Stone. Principles and preliminary design for ML2000, March 1999.

[14] J. Armstrong, R. Virding, C. Wikström, and M. Williams. *Concurrent Programming in Erlang*. Prentice-Hall, second edition, 1996.

[15] U. Asmann. *Invasive Software Composition*. Springer-Verlag Heidelberg, February 2003.

[16] D. Bacon, J. Bloch, J. Bogda, C. Click, P. Haahr, D. Lea, T. May, J.-W. Maessen, J. Mitchell, K. Nilsen, B. Pugh, and E. G. Sirer. The "double-checked locking is broken" declaration, 2000.
`http://www.cs.umd.edu/~pugh/java/memoryModel/DoubleCheckedLocking.html`.

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

[18] D. Batory and S. O'Malley. The design and implementation of hierarchical software systems with reusable components. *ACM TOSEM*, October 1992.

[19] L. Bauer, A. W. Appel, and E. W. Felten. Mechanisms for secure modular programming in Java. Technical Report TR-603-99, Department of Computer Science, Princeton University, 1999.

[20] K. Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, October 1999.

[21] K. Beck and M. Fowler. *Planning Extreme Programming*. Addison-Wesley, October 2000.

[22] A. Bergel, S. Ducasse, and R. Wuyts. Classboxes: A minimal module model supporting local rebinding. In *Proceedings of JMLC 2003*, Klagenfurt, Austria, August 2003.

[23] L. Bergmans and M. Aksit. Composing crosscutting concerns using composition filters. *Communications of the ACM*, 44(10):51–57, October 2001.

[24] L. Bettini, V. Bono, and B. Venneri. Coordinating mobile object-oriented code. In *Proceedings of Coordination 2002*, York, UK, April 2002.

[25] M. Blume and A. W. Appel. Hierarchical modularity. *ACM Transactions on Programming Languages and Systems*, 21(4), July 1999.

[26] V. Bono, A. Patel, and V. Shmatikov. A core calculus of classes and mixins. In *Proceedings of the 13th European Conference on Object-Oriented Programming*, pages 43–66, Lisbon, Portugal, 1999.

[27] J. Bosch and A. Ran. Evolution of software product families. In *3rd International Workshop on Software Architectures for Product Families*, LNCS 1951, pages 168–183, Las Palmas de Gran Canaria, Spain, 2000.

[28] G. Bracha. *The Programming Language Jigsaw: Mixins, Modularity and Multiple Inheritance*. PhD thesis, University of Utah, 1992.

[29] G. Bracha and W. Cook. Mixin-based inheritance. In N. Meyrowitz, editor, *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications*, pages 303–311, Ottawa, Canada, 1990. ACM Press.

[30] G. Bracha and D. Griswold. Extending Smalltalk with mixins. In *OOPSLA '96 Workshop on Extending the Smalltalk Language*, April 1996.

[31] G. Bracha and G. Lindstrom. Modularity meets inheritance. In *Proceedings of the IEEE Computer Society International Conference on Computer Languages*, pages 282–290, Washington, DC, 1992. IEEE Computer Society.

[32] G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In *Proceedings of OOPSLA '98*, October 1998.

[33] T. Bray, J. Paoli, C. Sperberg-McQueen, and E. Maler. Extensible markup language (XML) 1.0 (second edition), 2000. `http://www.w3.org/XML/`.

[34] K. B. Bruce. Safe static type checking with systems of mutually recursive classes and inheritance. Technical report, Williams College, Williamstown, MA, USA, 1997.

[35] K. B. Bruce. *Foundations of Object-Oriented Programming Languages: Types and Semantics*. MIT Press, Cambridge, Massachusetts, February 2002. ISBN 0-201-17888-5.

[36] K. B. Bruce, A. Fiech, and L. Petersen. Subtyping is not a good "Match" for object-oriented languages. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 104–127, 1997.

[37] K. B. Bruce, L. Petersen, and J. Vanderwaart. Modules in LOOM: Classes are not enough. Technical report, Williams College, Williamstown MA, USA, 1998.

[38] M. Büchi and W. Weck. Compound types for Java. In *Proceedings of OOPSLA 1998*, pages 362–373, October 1998.

[39] M. Büchi and W. Weck. Generic wrappers. In *Proceedings of the 14th European Conference on Object-Oriented Programming*, pages 201–225, June 2000.

