Fakultät für Informatik
der Technischen Universität München

Flexible Construction of Software Components: A Feature-Oriented Approach

Dr. Christian Prehofer

Mai 1999

Vollständiger Abdruck der Habilitationsschrift zur Vorlage an der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

habilitierten Doktors (Dr. rer. nat. habil.)

Abstract

We present a novel approach for designing software components in a flexible way by a new structuring method, for which we develop specification and programming language concepts. We propose to construct components (or objects) from a set of small services, called features. The main point is that features can be described in isolation, but their core functionality usually has to be extended or adapted in the presence of other features. This is known, e.g. in telecommunications, as the feature interaction problem. We describe these interactions or dependencies between the individual features separately in so-called interaction specifications. On this basis, we propose feature-oriented design as a generalization of object-oriented concepts. With features, we can avoid highly entangled class hierarchies and instead define features with well defined dependencies. We show that a wide range of concepts, techniques and technical results can be generalized to this new paradigm. For specification and programming with features, we cover several concepts of object-oriented languages, such as subtypes, parametric features and virtual functions. We introduce generic features which affect others in a schematic way. Furthermore, we consider features which add exception handling in a flexible way.

For the modular specification of complex components, we outline a methodology based on decomposition into feature and interaction specifications. In order to reason from the feature specifications about the behavior of composed objects, we develop refinement concepts. These lift properties of features to feature combinations. For the feature interactions, we show that a certain class of conservative interaction handlers preserves feature specifications, which addresses the issue of semantic subtyping in our setting. We first focus on the specification of a single component and then proceed to object networks with references and exceptions.

On the programming level, we use feature interaction handlers for two features at a time. These enable the automatic generation of customized components. For a set of features, an exponential number of different feature combinations can be handled by a quadratic number of interaction resolutions. This is presented as an extension of the language Java. The generalization of object-oriented concepts is shown by two translations of our concepts into aggregation and inheritance, respectively. Our flexible programming model is complemented by expressive typing concepts including parametric features and type relations between features.

Acknowledgments

The main part of this work was carried your in the fruitful environment in the group of M. Broy and T. Nipkow. The author is grateful to M. Broy for continuos support and encouragement, particuarly in the early phases of this work. Many others, in particular A. Poetzsch-Hefter, H. Hußmann, T. Nipkow, M. Odersky, K. Lieberherr, W. Naraschewski, B. Rumpe, P. Müller, I. Krüger, and C. Klein, have contributed by discussions, insights and ideas.

Contents

Chapter 1

Overview

It is well known that developing reliable software efficiently is a challenging problem, which is — in current practice — rarely solved in a convincing manner. There are many reports on failing software projects and an ongoing discussion on the so-called software crisis [WG94]. The main reason is that the complexity of a software product is generally underestimated. Furthermore, software has to be adapted or extended later on. In many cases, the maintenance of a software product over the full life cycle accounts for more than fifty percent of the total cost.

Apart from several other important factors for successful program development, the internal structure of the software system is a key success factor. The large-scale design of a software system is called its software architecture. The architecture of a system describes the key components of a system and their relations. The design of the individual components strongly depends on the choice of the programming and component or module concepts.

Since software development is costly and time-consuming, there are many approaches on reusing software. Although the idea to develop generic, reusable software is not new, there are many technical and non-technical obstacles. In many cases, software is written again from scratch, since existing software is difficult to understand and usually does not fully match the current needs.

Several recent design concepts and methodologies, centered around object-oriented programming, have raised new hope for more extensive software reuse. First of all, it is claimed that the language concepts of object-oriented languages, such as inheritance and subtyping, are the key for reusing and extending software. Furthermore, so-called application frameworks have been proposed to capture the essential design of particular application domains. Frameworks provide the reusable skeleton of an application domain and have to be completed and/or adapted for each individual application.

However, these design and reuse technologies are insufficient in many respects. In many cases, the ambitious use of object-oriented concepts to build highly flexible and generic software led to systems of unmanageable complexity. This is caused by highly entangled system components which are difficult to use, to maintain, and to extend.

In this thesis, we present a novel approach for designing software components in a

more flexible way by new structuring concepts. We propose to construct components (or objects) from a set of small services, called features. The main point is that features can be described in isolation, but their core functionality usually has to be extended or adapted in the presence of other features. This adaption can be due to conflicting functionality or can be due to a required cooperation. We describe these interactions or dependencies between the individual features separately in so-called interaction specifications.

On the programming level, we use adaptors for feature interaction resolutions for two features at a time. These enable the automatic generation of customized components, which only include the needed services and not the full overhead of a generic application. For modular specification of features, we can compose the feature plus the interaction specifications to obtain a specification of an object with a particular feature selection.

Our new structuring concepts generalize object-oriented concepts such as inheritance and aggregation. Features, similar to subclasses, add functionality, but are not bound to a class hierachy. Hence we can avoid highly entangled class hierachies and instead define features with well defined dependencies. With our expressive feature composition method, we can conveniently use and reuse features. Similarly, our concepts for feature specification apply to object-oriented specifications as well and utilize the benefits of feature-oriented structuring.

This idea of features was strongly motivated by problems in the development of telecommunication systems, which recently received much attention [BV94, CO95, Din97]. In telecommunications, the term feature is often used to describe the growing number of small, individual services provided by telephone and network switching systems. In particular, the problem of interactions between the services turned out to be a crucial problem.

On the other hand, research in theoretical computer science has led to both simple and powerful techniques for structuring the language features of programming languages [LHJ95]. These employ abstract concepts of monads [Mog91] and category theory [Pie91]. The contribution here is to select some of these theoretical concepts and use them to develop a formalism for feature specifications. Furthermore, this can be used successfully to enhance and advance object-oriented programming. The theoretical underpinnings are visible on the specification side, but not on the applied programming side.

In the following, we give an overview of the existing concepts, followed by the main contributions of this thesis.

Object-Oriented Programming

Object-oriented programming was introduced about thirty years ago. In the last decade, it has received considerable attention in the commercial software market. It is widely argued that modeling a software system as a set of interacting objects is natural in many applications. Object-oriented programming languages encourage a

different software design, which supports particular cases of software reuse. The main contributions of object-orientation are the following language concepts:

- Encapsulation of state (instance variables) and functionality (methods) into objects. Objects are created as instances of classes, which define the type of the state and the functionality of its member objects.
- Subclassing with inheritance allows one to create new classes from existing ones. A subclass can extend the used state and functionality, as well as redefine functionality.
- Subclass or subtype polymorphism permits to use objects of a subclass in place of objects of the superclass. With this kind of polymorphism, code written for a certain class can be reused for derived subclasses.

The last two items are the central techniques for reuse in object-oriented languages. It has been claimed that these are sufficient for developing adaptable and reusable systems. As all classes are built incrementally via subclassing and inheritance in object-oriented languages, this often leads to highly entangled inheritance hierarchies which hamper both development and reuse. As examined in [BBM96], deep inheritance trees are statistically correlated with a higher error probability. These problems are particularly pronounced in large and complex applications, such as frameworks, which is discussed below.

Software Components and Frameworks

A current trend in commercial software is componentware, which aims for composing smaller, reusable software components to larger ones. The idea of software components is to provide generic software pieces for recurring tasks, which can be reused in a simple fashion in several applications. Components are often identified with objects which have a non-trivial internal structure. Hence components may also have internal, local objects, but to the outside, they appear as a single object.

In industrial applications, componentware is largely successful by standardizing component platforms, architecture and interfaces, as for instance demonstrated by CORBA [Gro94] and OLE/COM [Bro95]. Although the idea of simple plug-and-play with software components is tempting, it is a bit simplistic. Even when syntactic interfaces are compatible, a typical problem is that interactions or conflicts occur between individual components. Often, one needs variations of components, for instance regarding the functionality, the user interface, or error handling techniques.