[40] J. Buckley, T. Mens, M. Zenger, A. Rashid, and G. Kniesel. Towards a taxonomy of software change. To appear in *Journal of Software Maintenance and Evolution: Research and Practice (Special Issue on USE)*, 2004.

[41] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern–Oriented Software Architecture: A System of Patterns*. John Wiley & Sons, 1996.

[42] L. Cardelli. Typeful programming. In E. J. Neuhold and M. Paul, editors, *Formal Description of Programming Concepts, IFIP State-of-the-Art Reports*, pages 431–507, New York, 1991. Springer-Verlag.

[43] L. Cardelli. Type systems. In A. B. Tucker, editor, *The Computer Science and Engineering Handbook*, chapter 103, pages 2208–2236. CRC Press, 1997.

[44] L. Cardelli, J. Donahue, L. Glassman, M. Jordan, B. Kalsow, and G. Nelson. Modula-3 language definition. *ACM SIGPLAN Notices*, 27(8):15–42, August 1992.

[45] L. Cardelli and X. Leroy. Abstract types and the dot notation. In *Proceedings of IFIP TC2 Working Conference on Programming Concepts and Methods*, pages 479–504, North-Holland, 1990.

[46] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4):471–522, December 1985.

[47] N. Carriero and D. Gelernter. How to write parallel programs: A guide to the perplexed. *ACM Computing Surveys*, 21(3):323–357, September 1989.

[48] D. Cavin. Synchronous Java compiler. Projet de semestre. École Polytechnique Fédérale de Lausanne, Switzerland, February 2000.

[49] C. Chambers and Cecil Team. The Cecil language, specification and rationale, December 1998.

[50] C. Clifton, G. T. Leavens, C. Chambers, and T. Millstein. MultiJava: Modular open classes and symmetric multiple dispatch for Java. In *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications*, pages 130–145. ACM Press, October 2000.

[51] W. R. Cook. Object-oriented programming versus abstract data types. In J. W. de Bakker, W. P. de Roever, and G. Rozenberg, editors, *Foundations of Object-Oriented Languages, REX School/Workshop, Noordwijkerhout, The Netherlands, May/June 1990*, volume 489, pages 151–178. Springer-Verlag, New York, NY, 1991.

[52] W. R. Cook, W. L. Hill, and P. S. Canning. Inheritance is not subtyping. In *Proceedings of the 17th Annual ACM Symposium on Principles of Programming Languages*, pages 125–135, San Francisco, California, USA, January 1990.

[53] J. Corwin, D. F. Bacon, D. Grove, and C. Murthy. MJ: A rational module system for Java and its applications. In *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Anaheim, CA, USA, October 2003.

[54] K. Crary, R. Harper, and S. Puri. What is a recursive module? In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 50–63, 1999.

[55] K. Czarnecki. *Generative Programming*. Addison-Wesley, May 2000. ISBN 0-210-30977-7.

[56] F. Deremer and H. H. Kron. Programming in the large versus programming in the small. *IEEE Transactions on Software Engineering*, June 1976.

[57] D. Duggan and C. Sourelis. Mixin modules. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming*, pages 262–273, Philadelphia, Pennsylvania, June 1996.

[58] E. Ernst. *gBeta: A language with virtual attributes, block structure and propagating, dynamic inheritance*. PhD thesis, Department of Computer Science, University of Aarhus, Denmark, 1999.

[59] E. Ernst. Family polymorphism. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 303–326, Budapest, Hungary, 2001.

[60] E. Ernst. Loosely coupled class families. In *ECOOP Workshop on Advanced Separation of Concerns*, 2001.

[61] E. Ernst. Higher-order hierarchies. In *Proceedings of the European Conference on Object-Oriented Programming*, Darmstadt, Germany, July 2003.

[62] P. Eugster, R. Guerraoui, and C. Damm. On objects and events. In *Proceedings for OOPSLA 2001*, Tampa Bay, Florida, October 2001.

[63] R. B. Findler. Modular abstract interpreters. Unpublished manuscript, Carnegie Mellon University, June 1995.

[64] R. B. Findler and M. Flatt. Modular object-oriented programming with units and mixins. In *Proceedings of the ACM International Conference on Functional Programming*, volume 34(1), pages 94–104, Baltimore, Maryland, 1999.

[65] K. Fisher and J. Reppy. Foundations for Moby classes. Technical report, Bell Labs, Lucent Technologies, Murray Hill, NJ, USA, February 1999.