The same goal of reusing prefabricated software also motivated the idea of application frameworks [JF88, Lew95]. These are generic applications for particular application domains. Frameworks can be viewed as program libraries which include a particular, but possibly incomplete, design for the full application, to be customized by subclassing and inheritance. In contrast to components, the internals of a framework

are known to the application developer (at least for so-called white-box frameworks.) Similar to components, some kinds of frameworks, called black-box frameworks, can only be adapted at designated locations.

However, the goal of integrating all possible variations and extensions into one system significantly complicates the resulting system. This leads to systems which are hard to learn, to use, and to maintain [Tal97]. As stated in [Gam96], the dependencies between classes are difficult to control if the number of classes in a framework grows.

Software Architecture

Software architecture [SG96, GS92, HHK+96] is the large-scale structure of a software system. Although there exist several views on software architecture, typical architectural components are modules, classes or objects and their relations. The latter usually include the logical dependencies, although sometimes the physical structure of deployment is covered as well.

Current research focuses on techniques for describing architectures and on developing reference architectures for particular domains. Typical relations are module or function usage, synchronous or asynchronous communication. This gives a static abstraction of the data- or control flow. Except for module structure, it does however not present an abstraction of the code structure and its dependencies.

Our contribution can be seen as a particular small-scale software architecture for describing and composing small services, called features, to components.

Design Patterns

Design patterns [GHJV94] have received enormous attention in the last years. The central idea of patterns is to capture software design which is proven to work in several existing systems. Certain methodologies have been developed to describe patterns. It is important to stress that patterns are intended to reflect only an implementation idea, not some specific code. In this sense, patterns only present an "informal abstraction". This has the advantage that these are usually more accessible than other formal descriptions. In turn, they lack precision and it is not possible to generate code from the descriptions or to reason formally about patterns.

Design patterns exist on several levels of abstractions. For instance, there are patterns, e.g. a layered software structure [BMR+96] , which can be viewed as architectural patterns. Others, sometimes language specific ones, give particular solutions to small and well defined problems.

Although design patterns are a valuable contribution to software engineering, they are only helpful on the conceptual level. The reason is that they are not intended to be used like a program library. In contrast to this, we aim for general purpose design principles which are useful as specification and programming language constructs.

Functional Programming

Compared to object-oriented programming, functional programming provides other kinds of useful techniques for writing abstract, reusable programs. In particular, functional programming languages [MTH90, PHA+96] support parametric polymorphism and functional abstraction. This allows one to write programs with minimal assumptions about the used data or program structures, hence encourage one to write reusable software. Therefore, functional languages are well suited for building new functionality on top of existing ones. They enable a concise and clear programming style, which slowly gains acceptance for industrial applications. For instance, the language Erlang [AWV93] is used for concurrent programming in telecommunications.

Our contribution here is to integrate object- and feature-oriented concepts with functional languages. This is possible with concepts for integrating imperative programming [Wad92] and extensions of object-oriented languages [OW97].

Feature-Oriented Design

The main contribution of feature-oriented design is to structure objects or components into small services, called features. The idea is to develop the features independently and to specify, in addition, their relations for composition (interaction or cooperation) separately. Compared to object-oriented programming, inheritance trees with highly entangled classes can be avoided. Similar dependencies occur with aggregation of objects into an encapsulating class. We show that feature-oriented structuring generalizes these techniques and fosters clear and reusable design.

Our design techniques support different specification styles, high-level functional prototypes, and imperative implementations. These three steps form the three main parts of the work. Our focus is on developing practical specification and programming language concepts which are oriented towards existing languages.

We demonstrate that feature-oriented design is useful for the formal specification of object behavior. Our techniques allow one to specify features and their interactions on different levels of abstraction. These individual specifications are then used to reason about the behavior of a composed object. Furthermore, we discuss when feature compositions (similar to subclassing) are behavioral refinements of the individual specifications. A refinement relation is highly desirable in order to guarantee the individual specifications for the composed object as well.

Furthermore, we consider features which schematically affect other features, e.g. a feature which locks or disables all other features. As we formalize feature compositions in an abstract way, it is possible to specify this in our setting. Another example is an undo feature, which revokes the effect of other features.

The technical novelty of the specification approach is that we use implicit state in the form of monads for specification. This allows for abstract specifications and eases refinement. It also yields a fully functional model of an object-oriented language with references and virtual functions. In earlier works [JW93, Wad93], monads are used to

write easily modifiable code with stateful features. We go the step beyond and write easy to compose components. In other words, we make the possible "modifications" explicit as features.

On the programming level, feature-oriented programming is a generalization of conventional object-oriented programming, which generalizes subclassing and inheritance. Note that we do not have a notion of a (sub-)class, since we only compose concrete objects from features. On the programming level, the notion of a feature is similar to a class, but without any fixed class hierarchy and without inheritance As we force separation of concerns by the proposed programming language concepts, we can apply flexible composition concepts. These allow the automatic creation of objects with an individual selection of features or services, a main contribution of this work.

Compared to object-oriented techniques, feature-oriented design is advantageous for the following reasons:

- It yields more flexibility, as objects with individual services can be composed from a set of features. This is clearly desirable if different variations of one software component are needed or if new functionality has to be incorporated.
- As the core functionality is separated from interaction handling, it provides more structure and clarifies dependencies between features. Hence it encourages to write independent, reusable code. In many cases subclasses in object-oriented programs should be independent entities, and not subclasses in a fixed hierachy.
- As we consider only interactions between two features at a time, the programming model is as simple as possible. Dependencies between several features can usually be reduced to pair-wise interactions between features.

Compared to object-oriented programming, where classes of objects are developed in an incremental manner, we just compose objects from a set of features, which replace classes. This approach was motivated by the recent interest in feature interactions in telecommunications. The crucial point is that some features may interact and have to be adapted in the presence of each other. This idea will be used for a novel approach to object-oriented programming.

We present feature-oriented programming in two different language settings. An exposition of feature-oriented programming as an extension of an imperative language, namely Java, is shown in Chapter 6. This also includes a detailed comparison to object-oriented programming. Feature-oriented programming in a functional language with monadic state is presented in Chapter 5.

How To Read This Thesis

In the spirit of feature-oriented design, the three parts of this thesis can be read independently. Clearly, the interactions between them are stated explicitly. Only Chapters 3 and 4 build upon each other. Chapter 5 and Chapter 6 present feature-oriented

programming in a functional and, respectively, an object-oriented language. The two chapters contain comparable programming techniques, but in different settings and with different emphasis. Chapter 5 is mainly intended to support the earlier two chapters and uses advanced functional programming concepts. Chapter 6 extends common object-oriented languages and includes further discussions and more examples. There is some intended overlap between Chapters 3 and 5, as well as both Chapters 5 and 6.

Several parts of this thesis have been published at conferences on (object-oriented) programming [Pre97a, Pre97b, Pre97c] and telecommunications [Pre97d].

Chapter 2

Introduction

In this chapter, we introduce the main ideas and contributions of this work. The basic concepts are introduced in the next two sections. The following sections present the main topics, feature specification and programming concepts. We cover these here in reverse order as in the following main chapters. The reason is that the order is not essential, as the specification and programming parts can be read independently

2.1 Features and Feature Interactions

Features are individual services offered by objects. That is, objects consist of a set of features which partition their functionality. A feature has a syntactic interface, an intended semantic behavior (a specification), and one or more implementations. Some features are only sensible in a particular system, others are self contained and meaningful in isolation.

While individual features are often easy to understand, their combination, particularly if their number is large, often creates new problems. This is called feature interaction and is defined in telecommunications research as follows (taken from [KV95]):

> *A feature interaction occurs when the behavior of one feature is affected by the behavior of another feature [or another instance of the same feature].*

The conceptual framework developed in this application domain serves as the starting point for our design method which generalizes object-oriented programming.

In the area of telecommunication and multimedia service development, the problem of feature interactions has received considerable attention [BDC+89, Zav93, BV94, CO95, Din97]. In this application area, the problem of feature interactions stems from the abundance of features telephones and switching systems have. For instance, consider the following telephone features:

- ICS (incoming call screening)
- Forwarding of calls

Although both features are simple when considered in isolation, there is a conflict: Assume B forwards calls to his phone to C. C screens calls from A (ICS, incoming call screening). Should a call from A to B be screened by C? For many other examples we refer to [CGN+94]. From the abundance of features and interactions in telecommunication systems, we can identify two typical classes of interactions:

- Resource conflicts or priorities between features. This is addressed in our approach by composing features in layers, where the outer layers have precedence over the inner ones.
- Extension of functionality is required due to the presence of other features. In object-oriented implementations, these extended requirements are resolved via method redefinitions.

A similar problem of flexible services occurs in groupware applications, as shown in [Tee97]. For feature interactions in a larger application context see [Rya97]. Although most examples cover features which are relevant to the user, some of them may also deal with internals like resource management or communication with internal components.

While many research efforts in telecommunications focus on detecting interactions in this particular domain, we aim here at more general design principles. The main contribution of this work is to show that designing complex software components in terms of features improves existing techniques, from both practical and technical sides. In particular, we claim that feature-oriented design is preferable over incremental development with usual object-oriented techniques like the inheritance relation between classes.

In general, feature interactions can occur between a set of objects. The main idea for making this approach practical is to consider only interactions between two features at a time. In case some interaction only occurs with more than two features, it is usually possible to redesign the feature combination. For instance, introducing auxiliary features can reduce dependencies. Hence we argue that this assumption suffices in most cases and is the key to use the ideas as programming language concepts. For general purpose languages, one has to balance expressiveness with conceptual simplicity and often efficiency.

2.2 Feature-Oriented Design

We discuss in the following the main concepts and the general issues of feature-oriented design. Our contributions cover the formal specification of features as well as programming language concepts for implementing features. Although the latter can be used without the former, both complement each other.

For the specification of features, we separate feature specifications from the specification of their interactions. For the composition of several feature implementations,

we want that the individual specification and interaction specifications hold. A typical problem is that specifications are overly specific, which can lead to unintended and unnecessary inconsistencies with other feature specifications. For this purpose, we develop specification styles which avoid overspecification. As another solution, abstraction concepts help to establish an individual specification for a feature combination in a refined way. These abstraction concepts have to be in line with the programming-level feature and feature combination concepts. The reason is that abstraction and refinement concepts often take the computational structure of the feature implementation and combination into account. For instance, a typical abstraction to establish properties of a composed object is to disregard the state of all other feature implementations. Another example are exceptions, as discussed in the last section.

On the programming level, we propose a feature combination technique which allows one to construct objects from individual features in a fully flexible and modular way. Its main advantage is that objects with individual services can be created just by selecting the desired features, unlike object-oriented programming. A feature implementation is similar to an abstract class and consists of a base implementation which is characterized as follows:

- It adds functionality to an object (new methods).
- It may add local state to the object (new instance variables).
- It may assume that the extended object provides other features (similar to importing modules).

The main difference to classes in object-oriented languages is that we separate the core functionality of a subclass from overriding methods of the superclass. We view overriding more generally as a mechanism to resolve dependencies or interactions between features, i.e. some feature must behave differently in the presence of another one. For this purpose, we need to provide lifters, which adapt a feature to the context of another feature by overriding methods. This leads to a new view of inheritance, as feature interactions are resolved between two features at a time. In contrast, inheritance may override the methods of all superclasses.

In the following, we first introduce the novel programming concepts, followed by the specification techniques. Depending on the context, we often refer to a feature specification or feature implementation simply by the word feature.

2.3 Feature-Oriented Programming

We introduce feature-oriented programming in the following as a generalization of object-oriented programming. A major contribution of object-oriented programming is reuse by inheritance or subclassing. Its success and its extensive use have led to several approaches to increase flexibility (mix-ins [SCD+93, BC90], around-messages

in Lisp [LM91], class refactoring methods [OJ90]) and to approaches using different composition techniques, such as aggregation and (abstract) subclasses.

Our new model supports many of the above concepts as well as generalizes object-oriented concepts such as inheritance and and aggregation. In addition, it includes many of the above mentioned extensions and new concepts. Instead of a rigid class structure, we write features which are composed appropriately when creating objects. The main difference is that we separate the core functionality of a subclass from overriding methods of the superclass. We view overriding more generally as a mechanism to resolve dependencies or interactions between features, i.e. some feature must behave differently in the presence of another one. Without addressing interactions, the idea of structuring objects into small, coherent parts was suggested in [LA94] and developed further for reasoning about programs in [SG95].

We resolve feature interactions by lifting functions of one feature to the context of the other. A lifter, also called adaptor or modifier, adapts the functions of one feature in the presence of another one. Similar to inheritance, this is accomplished by method overriding, but lifters depend on two features and are separate entities used for composition. In contrast, inheritance just overrides methods of the superclasses.

Our new model allows one to create objects with individual services just by selecting the desired features, unlike object-oriented programming. Hence feature-oriented programming is particularly useful in applications where a large variety of similar objects is needed. Examples are generic frameworks or libraries which can be used in different applications in various ways or in application domains which by nature require many variations. For the former, we present a small example of a generic data structure; for a larger example we refer to [BSST93]. The area of telecommunication software is by now a prominent example of a rapidly evolving application domain which sparked the research of feature interactions. An example from this area is shown in Section 6.6.3. The main novelty of this approach is a modular architecture for composing features with the required interaction handling, yielding a full object.

To demonstrate our specification techniques, we use a small running example modeling simple data structures with the following features:

Stack, providing push and pop operations on a stack

Queue, providing enqueue and dequeue operations on a queue.

Counter, which adds a local counter (used for the size of a data structure).

Bound, constraining the elements of a data structure to a certain size

Undo, adding an undo function, which restores the state as it was before the last access to the object.

Lock, disallowing operations on a object

Clearly, these features are not independent. For instance, when adding the counter, the functions push and pop must, in addition, increment or decrement the counter. With traditional inheritance, this is achieved by overriding methods and by possibly calling the method of the superclass. This is done here via lifters.

Note that the above informal feature specifications are not precise and in particular do not clarify their interactions. For instance, the counter is used to count the number of stack elements. We view this dependency as an interaction which we will specify appropriately. Many other interactions exist, particularly for undo and lock, which will be considered in later chapters.

On the programming level, it is possible to create a stack or a queue with a customized selection of the other features. For instance, one can create objects with the following features:

- Stack with Counter and Undo. In this combination, the counter is used to maintain the number of stack elements. The undo functionality has to undo not only the stack operations, but the added counter as well.

- Queue with Undo and Lock. For this combination, one has to decide if undo has precedence over lock or vice versa. This means that either undo can revoke the locking, or locking disables undo.

- Stack with Counter, Bound and Undo. This combination has interactions between Bound and Undo, since the bound feature disables inserting elements which are too big. In case an element is not added to the stack due to the size check, what happens if this is followed by an undo operation? Shall we undo an "empty" operation?

The program language techniques required for such flexible combinations will be discussed in Chapters 5 and 6.

Note that other features which occur in existing data structure libraries are quite similar in nature. Clearly, all the above features can be used on other data structures, like sequences, bags or maps. For instance, in the Java [GJS96] class library similar variations of data structures with hash functions, orderings and enumerators can be found.

In an object-oriented language, one would extend a class of stacks by a counter and proceed similarly with the other features. Usually, a concrete class is added onto another concrete class. We generalize this to independent features which can be added to any object. For instance, we can create a counter object with or without lock. Furthermore, it is easy to imagine variations of the features, for instance different counters or a lock which not even permits read access. With our approach, we show that it is easy to provide such a set of features with interaction handling for simple reuse.