[66] K. Fisher and J. Reppy. Report on the Moby programming language, November 2001.

[67] K. Fisher and J. Reppy. Inheritance-based subtyping. *Information and Computation*, 177(1):28–55, August 2002.

[68] K. Fisher and J. H. Reppy. The design of a class mechanism for Moby. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 37–49, 1999.

[69] M. Flatt. PLT MzScheme: Language manual. Technical Report TR 97-280, Rice University, 1997.

[70] M. Flatt. *Programming Languages for Reusable Software Components*. PhD thesis, Rice University, Department of Computer Science, June 1999.

[71] M. Flatt and M. Felleisen. Units: Cool modules for HOT languages. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pages 236–248, 1998.

[72] M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and mixins. In *Proceedings of the 25th ACM Symposium on Principles of Programming Languages*, pages 171–183, San Diego, California, 1998.

[73] C. Fournet and G. Gonthier. The Reflexive Chemical Abstract Machine and the Join-calculus. In *Proceedings of the 23rd ACM Symposium on Principles of Programming Languages*, pages 372–385, January 1996.

[74] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[75] D. Garlan, R. Allen, and J. Ockerbloom. Exploiting style in architectural design environments. In *Proceedings of SIGSOFT '94: Foundations of Software Engineering*, pages 175–188, New Orleans, Louisiana, USA, December 1994.

[76] D. Garlan, R. Monroe, and D. Wile. ACME: An architecture description interchange language. In *Proceedings of CASCON '97*, November 1997.

[77] J. Garrigue. Programming with polymorphic variants. In *ML Workshop*, September 1998.

[78] J. Garrigue. Code reuse through polymorphic variants. In *Workshop on Foundations of Software Engineering*, Sasaguri, Japan, November 2000.

[79] N. Glew and G. Morrisett. Type-safe linking and modular assembly language. In *Conference Record of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 250–261, San Antonio, Texas, 1999.

[80] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, Reading, Ma., 1983.

[81] J. Gosling. The evolution of numerical computing in Java. Sun Microsystems Laboratories. `http://java.sun.com/people/jag/FP.html`.

[82] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Java Series, Sun Microsystems, second edition, 2000. ISBN 0-201-31008-2.

[83] O. M. Group. The Common Object Request Broker: Architecture and specification, revision 2.0, February 1997.

[84] R. Harper and M. Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *Proceedings of the 21st ACM Symposium on Principles of Programming Languages*, January 1994.

[85] W. Harrison and H. Ossher. Subject-oriented programming — A critique of pure objects. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, September 1993.

[86] A. Hejlsberg and S. Wiltamuth. C# language specification. Microsoft Corporation, 2000.

[87] S. Herrmann. Object Teams: Improving modularity for crosscutting collaborations. In *Proceedings of Net.ObjectDays*, Erfurt, Germany, 2002.

[88] T. Hirschowitz and X. Leroy. Mixin modules in a call-by-value setting. In *Proceedings of the European Symposium on Programming*, Grenoble, France, April 2002.

[89] I. Holland. Specifying reusable components using contracts. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 287–308, 1993.

[90] U. Hölzle. Integrating independently-developed components in object-oriented languages. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 36–56, 1993.

[91] S. Hudson, F. Flannery, S. Ananian, D. Wang, and A. Appel. *JavaCup User's Manual*, March 1998.

[92] G. Hultgren. Delta modules, February 2002. `http://minnow.cc.gatech.edu/squeak/2063`.

[93] R. Ibrahim. COMEL: A formal model for COM. Technical report, Queensland University of Technology, Brisbane, Australia, 1998.

[94] Y. Ichisugi and A. Tanaka. Difference-based modules: A class independent module mechanism. In *Proceedings of the European Conference on Object-Oriented Programming*, Málaga, Spain, June 2002.

[95] A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages & Applications*, volume 34(10), pages 132–146, 1999.

[96] A. Igarashi and B. C. Pierce. Foundations for virtual types. In *Proceedings of the European Conference on Object-Oriented Programming*, Lisbon, Portugal, 1999.

[97] D. Ingalls, T. Kaehler, J. Maloney, S. Wallace, and A. Kay. Back to the future: The story of Squeak, a practical Smalltalk written in itself. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 318–326, Atlanta, GA, USA, October 1997.

[98] G. S. Itzstein and D. Kearney. Applications of JoinJava. In *Proceedings of the 7th Asia-Pacific Conference on Computer Systems Architecture*, pages 37–46, Melbourne, Australia, 2002. Australian Computer Society, Inc.