In general, an exponential number of different combinations is possible. In case of specifications, we similarly conjoin specifications for features and their interactions to

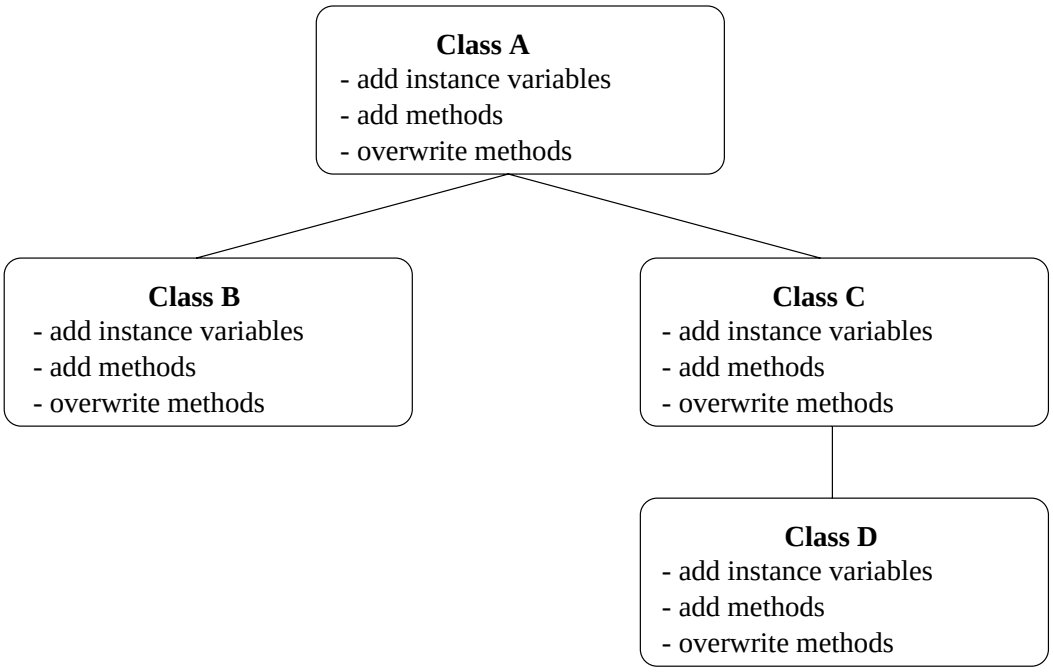

Figure 2.1: Typical Class Hierarchies

customized combinations. For combining specifications, we need to refine some feature specifications in order to account for a more specialized setting.

With feature-oriented programming, we need a method to compose implementations for features and interaction handling. In our model, a feature repository replaces the rigid structure of conventional class hierarchies. Both are illustrated in Figures 2.1 and 2.2. The composition of features in Figure 2.2 uses an architecture for adding interaction resolution code, which is similar to constructing a concrete class hierarchy. To construct an object, features are added one after another in a particular order. If a feature is added to a combination of n features, we have to apply n lifters in order to adapt the inner features. As we consider interactions of two features at a time, there is only a quadratic number $\binom{n}{2} = \frac{n^2-n}{2}$ of lifters, but an exponential number $\binom{n}{k}, k = 1, \ldots, n$ of different feature combinations can be created. For instance, in the above example, we have 5 features with 10 interactions and about 30 sensible feature combinations. This number grows if different implementations or variations of features are considered (e.g. single- or multi-step undo). The observation that most interactions can be reduced to interactions between two features at a time is a major premise of this approach. This is discussed in more detail in Section 6.8.

We show that feature-oriented programming generalizes object-oriented techniques and gives a new conceptual model of objects and object composition. To support this, we will show how to create Java [GJS96] code for concrete feature selections, first using

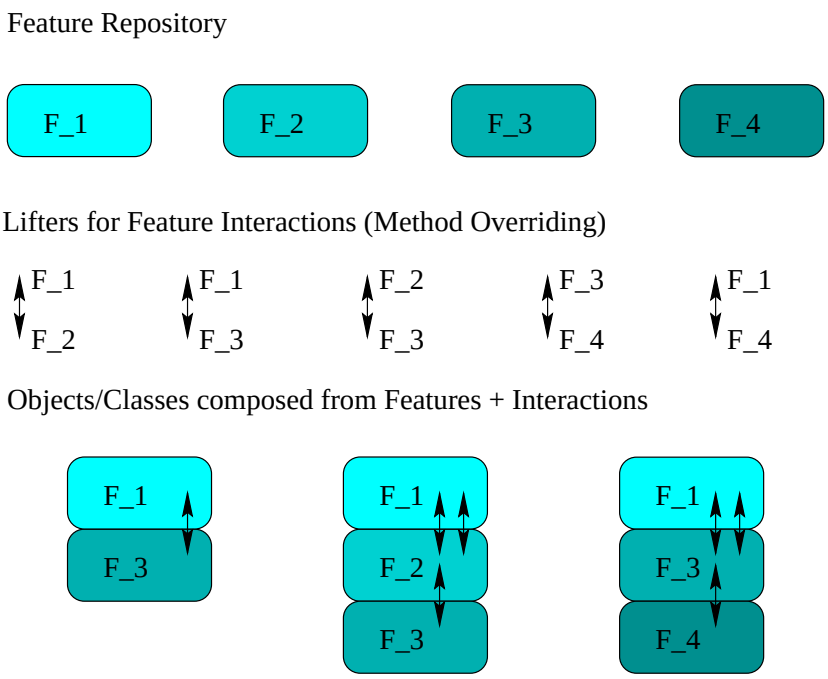

Figure 2.2: Composing Objects in the Feature Model

inheritance and then using aggregation and delegation. This explains the relation with known techniques and compares both techniques. In fact, we will show two cases where aggregation is more expressive than inheritance, refining earlier results [Ste87].

Other contributions include parametric features, similar to parametric classes, and generic method redefinitions. An interesting observation is that interactions can also appear, for parametric features, on the type level. We will examine cases where type parameters of classes have to adapt under feature composition. This is the case if the functionality and the syntactic interface of a feature depend on other features. In many cases, the adaption process for a feature is generic for each method, e.g. for the lock feature, all other methods are disabled. This is possible with generic lifters.

In the following section, we explain the architecture for composing features and lifters, which is the main idea of feature-oriented programming.

2.4 Feature Composition and Lifters

In object-oriented programs it is customary to override methods of a superclass in a subclass in order to adapt them to a more specialized setting. We view overriding more generally as a mechanism to resolve dependencies or interactions between features, i.e. some feature must behave differently in the presence of another one. For this purpose, we need to provide lifters, which adapt a feature to the context of another feature by

overriding methods. This leads to a new view of inheritance, as feature interactions are resolved between individually two features. In contrast, inheritance just overrides the method of the superclass.

We start with a well-known example to introduce the basic adaption techniques. Consider the stack and counter features of the above feature set. For each feature we first define an interface, followed by an implementation, as shown below. We follow Java [GJS96] syntax, except for feature definitions, which replace class definitions. Features consist of an interface and a feature constructor, here `SF` and `CF`. (In general, there can be more constructors for one feature, but this is not relevant here.) The constructors are written similar to class definitions, except for the new keyword `feature`.

```
interface Stack {
  void empty();
  void push(char a);
  void push2(char a);
  void pop();
  char top();
}

feature SF implements Stack {
   String s = ...
   void empty() {...}
   void push(char a) { ... };
   ...
}

interface Counter {
  void reset();
  void inc();
  void dec();
  int size();
}

feature CF implements Counter {
   int i = 0;
   void reset() {i = 0; };
   void inc() {i = i+1; };
   ...
}
```

In addition to the individual specifications, we have to consider interactions. In this case, we adapt the stack operations in the presence of the counter. This is done by

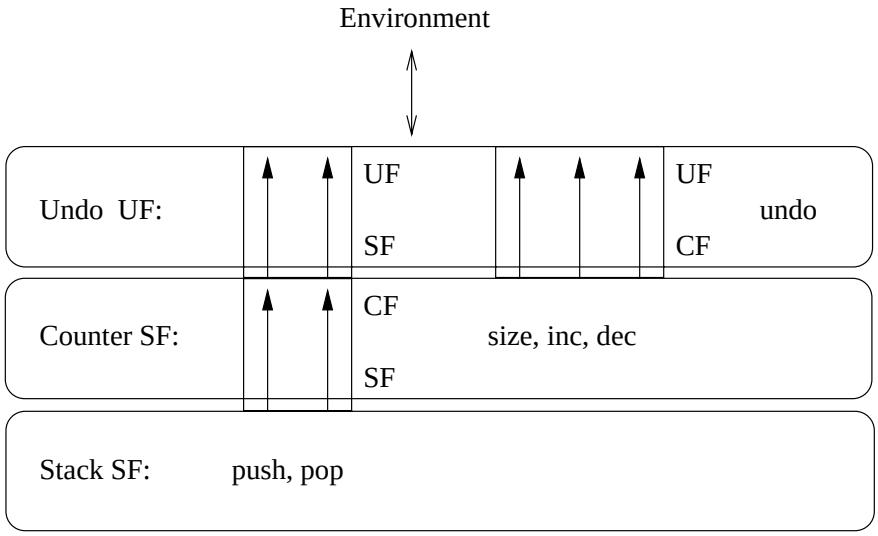

Figure 2.3: Composing features (rounded boxes) by lifters (boxes with arrows)

a lifter, which redefines the stack operations in order to use the counter, written as follows.

In the following code, the new keyword `lifts` states that some implementation of the stack interface is redefined in the presence of the constructor `CF`. It is important that we do not redefine just a single implementation, but specify this for all stack implementations. This enables repeated adaptations.

```
feature CF lifts Stack {
  void empty() {this.reset(); super.empty();};
  void push(char a)
        {this.inc(); super.push(a) ;};
  void pop() { this.dec(); super.pop() ;};
}
```

In a conventional object-oriented program, one would construct the counter as a subclass of stack and do the redefinitions directly. Although this may seem to be sufficient for small examples with a few features, we favor the separation of these code pieces. In turn, they can be combined in a flexible fashion. For combining more than two features, we need a suitable feature combination architecture. (The composition architecture is trivial for the above example with two features.)

We show in Figure 2.3 an example for a feature composition with liftings, many more combinations are shown in Section 5.6. In this example we first add the counter to the basic stack. For this new object to support the stack feature, we have to lift the functions push and pop, indicated by arrows in the box denoting the lifting. This

gives, like inheritance, a new object with two features, consisting of the inner two boxes. Since there are interactions between the two features, we must provide individual lifters for push and pop. Otherwise, one can use the default ones for composing orthogonal, independent features.

With the undo feature, we proceed in the same way, although undo is inherently more complicated. The goal is that undo can be added to any feature combination and undo works with all selected features. Hence undo has to undo the stack operation, the counter and possibly other features. The idea of our simple and flexible undo implementation is to save the local state of the full object each time a function of the other features is applied (e.g. push, pop). In this sense, undo is generic wrt all other features. The undo feature depends essentially on all "inner" features, since it has to know the internal state of the composed object. This multi-feature interaction is solved by an extra feature, which allows to read and write the local state (which is not shown here). Since the state depends on the feature combination, the typing for state access functions depends on this as well. This dependency can be described by type relations between features, which are expressed as interactions. With these expressive type concepts, a statically typed undo feature is easily implemented and is polymorphic in the state of the inner features. Apart from the typing issues, the undo feature works with the same feature composition scheme shown earlier. The functions push and pop are lifted again to undo, now with the lifter from stack to undo.

In the above example, there are two lifters needed (two boxes) for adding undo to the object with counter and basic stack features. This is the main difference to inheritance, where a concrete class undo would extend a class with counter and stack and would redefine some of their functions. Whereas all this happens, conventionally, in one subclass, it is divided into three entities in our approach: one feature and two lifters. Note further that lifting push and pop to undo does not depend on the counter; only the lifted versions of push and pop are lifted again by a lifter which depends on undo and basic stack.

It may appear that dependencies are described by a lifting from one concrete feature to a new feature. This is however not sufficient, as can be seen in the above example. The point is that a lifter can be used to adapt any component which has this feature. More formally, we have the following: for any object having the set of features A, we can add feature b and lift the functions of each feature in A individually to the new context. Then we have an object which provides b as well. For instance, in Figure 2.3, the lifter which adapts the stack functionality to undo can be used to adapt any combination which includes the stack. In Figure 2.3, it is used for adapting a counter plus stack object, but it is equally possible to apply it to the stack directly.

Using the structure of liftings, it is easy to model classical inheritance. Consider adding a feature b to an object with a set of features A. To obtain a concrete subclass, one has to merge the code of the feature a with all the lifters from $a \in A$ to b in the appropriate way. Repeating this for all features, we can create a concrete class hierarchy for a particular object composed from some features. This amounts to the main difference to inheritance.

2.5 Functional Feature-Oriented Programming

A fully functional model of an object-oriented language using the above concepts is presented in Chapter 5. In the functional setting, we compose features with the appropriate interaction handling using the type system of a particular functional language, namely Gofer [Jon91]. The flexible composition of features is achieved by advanced concepts of functional programming, such as monads and monad composition techniques [Mog91]. This also presents a concrete model of the monad constructions used for specification, as discussed later. For this integration, no language extension is needed and both programming styles can be used interchangeable.

In our functional setting, (constructive) functional specifications are executable, jointly with imperative ones. This gives a convenient implementation and prototyping language, which is close to specifications. It allows one to use object-oriented techniques while preserving the benefits of a functional language with higher-order types.

Furthermore, we demonstrate that type systems of current functional languages can express the concepts needed for feature-oriented programming. For instance, we model so-called binary functions [BCC+96], which express that two object types are exactly of the same (sub-)type. For instance, a copy function returns an object of exactly the same type. Binary functions are modeled here by adding a type parameter. This embedding gives a type inference algorithm for features, with only a few restrictions.

Our concepts are implemented in Gofer and generalize some monadic programming techniques, where objects correspond to monads, features to monad transformers, and feature interactions are resolved by lifting functions through monad transformers.

2.6 Imperative Feature-Oriented Programming in Java

Feature-oriented programming in an imperative setting is presented in Chapter 6. In particular, we show how an existing language, namely Java, can be extended to support feature-oriented programming. Since some of the earlier concepts are simplified and restricted for this integration, this version of feature-oriented programming is easier to use. This in turn gives an ideal implementation platform for feature-oriented development.

We show that Java is easily extended to feature-oriented programming. In this extension, we discuss the typing and composition problems of features and interactions. This provides for an imperative implementation language, which integrates feature-oriented programming in Java. Our extensions are defined via direct translation to plain Java via inheritance or aggregation. This delineates the relation to object-oriented concepts and in addition gives a detailed comparison between inheritance and aggregation. We show that aggregation is more expressive for some more advanced composition concepts.

The translation via inheritance builds a concrete class hierarchy for each used feature combination. In this class hierarchy, the features and interactions are merged. Similarly, the translation via aggregation composes the features and the interactions into a class with delegate objects for each feature. This provides for a close link with object-oriented programming and allows the use of tools, e.g. debuggers, developed for these languages.

We model more advanced programming concepts via Pizza [OW97] (which includes a translation to Java). Pizza extends Java by several key ingredients of functional languages. These include parametric polymorphism, data type declarations with pattern matching (or class cases in object-oriented parlance), and higher-order functions. With these extensions, we are able to add the advanced type and composition concepts for features developed in Chapter 5. The chapter concludes with several examples covering variations of design patterns, telecommunication applications and groupware systems modeled by automata.

2.7 Feature-Oriented Specification

We outline in the following our specification concepts for feature-oriented design. The goal of this section is to show that abstract specifications of complex objects can be composed from individual feature specifications plus the interaction specification. The detailed treatment in Chapter 3 focuses on the simpler case of just specifying a single object, whereas Chapter 4 covers the more difficult case of object networks.