[99] M. Jazayeri, A. Ran, and F. van der Linden. *Software Architecture for Product Families: Principles and Practices*. Addison-Wesley, 2000.

[100] M. P. Jones. Using parameterized signatures to express modular structure. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, St. Petersburg Beach, Florida, USA, January 1996.

[101] G. Kiczales. Aspect-oriented programming — The fun has just begun. In *Workshop on New Visions for Software Design and Productivity: Research and Applications*, Vanderbilt University, Nashville, Tennessee, December 2001.

[102] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *Proceedings of the European Conference on Object-Oriented Programming*, Budapest, Hungary, June 2001.

[103] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming*, pages 220–242, Jyväskylä, Finland, 1997.

[104] J. Kienzle and R. Guerraoui. AOP does it make sense? The case of concurrency and failures. In *Proceedings of the European Conference on Object-Oriented Programming*, Málaga, Spain, June 2002.

[105] G. Kniesel. Type-safe delegation for run-time component adaptation. In *Proceedings of the 13th European Conference on Object-Oriented Programming*, pages 351–366, Lisbon, Portugal, 1999.

[106] S. Krishnamurthi, M. Felleisen, and D. Friedman. Synthesizing object-oriented and functional design to promote re-use. In *European Conference on Object-Oriented Programming*, pages 91–113, 1998.

[107] C. W. Krueger. Software reuse. *ACM Computing Surveys*, 24(2):131–183, 1992.

[108] T. Kühne. The translator pattern — external functionality with homomorphic mappings. In R. Ege, M. Singh, and B. Meyer, editors, *The 23rd TOOLS conference USA 1997*, pages 48–62. IEEE Computer Society, 1998. 28.7-1.8, 1997, Santa Barbara, California.

[109] X. Leroy. Manifest types, modules and separate compilation. In *Proceedings of the 21st ACM Symposium on Principles of Programming Languages*, January 1994.

[110] X. Leroy. A modular module system. *Journal of Functional Programming*, 10(3):269–303, 2000.

[111] X. Leroy, D. Doligez, J. Garrigue, D. Rémy, and J. Vouillon. The Objective Caml system release 3.00, documentation and user's manual, April 2000.

[112] S. Liang, P. Hudak, and M. Jones. Monad transformers and modular interpreters. In *Symposium on Principles of Programming Languages*, pages 333–343, January 1992.

[113] K. Lieberherr and D. Lorenz. Coupling aspect-oriented and adaptive programming. Unpublished, 2002.

[114] K. J. Lieberherr. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*. PWS Publishing Company, Boston, 1996. ISBN 0-534-94602-X.

[115] K. J. Lieberherr and I. Holland. Assuring good style for object-oriented programs. *IEEE Software*, pages 38–48, September 1989.

[116] K. J. Lieberherr and I. Holland. Tools for preventive software maintenance. In *Conference on Software Maintenance*, pages 2–13, Miami, Florida, October 16-19, 1989.

[117] K. J. Lieberherr, I. Holland, and A. J. Riel. Object-oriented programming: An objective sense of style. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications,* in *Special Issue of SIGPLAN Notices*, pages 323–334, San Diego, CA, September 1988.

[118] M. Lillibridge. *Translucent Sums: A Foundation for Higher-Order Module Systems*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA, May 1997.

[119] T. Lindholm *et.al.* JSR-133: Java memory model and thread specification revision, Java Community Process, Sun Microsystems, September 2003.

[120] Y. D. Liu, R. Rinat, and S. Smith. Component assemblies and component runtimes. Technical report, Johns Hopkins University, Department of Computer Science, March 2003.

[121] D. Luckham, L. Augustin, J. Kenney, J. Vera, D. Bryan, and W. Mann. Specification and analysis of system architecture using Rapide. In *IEEE Transactions on Software Engineering*, April 1995.

[122] M. Lumpe, F. Achermann, and O. Nierstrasz. A formal language for composition. In G. Leavens and M. Sitaraman, editors, *Foundations of Component Based Systems*, pages 69–90. Cambridge University Press, 2000.

[123] D. MacQueen. Modules for Standard ML. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 198–207, New York, August 1984.

[124] O. L. Madsen and B. Møller-Pedersen. Virtual Classes: A powerful mechanism for object-oriented programming. In *Proceedings OOPSLA'89*, pages 397–406, October 1989.