The specification of composed objects is important in several ways. For the development of software components in a precise way, modular specification techniques are helpful for capturing requirements. Similarly, the user of a component, e.g. in a program library, wants to understand the exact behavior of a component without reading the detailed source code. In this case, specifications are used for precise documentation.

The main contributions of our specification techniques for sequential, stateful objects are explained in the following.

Specification Styles

Our techniques allow the specification of features and their interactions on different abstraction levels. We use behavioral specification with implicit state, which just express equalities between operation sequences, in the style of algebraic specifications [Wir90, EM85]. Alternatively, our model permits state-oriented specifications which directly talk about the effect of an operation on the state. These are similar to Hoare calculi with pre- and postconditions [Hoa69]. In both cases, our techniques admit equational reasoning about imperative programs.

We specify the behavior of objects by equations between programs. As an example of a behavioral specification consider the following specification of a stack, where `push x` adds an element `x` and `top` returns the top element. Observe that the notation differs

from the imperative version presented earlier. For simplicity, the object on which these operations work on is kept implicit.

```
push x; top = push x; result x
```

This equation states that the two computations produce the same result and have the same effect on the state. Note that `result` is the operation which only returns its parameter as the result of a computation. It is important to stress that we cannot simply write `result x` on the right. Since operations modify state, equality also has to consider the (implicit) state.

It is instructive to compare this to a state-oriented specification of `push`, which assumes a concrete implementation using lists. This specification assumes a mutable variable of type $[Int]$, which can be accessed via a function `get_list`. The effect of `push` can then be expressed by comparing the state before and after its execution. We write this as a context equation, which states that after reading the state and executing `push`, the following equation between programs holds:

```
l <- get_list; push x  =>  get_list = result (x :: l)
```

Note that we write `l <- get_list` to assign the result of the function call to the locally bound variable `l`. The equation expresses that the state has changed to `x::l`, which denotes the list with `l` appended to `x`.

Interaction Specification

In addition to the individual feature specifications, we have to specify how a feature behaves in the presence of another. The simplest case is when two features are fully independent. Typically, independent features work on different parts of the state and operations of the two features can be treated separately. For instance, we can reorder operations of the two features, as the order of two independent operations is immaterial.

Our design goal is that the specification of interactions only adds new information about a component without invalidating the individual specifications. Thus, feature specifications and interaction specifications complement each other.

In case of an interaction, we have to specify how a component with both features works. This can be realized in specifications as follows. The feature specifications define properties of operation sequences which only contain operations of a single, individual feature. In turn, the interaction specifications also cover "mixed" operation sequence. Assume for instance the above stack specification and the counter feature. The counter has functions `inc` and `size` etc to increment and, respectively, read the counter value. The interaction with the counter is a particular cooperation: the counter is intended to count the stack elements. We specify this by the following context equation:

```
c <- size; push x => size = result c+1
```

This specification applies to components which have both features (but possibly more). It specifies a relation between both features. To fulfill this specification, an implementation may add extra functionality (via method redefinitions) to `push` (i.e. incrementing the counter), as shown earlier. In addition to the above, we have to assure that the individual specifications still hold for the feature combination, which is in general not the case. This problem is considered below under the heading refinements.

Generic Features

Another contribution are specification concepts for generic features. These are features which affect others in a schematic way. Examples are the undo or lock features. Both can be specified, formally and informally, by referring to a set of features without knowing details of the other features. For instance, lock is informally defined as disabling other features. We show that such informal statements can often be expressed by high-level specifications in a concise way. This is possible via generic access to state. Many generic features can be specified in terms of the (abstract) state access of other features. More technically, these features are polymorphic in the state of the inner features.

Refinement Principles

Our goal is to obtain the specification of a feature combination by composing individual features. The easy way to do this is to conjoin the individual specifications plus the interaction specifications. Unfortunately, it is not always the case that the individual specifications can be used as is. This problem occurs in case of conflicts (in specifications) between features. For instance, the lock feature is intended to disable other features. Hence the laws may not hold as before. Another example is the bound feature, which restricts the stack elements to be of a certain size.

Our contribution is to identify the typical cases of refinements and to develop formal refinement concepts. The idea is that some specification laws may only hold under some abstraction. Typically, one uses abstractions on the state to show that laws hold. The main benefit of our approach is that typical abstractions can be defined using the types of objects. Since in our case the types of objects represent the feature combination structure, is it possible to write schematic abstractions suitable for feature- and object-oriented programs.

However, (conventional) abstractions are not sufficient in some practical cases. For instance, there is no abstraction on the state to show that a bounded stack fulfills the stack laws. The problem is that the laws do not hold for out-of-bound elements, which cannot be inserted. A common solution for this problem is conditional refinement, which expresses a global condition on the usage of some feature. In this example, we need the condition that the stack is only used with elements fulfilling the size restrictions. It is however not practical to verify this for each particular usage.

Our solution to this problem is a simple application of our underlying computational model. As we model exceptions, we can define refinement modulo exceptions. In this approach, the bounded stack raises an exception if an element is inserted which is too big. The refinement, similar to partial correctness, states that the specification holds unless an exception occurs. The main advantage of this techniques is that we do not need to verify the conditions for every usage. If no exception occurs, we can be assured that a program with a bounded stack has produced correct results. This is important for semantic subtyping, as discussed below. On the other hand, this does not show that no exception occurs. This is usually not possible for all programs using stacks, but has to be done for each concrete usage. In this case, we still have to verify the conditions, which can be difficult.

Semantic Subtyping

While in object-oriented programs subclasses create subtypes, in our setting subtypes are induced by set inclusion of feature combinations. Thus, an object which implements more features than another one is a subtype of it. The idea of subtyping is that a subtype object can be used at any place where a supertype can. This syntactic rule should be reflected by semantic properties. Thus, if we know what a procedure does, we can use the procedure with a subtype and still get the same behavior. This is however not the case in general. In the disguise of features, we address the typical specifications problems of object-oriented systems with behavioral subtyping [Ame90, LW90, DL97, LW93].

In order to guarantee well-behaved subtyping, we use the restriction to conservative redefinitions. Informally speaking, these redefinitions only allow to add extra behavior, but no modification of existing behavior. We present a result which is similar to the ones in [DL97, LW93], but is based on conservative redefinitions. We claim that the notion of conservative redefinitions is practical and sufficient for our methodology. It is in general weaker than the much more involved techniques in [DL97, LW93]. However, our results go beyond current ones for the case of homogeneous object networks and for the case of exception handling. With abstraction over exceptions, we can establish a subtype relation as sketched in the last section.

Virtual Functions

The notion of virtual functions as in object-oriented programs extends naturally to our setting. Virtual functions are bound at run time, depending on the actual subclass/feature combination. For modular design, we want to specify features individually without referring to other features of an object. For virtual functions, the actual behavior however depends on the added features, which can redefine methods.

For specifications purposes, virtual methods in a feature can be interpreted by an assumption on the full object, which is composed of several features. When creating

an object, this assumption can be discharged. For technical reasons, it is practical to model this with a global object store, as discussed below.

Our technique for the specification of virtual functions is the same as for semantical subtyping. We use an abstraction function to ignore other features and just specify virtuals partially by the effect of the virtual function on the actual feature of concern. In case of conservative method redefinitions, we are assured that the local behavior is not changed, only extended.

Object Networks

We show in Chapter 4 that the main results for individual objects of Chapter 3 also hold for systems with linked objects. For this inherently more difficult case, some results are harder to obtain and have to be restricted somewhat.

To accommodate a set of objects, we use a dedicated data type for object identifiers. Furthermore, the object state is replaced by an indexed set of objects (which is still kept abstract via monads). By convention, each operation takes an object identifier as the first parameter, which represents the object it works on. This is usually written in different, "message passing" or record selection syntax in object-oriented programs.

In the presence of linked objects, some specifications, particularly with invariants, are tedious to model with our specification style. For instance, to maintain the invariant that two objects are not reachable by some reference chain, it is usually easier to reason about predicates specifying this property instead of reasoning about algorithms which determine such reference chains. For this case, we show that we can integrate common predicative specifications with quantifiers. As we use these only for invariants, these can be integrated smoothly.

Monads for Specification

The technical novelty of the approach is that we use implicit state in the form of monads for specification. We focus on two particular monads, state transformers and error monads, which are used to model exceptions. We show that specifications with implicit state enable abstract, high-level specifications. These are useful for modeling feature interaction (or inheritance) as refinement, similar to behavioral subtyping and data-type refinement concepts.

With monads, we can give a simple feature-oriented computational model which includes most object-oriented language concepts. We build upon parametric polymorphism and a mechanism for function overloading, as used in the functional languages Gofer and Haskell. We need more advanced concepts such as type constructors and dependent types for the global object store and for the used monads.

2.8 Related Work

Since we cover the full range from specification to programming techniques, there is an abundance of literature, particularly for object-oriented programming. Most of the relevant literature falls into the categories of the following two sections, object-oriented programming and specifications.

The work on feature interactions largely concentrates on detecting interactions and on interaction avoidance. A similar problem of flexible services occurs in groupware applications, as shown in [Tee97]. The approach there covers a wider range of issues, such as dynamic change of services at run time and aims at configuration by the end-user. In contrast to our approach, [Tee97] does not consider interaction resolution, e.g. by adapting functionality as done here.

Note that our notion of features is not related to the notion of features of an object in Eiffel [Mey92] or other languages. Furthermore, feature-oriented domain analysis [KCH+90] is a technique for requirements analysis which aims for a similar structuring into features but on a more informal level with different goals.

2.8.1 Extensions of Object-Oriented Programming

We compare the feature model to other approaches and discuss some particularities of our approach below. Although there exist other approaches on class composition and inheritance, we argue that the feature model provides maximal flexibility (with static typing) and is as simple as possible.

- Mixins [BC90] have been proposed as a basic concept for modeling other inheritance concepts. Mixins are similar to classes which can be added in a flexible fashion like features. The main difference is that we consider interactions and separate a feature from its interaction handling. If mixins are used also as lifters, then the composition of the features and their quadratic number of lifters can be done manually in the appropriate order. Instead, we can just select features in our approach.

- Method combination with before, after and around messages in CLOS [LM91] follows a similar idea as interactions. As with mixins, this does not consider interactions between two classes/features and gives no architecture for composition of abstract subclasses. Such after or before messages can be viewed as a particular class of interactions.

- Composition filters have been proposed in [BA96] to compose objects in layers, similar to the ordering of features in our approach. Messages are handled from outside in by each layer. The main difference is that we consider interactions on an individual basis and separate a feature from interaction handling.

- Merging different aspects, functional and non-functional, of a software system is pursued by aspect oriented programming [KLM+97]. This can be viewed as an

even more general approach than feature composition, but is so far pursued only for certain application domains. We think that feature-oriented programming can contribute to a more general theory of aspect composition.

- Several other approaches allow to change class membership dynamically or propose other compositions mechanisms [MMPN93, US91, Frö96, SPL96, Mez97]. One of the main ingredients for feature-oriented programming, lifting to a context, can also be found in [SPL96]. All of these do not consider a composition architecture as done here, and address other problems, such as name conflicts. Due to dynamic change, static typing can be problematic. Clearly, the idea of features can also be applied to dynamic composition, but this remains for future work.

- Subject-oriented programming [HO93] has been proposed as a model for capturing different views or roles of the same object. Feature-oriented programming can contribute to this idea as a composition technique, as different views may clearly interact.

- Role based design and implementation is studied in [Van97]. The exposition however only covers layered composition, but no notion of interactions. (Except for our notion of lifters, which is already quoted and discussed there.) Clearly, interactions can be treated as extra layers, which is however not practical for larger compositions.

- Design patterns appear similar to feature-oriented design, as they try to capture good design solutions. It is possible to view the feature composition concepts developed here as a particular design pattern. However, feature-oriented programming aims at different goals. First of all, it is our goal to compose features automatically, not to implement for each used feature combination individually. Second, this view only considers implementation but not design techniques for feature combinations (as e.g. shown in Section 6.2).

  We view feature interactions and feature composition as a design principle which deserves explicit language support, similar to inheritance. Also, we can use feature-oriented design to implement a flexible and adaptable design pattern library, as sketched by a few examples in Section 6.6.1.

- Frameworks have been discussed in the last chapter. A main problem of frameworks is that the dependencies between classes are difficult to control [Gam96]. Furthermore, frameworks often leave little flexibility due to a monolithic class hierarchy which is often difficult to apprehend.

- Another approach to flexible software libraries using code generators was presented in [BSST93, BG97]. This approach also uses layers for composition, but does not consider interactions between components.

Intentional programming [Sim96] follows similar goals, also with an emphasis on domain specific program optimizations.

2.8.2 Specification Techniques

There are surprisingly few approaches to the actual specification of object-oriented programs. For instance, [DL97, LW93] apply state-oriented specification to object-oriented systems. Other approaches which use explicit state (in the style of Hoare calculi [AL97]) or predicate transformers [MS97]. The latter focuses more on calculi and refinement concepts. The most interesting results, from our point of view, are the results on semantic subtyping in [LW90, DL97, Ame90, LW93]. We will redo and extend these in several respects, as mentioned earlier.

Concerning semantics of object-oriented programs, there is a large variety of elaborated semantical descriptions and logics. As most of these semantical descriptions do not consider more practical specification problems, such as refinement and semantical subtyping, we only discuss a representative selection of the existing literature. For instance, [KR94] compare operational and denotational semantics of object-oriented languages. Among the many other approaches, [SJE91, Ehr96] give a semantical model for object-oriented languages based on temporal logic. An abstract description of object-oriented systems using terminal algebras can be found in [Jac96a, Jac96b]. In contrast to the latter, we do not talk about semantic equality of classes. Our goal is to specify the behavior of concrete objects via features.

Type theoretic approaches, e.g. [AC95, Cas97, BCC$^+$96, PT94], aim at modeling object-oriented phenomena in a logical calculus. Although this is needed for any realistic specification language, these approaches focus on expressive typing systems and type inference for object-oriented languages. Furthermore, many of these works are not concerned with mutable variables.

Our specification style with implicit state is not new, but the usage of monads with the extensions considered here is new. Our type of behavioral specifications is called operation specifications in Raise [RAI92]. A similar technique is used in [WP94], called trace specifications. Both approaches do not consider the application to object-oriented systems and the extension to exceptions. There exist other formalisms [KPR97] which focus on state-based specifications of interactive components with asynchronous communication. Although features and (some) interactions can be semantically determined in [KPR97], no systematic method for interaction handling and composition is considered.

2.8.3 Summary of Contributions

Our main contribution is feature-oriented design, which we treat in the context of formal specification and programming languages. The main benefits of our novel structuring concepts are clarity between feature dependencies and therefore better reusability.

We claim that these improve object-oriented techniques such as inheritance and aggregation, which is also supported by many examples. Our main contributions to feature-oriented design are as follows:

- Our feature composition architecture from which we can generate code for customized feature selections. For a set of features, an exponential number of different feature combinations is possible, assuming a quadratic number of interaction resolutions.

- Modular specification and reasoning techniques which use the structure of features and their interactions. Thus the behavior of components with several features can be determined from the individual feature behavior.

We show that a wide range of concepts, techniques and technical results can be generalized to this new paradigm. For specification and programming languages, we cover several concepts of object-oriented languages in a more general setting. These include subtying, virtual functions and exceptions. In this way, several novel contributions appear. These are summarized in the following.

Several interesting kinds of features and interaction patterns have been developed:

- Generic features, which extend a set of other features in a schematic way based on generic state access. Prominent examples are the lock and the undo feature.