[125] O. L. Madsen, B. Møller-Pedersen, and K. Nygaard. *Object-Oriented Programming in the BETA Programming Language*. Addison-Wesley, June 1993. ISBN 0-201-62430-3.

[126] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying distributed software architectures. In *Proceedings of the 5th European Software Engineering Conference*, Barcelona, Spain, September 1995.

[127] J. Manson and W. Pugh. A new approach to the semantics of multithreaded Java. Technical report, University of Maryland, College Park, U.S.A., January 2003.

[128] S. McDirmid, M. Flatt, and W. Hsieh. Jiazzi: New-age components for old-fashioned Java. In *Proceedings of the 2001 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications*, October 2001.

[129] S. McDirmid, M. Flatt, and W. C. Hsieh. Mixing COP and OOP. In *OOPSLA Workshop on Language Mechanisms for Programming Software Components*, pages 29–32. Technical Report NU-CCS-01-06, Northeastern University, Boston, MA, October 2001.

[130] N. Medvidovic and R. N. Taylor. A classification and comparison framework for software architecture description languages. In *IEEE Transactions on Software Engineering*, volume 26, pages 70–93, January 2000.

[131] T. Mens. A state-of-the-art survey on software merging. *Transactions on Software Engineering*, 28(5), May 2002.

[132] T. Mens, J. Buckley, M. Zenger, and A. Rashid. Towards a taxonomy of software evolution. In *International Workshop on Unanticipated Software Evolution*, Warsaw, Poland, April 2003.

[133] B. Meyer. *Eiffel, The Language*. Object Oriented Series. Prentice Hall, Engelwood Cliffs, 1992.

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

[135] M. Mezini and K. J. Lieberherr. Adaptive plug-and-play components for evolutionary software development. In *Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 97–116, 1998.

[136] M. Mezini and K. Ostermann. Integrating independent components with on-demand remodularization. In *In Proceedings of the 17th ACM Conference on Object-Oriented Programming*, Seattle, Washington, USA, November 2002.

[137] M. Mezini and K. Ostermann. Conquering aspects with caesar. In *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development*, pages 90–100, Boston, USA, March 2003.

[138] M. Mezini and K. Ostermann. Modules for crosscutting models. In *International Conference on Reliable Software Technologies (Ada-Europe 2003)*, Toulouse, France, June 2003.

[139] M. Mezini, L. Seiter, and K. Lieberherr. Component integration with pluggable composite adapters. *Software Architectures and Component Technology*, 2000.

[140] H. Mili, F. Mili, and A. Mili. Reusing software: Issues and research directions. *IEEE Transactions on Software Engineering*, 21(6):528–562, 1995.

[141] P. Milne and K. Walrath. Long-term persistance for JavaBeans. Technical report, Sun Microsystems, November 1999.

[142] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes — Parts I and II. *Information and Computation*, 100(1):1–77, 1992.

[143] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, May 1997.

[144] J. C. Mitchell, S. Meldal, and N. Madhav. An extension of Standard ML modules with subtyping and inheritance. In *Conference Record of the 18th ACM Symposium on Principles of Programming Languages*, pages 270–278, Orlando, Florida, January 1991.

[145] M. Moriconi, X. Quian, and A. Riemenschneider. Correct architecture refinement. In *IEEE Transactions on Software Engineering*, volume 21, April 1995.

[146] H. Mössenböck and N. Wirth. The programming language Oberon-2. *Structured Programming*, 12(4), 1991.

[147] O. Nierstrasz. Software evolution as the key to productivity. In *Proceedings Radical Innovations of Software and Systems Engineering in the Future*, Venice, Italy, October 2002.

[148] O. Nierstrasz. Contractual types. Technical Report IAM-03-004, Institut für Informatik, Universität Bern, Switzerland, August 2003.

[149] N. Nystrom, M. Clarkson, and A. Myers. Polyglot: An extensible compiler framework for Java. In *Proceedings of the 12th International Conference on Compiler Construction*, pages 138–152, Warsaw, Poland, April 2003.

[150] M. Odersky. Objects + views = components? In *Proceedings of Abstract State Machines 2000*, March 2000.

[151] M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, and M. Zenger. The Scala language specification. EPFL, Switzerland, January 2004. `http://scala.epfl.ch/`

[152] M. Odersky, V. Cremet, C. Röckl, and M. Zenger. A nominal theory of objects with dependent types. Technical report IC/2002/70, EPFL, Switzerland, September 2002.

[153] M. Odersky, V. Cremet, C. Röckl, and M. Zenger. A nominal theory of objects with dependent types. In *Proceedings of the European Conference on Object-Oriented Programming*, Darmstadt, Germany, July 2003.

[154] M. Odersky and P. Wadler. Pizza into Java: Translating theory into practice. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages*, pages 146–159, January 1997.

[155] K. Ostermann. Implementing reusable collaborations with delegation layers. In *OOPSLA Workshop on Language Mechanisms for Programming Software Components*, pages 9–14. Technical Report NU-CCS-01-06, Northeastern University, Boston, MA, October 2001.

[156] K. Ostermann. Dynamically composable collaborations with delegation layers. In *Proceedings of the 16th European Conference on Object-Oriented Programming*, Malaga, Spain, 2002.

[157] J. Palsberg and C. B. Jay. The essence of the visitor pattern. Technical Report 5, University of Technology, Sydney, 1997.

[158] D. Parnas. On the design and development of program families. *IEEE Transactions on Software Engineering*, SE-2(1):1–9, 1976.

[159] D. E. Perry and A. L. Wolf. Foundations for the study of software architecture. In *ACM SIGSOFT Software Engineering Notes*, volume 17, pages 40–52, October 1992.

[160] L. Petersen. A module system for Loom, May 1997.

[161] C. Petitpierre. A case for synchronous objects in compound-bound architectures. Unpublished. École Polytechnique Fédérale de Lausanne, 2000.

[162] S. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall, Englewood Cliffs, 1987.

[163] B. C. Pierce. Advanced module systems (A guide for the perplexed). Invited Talk at the International Conference on Functional Programming, 2000.

[164] B. C. Pierce. *Types and Programming Languages*. MIT Press, February 2002. ISBN 0-262-16209-1.

[165] F. Plasil, D. Balek, and R. Janecek. SOFA/DCUP: Architecture for component trading and dynamic updating. In *Proceedings of ICCDS '98*, Annapolis, Maryland, USA, May 1998.

[166] R. Prieto-Díaz. Status report: Software reusability. *IEEE Software*, pages 61–66, May 1993.

[167] A. Radenski. Module embedding. *International Journal on Software — Concepts and Tools*, 19(3):122 – 129, 1998.

[168] A. Radenski. Derivation of secure parallel applications by means of module embedding. In J. Gutknecht and W. Wech, editors, *Modular Programming Languages*, pages 26–37. Springer, 2000.

[169] A. Radenski. Anomaly-free component adaptation with class overriding. *Journal of Systems and Software*, 70(1), 2002.

[170] A. Radenski. The subclassing anomaly in compiler evolution. *International Journal on Information Theories and Applications*, Vol. 10, 2003.

[171] R. Rinat and S. Smith. Modular internet programming with Cells. In *Proceedings of the European Conference on Object-Oriented Programming*, Málaga, Spain, June 2002.

[172] D. Rogerson. *Inside COM: Microsoft's Component Object Model*. Microsoft Press, 1997.

[173] Y. Roudier and Y. Ichisugi. Mixin composition strategies for the modular implementation of aspect weaving — the EPP preprocessor and it's module description language. In *Aspect Oriented Programming Workshop at ICSE'98*, April 1998.

[174] A. Rüping. Modules in object-oriented systems. In *Technology of Object-Oriented Languages and Systems*, 1993.

[175] C. Russo. *Types for Modules*. PhD thesis, University of Edinburgh, 1998.

[176] C. Russo. First-class structures for Standard ML. In *Proceedings of the 9th European Symposium on Programming*, pages 336–350, Berlin, Germany, 2000.

[177] Y. Saidji. Operator overloading in java. Projet de semestre. École Polytechnique Fédérale de Lausanne, Switzerland, June 2000.

[178] D. Schmidt and T. Harrison. Double-checked locking: An optimization pattern for efficiently initializing and accessing thread-safe objects. In *Proceedings of the 3rd Annual Pattern Languages of Program Design Conference*, 1996.

[179] J.-G. Schneider and O. Nierstrasz. Components, scripts and glue. In L. Barroca, J. Hall, and P. Hall, editors, *Software Architectures — Advances and Applications*, pages 13–25. Springer-Verlag, 1999.

[180] J. C. Seco. Adding type safety to component-oriented programming. In *Proceedings of the FMOODS 2002 Student Workshop*, University of Twente, The Netherlands, March 2002.

[181] J. C. Seco. Type safe composition in .NET. In *First Microsoft Research Summer Workshop*, Cambridge, UK, 2002.

[182] J. C. Seco and L. Caires. A basic model of typed components. In *Proceedings of the 14th European Conference on Object-Oriented Programming*, pages 108–128, 2000.

[183] P. Sewell. Modules, abstract types, and distributed versioning. In *Proceedings of the 28th ACM Symposium on Principles of Programming Languages*, London, UK, January 2001.

[184] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[185] Y. Smaragdakis and D. Batory. Implementing layered designs with mixin layers. In *Proceedings of the European Conference on Object-Oriented Programming*, Brussels, Belgium, July 1998.

[186] V. C. Sreedhar. ACOEL on CORAL: A component requirement and abstraction language. In *OOPSLA Workshop on Specification and Verification of Component-Based Systems*, October 2001.

[187] V. C. Sreedhar. Programming software components using ACOEL. Unpublished manuscript, IBM T.J. Watson Research Center, 2002.

[188] P. Steyaert, C. Lucas, K. Mens, and T. D'Hondt. Reuse contracts: Managing the evolution of reusable assets. In *Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 268–285. ACM Press, October 1996.

[189] B. Stroustrup. *The C++ Programming Language (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1991.

[190] Sun Microsystems. JavaBeans™. `http://java.sun.com/beans`, December 1996.

[191] Sun Microsystems. Java 2 Platform Enterprise Edition Specification, 2001.

[192] C. Szyperski. Import is not inheritance — Why we need both: Modules and classes. In *Proceedings of the 4th European Symposium on Programming*, Rennes, France, February 1992.

[193] C. Szyperski. Component-oriented programming: A refined variation of object-oriented programming. *The Oberon Tribune*, 1(2), December 1995.

[194] C. Szyperski. Independently extensible systems – software engineering potential and challenges. In *Proceedings of the 19th Australian Computer Science Conference*, Melbourne, Australia, 1996.

[195] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison Wesley / ACM Press, New York, 1998. ISBN 0-201-17888-5.

[196] S. T. Taft and R. A. Duff. *Ada 95 Reference Manual: Language and Standard Libraries*. Lecture Notes in Computer Science. Springer Verlag, 1997. ISBN 3-540-63144-5.

[197] W. Taha. *Multi-Stage Programming: Its Theory and Applications*. PhD thesis, CSE-99-TH-002, 1999.

[198] P. Tarr, H. Ossher, W. Harrison, and S. Sutton. *N* degrees of separation: Multi-dimensional separation of concerns. In *Proceedings of the International Conference on Software Engineering*, May 1999.

[199] K. K. Thorup. Genericity in Java with virtual types. In *Proceedings of the European Conference on Object-Oriented Programming*, LNCS 1241, pages 444–471, June 1997.

[200] M. Torgersen. Virtual types are statically safe. In *5th Workshop on Foundations of Object-Oriented Languages*, San Diego, CA, USA, January 1998.

[201] D. Ungar and R. B. Smith. Self: The power of simplicity. *Lisp and Symbolic Computation*, March 1991.

[202] M. VanHilst and D. Notkin. Using role components to implement collaboration-based designs. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 1996.

[203] P. Wadler and et al. The expression problem. Discussion on the Java-Genericity mailing list, December 1998.

[204] S. Weerawarana, F. Curbera, and M. Duftler. Bean Markup Language: A composition language for JavaBeans components. In *Proceedings of the 6th USENIX*

Conference on Object-Oriented Technologies and Systems, San Antonio, Texas, USA, January 2001.

[205] D. Weiss and C. Lai. *Software Product-Line Engineering*. Addison-Wesley, 1999.

[206] N. Wirth. *Programming in Modula-2*. Springer Verlag, Berlin, 1982.

[207] N. Wirth, H. Mössenböck, and B. H. et. al. Component Pascal language report. Technical report, Oberon Microsystems, Inc. Oberon microsystems, Inc., May 1997.

[208] A. Wittmann. Towards Caesar: Family polymorphism for Java. Master's thesis, Technische Universität Darmstadt, Fachbereich Informatik, 2003.

[209] A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115, 1994.

[210] M. Zenger. Erweiterbare Übersetzer. Master's thesis, University of Karlsruhe, August 1998.

[211] M. Zenger. Evolving software with extensible modules. In *International Workshop on Unanticipated Software Evolution*, Málaga, Spain, 2002.

[212] M. Zenger. Type-safe prototype-based component evolution. In *Proceedings of the European Conference on Object-Oriented Programming*, Málaga, Spain, June 2002.

[213] M. Zenger. Keris: Evolving software with extensible modules. To appear in *Journal of Software Maintenance and Evolution: Research and Practice (Special Issue on USE)*, 2004.

[214] M. Zenger and M. Odersky. Extensible algebraic datatypes with defaults. In *Proceedings of the International Conference on Functional Programming*, Firenze, Italy, September 2001.

[215] M. Zenger and M. Odersky. Implementing extensible compilers. In *ECOOP Workshop on Multiparadigm Programming with Object-Oriented Languages*, Budapest, Hungary, June 2001.

[216] S. Zermatten. Compound Types in Java. Projet de semestre. Laboratoire des méthodes de programmation, École Polytechnique Fédérale de Lausanne, Switzerland. `http://lamp.epfl.ch/jaco/cjava.html`, June 2000.

Index

Curriculum Vitae

Matthias Zenger

Education and Work

I was born on February 8, 1973, in Miltenberg, Germany. In Miltenberg I attended Johannes-Butzbach-Gymansium, a high-school from which I graduated in July 1992, receiving the german equivalent to A-levels (Abitur). In fall 1993, I started studying computer science at the University of Karlsruhe, Germany. In 1997, I visited the University of South Australia to perform my diploma project under the supervision of Prof. Martin Odersky. In fall 1998, I obtained a degree in computer science (Dipl.inform.) from the University of Karlsruhe. After another 6 months time spent at the University of South Australia as a visitor researcher, I joined the Programming Methods Laboratory of the Swiss Federal Institute of Technology, Lausanne, working as a research assistant and Ph.D. student in the area of programming language design and implementation. In the last 5 years, I was also involved in the development of commercial software for Borland Software Corp. and ChoiceMaker Technologies, Inc.

Refereed Publications

1. J. Buckley, T. Mens, M. Zenger, A. Rashid, and G. Kniesel. Towards a taxonomy of software change. To appear in *Journal of Software Maintenance and Evolution: Research and Practice (Special Issue on USE)*, 2004.
2. M. Zenger. Keris: Evolving software with extensible modules. To appear in *Journal of Software Maintenance and Evolution: Research and Practice (Special Issue on USE)*, 2004.
3. M. Odersky, V. Cremet, C. Röckl, and M. Zenger. A nominal theory of objects with dependent types. In *Proceedings of the European Conference on Object-Oriented Programming*, Darmstadt, Germany, July 2003.
4. T. Mens, J. Buckley, M. Zenger, and A. Rashid. Towards a taxonomy of software evolution. In *International Workshop on Unanticipated Software Evolution*, Warsaw, Poland, April 2003.
5. M. Odersky, V. Cremet, C. Röckl, and M. Zenger. A nominal theory of objects with dependent types. In *Foundations of Object-Oriented Languages (FOOL 10)*, New Orleans, USA, January 2003.

6. M. Zenger. Type-safe prototype-based component evolution. In *Proceedings of the European Conference on Object-Oriented Programming*, Málaga, Spain, June 2002.

7. M. Zenger. Evolving software with extensible modules. In *International Workshop on Unanticipated Software Evolution*, Málaga, Spain, 2002.

8. M. Zenger and M. Odersky. Extensible algebraic datatypes with defaults. In *Proceedings of the International Conference on Functional Programming*, Firenze, Italy, September 2001.

9. M. Zenger and M. Odersky. Implementing extensible compilers. In *ECOOP Workshop on Multiparadigm Programming with Object-Oriented Languages*, Budapest, Hungary, June 2001.

10. M. Odersky, C. Zenger, and M. Zenger. Colored local type inference. In *Proceedings of the 28th ACM Symposium on Principles of Programming Languages*, pages 41–53, London, UK, January 2001.

11. M. Philippsen, M. Zenger, and M. Jacob. JavaParty — portables, paralleles und verteiltes Programmieren in Java. In C. Cap, editor, *Java-Informations-Tage 1998, Serie Informatik-Aktuell*, pages 22–38, Frankfurt, Germany, November 1998.

12. M. Philippsen and M. Zenger. JavaParty — transparent remote objects in Java. *Concurrency: Practice and experience*, 9(11):1225–1242, November 1997.