- Parametric features with type dependencies between features are examined. This shows that the idea of interactions is valuable also on the type level.

- Features with exceptions add new flexibility in programming and specification. The novel idea for programming is that exceptions can be added or omitted for a program as needed. For specification, they can ease the specification of programs which possibly raise exceptions.

For the specification of feature-oriented programs, we show that several results for object-oriented programs can be simplified in our more general setting. These and other new results are the following:

- We outline a methodology for modular specification of complex components by feature and interactions specifications. These include calls to virtual functions with late binding and exceptions.

- Refinement concepts are adapted to our setting in order to reason about the behavior of components which comprise several features. It is discussed how different specification styles affect refinement. Furthermore, we show how abstraction functions needed for refinements can be generated from our type system.

- We show that a certain class of conservative interaction handlers preserves feature specifications. This generalizes results on object-oriented systems for semantic subtyping to our setting.

- The results for semantic subtyping can be extended to the context of object networks. For this setting with object references, we can treat homogenous structures by a new model for object networks with a two-dimensional structure of the global store.

For feature-oriented programming languages, we show the essential steps needed to extend an object-oriented languages by our concepts of features. These lead to several interesting insights about feature- and object-oriented programming. Our main contributions are:

- Extension of a concrete programming language (Java, Pizza) by features with advanced type concepts and exception handling.
- For parametric features, we give two alternative translations to object-oriented concepts, which reveal differences between aggregation and inheritance in object-oriented languages.
- We propose translations for generic features and features with exceptions to common object-oriented language concepts.

We discuss the basic ingredient of our feature composition approach, the focus on two-feature interactions. This assumption enables our flexible programming and specification concepts; we claim that these outweigh the efforts needed to circumvent the rare case of true multi-feature interactions. This is supported by many examples and analysis of interactions.

Chapter 3

Feature-Oriented Specification

In this chapter we consider the specification of feature-oriented software components, which we view as stateful objects with some services. We focus on two main goals: practical specification techniques for feature-oriented systems which include the essential ingredients of current object-oriented languages. Furthermore, we discuss the problem of composing feature and interaction specifications in order to obtain the specification of a larger object from its individual features. This generalizes the known problem of behavioral subclassing to our more general setting of feature composition. In this way, we treat several problems which occur similarly for the specification of object-oriented programs. As we focus on the specification of just one isolated object here, we do not cover object references, which are treated in Chapter 4. Hence we do not use any kind of object identifiers, which are introduced in Section 4.1.

A feature implementation provides a particular service and is similar to a subclass which adds functionality and state to an object. A feature specification describes the behavior of the functions (or methods) of the feature. The interaction description details how the functions of two features must behave in the presence of both features.

In our modular specification approach, we specify each feature separately and then focus on the interaction specification for two features at a time. For feature specification, we discuss several specification styles, which differ in the way they refer to the state. As we specify features via equations on state-transforming operations, it is easy to give verification techniques in order to reason abstractly about the behavior of a composed, complex object.

From the outside, an object simply consists of a set of features. It is desirable to keep specifications abstract, without any details on the order of the feature composition. We will however see that some advanced specifications are difficult to express without information about the feature composition.

We assume that features are added in an ordered, linear fashion. As in object-oriented programming, adding a feature allows to redefine functions of the underlying, inner features. The ordering of features induces priorities between features, which is used for interaction resolution on the programming level. The redefinition is needed, since functions of a feature must fulfill the additional properties of the interaction

specification in case interacting features are added. Furthermore, the individual specifications should be preserved under such redefinitions. This is however not the case in general. Often properties only hold under some abstraction or condition on the usage of the composed object. In this case, we speak of a refinement of the specification of the features.

We examine in what cases feature specifications are preserved under feature composition. This problem is particularly important for subtyping, since adding features to an object type creates a subtype object. As in object-oriented languages, this can be used instead of an object of the required type. Therefore, we want to know when this is semantically sound. In other words, a program supplied with a subtype object should behave as an object of the specified type. We show that a particular class of method redefinitions allows to establish such a generic result. In this more general form, we address the typical problems of state-based specifications with behavioral inheritance [Ame90, LW90, DL97, LW93].

As we introduce a language with inheritance, virtual functions and binary methods [BCC+96] in a functional setting, this also yields a functional model of an object-oriented language. Furthermore, our specification techniques provide for a smooth transition from functional modeling towards state-oriented specifications and implementations.

Another novel contribution are specifications for features which affect others in a schematic way. Consider for instance an undo function, which is informally specified as "revoking the effect of the last operation from any other feature". Another example is the lock feature, which is intended to disable the services of the other features. Our techniques will allow to specify such generic features formally in a concise way.

Our specification formalism uses monads as the underlying computational structure. Monads are a quite abstract model which will turn out to be useful in many cases. Generic features are just one example. In general, monads are convenient to model many effects of programming languages, as discussed in [LHJ95]. Since we use implicit state with monads, we do not need any notion of (extendible) records [CM91], as in many other models of object-oriented languages.

For the following specification techniques, we limit ourselves to sequentially used objects. Furthermore, we use a higher-order logic with total functions. This is sufficient for our purposes, since we only want to describe the behavior of components. For a component, we expect that each service function terminates in finite time. Since we focus on the interaction on services, we are only dealing with the specification of finite operation sequences. Furthermore, we are not concerned with non-determinism, as for instance modeled in [KPR97].

3.1 Basic Definitions and Conventions

Our notation for functional programs and specifications is similar to common functional programming languages like Haskell [PHA+96] and SML [MTH90]. For more detailed

treatment of higher-order logic we refer to [Gor88] and for λ-calculus to [Bar84].

We use in the sequel the following basic language constructs and naming conventions:

- $Int, Bool, String, ...$ are basic data types. Note that () is the empty type. For equality on data types we use `==`.
- a, b and sometimes α etc are type variables, possibly type constructors.
- $\_ \to \_, (\_, \ldots, \_)$ and $[\_]$ are the usual type constructors. If a, b are types, then $a \to b$ is the type of functions from a to b. Arbitrary n-tuple types are constructed with $(a_1, \ldots, a_n)$, where a_i are types. $[a]$ is the type of lists over a.
- m is a variable over type constructors. In particular, m is a monad by convention, as detailed later.
- We write `t` $:: a$ if the term `t` has type a. Note that terms are written in typewriter font, while types are usually kept in italics.
- `True, False, [], [1,2,3]` are elements of data types (here booleans and lists).
- `push, pop, size` and other lowercase variables denote functions of some feature.
- `op, op'` are variables standing for operation sequences.
- `f x` or `f (x)` is a function application where `f`$:: a \to b$ and `x`$:: a$.
- λ`x.t` is a functional abstraction of type $a \to b$, where `x`$:: a$ and `t`$:: b$.

For readability, we generally use italic fonts for types and typewriter font for other program constructs.

We often use tuple notation e.g. $(1, 2)$, which should not be confused with parentheses used for function applications, e.g. `f (g a b)`. As in functional languages [PHA+96, MTH90], we use pattern matching if some data type is defined via a set of constructors. For instance, we write `length [] = 0`. This includes nameless functions presented as λ-abstractions. For example, a function which takes a pair as parameter is written λ`(x,y).t`.

3.2 Features, Interfaces and Programs

In the following, we will introduce our computational model. Up to Section 3.5, we adopt a very simple and abstract view of features and components. An object consists of a set of features. This includes the syntactic interfaces of the features, as well as their specifications. Clearly, the latter is more problematic and will be the main issue of this chapter.

We distinguish between the syntactic interfaces of features and concrete feature constructors. The latter stand for one implementation of a feature and may declare local state in form of instance variables. For both, we may add specifications. Since there is no state defined at the interface level, only abstract specifications of the stateful behavior are possible. We require that specifications or implementations for feature constructors must imply the specifications of the corresponding interface.

We assume in the following a set of feature constructors $\mathcal{F}$ and a set of feature interfaces $\mathcal{I}$, where each $F_i \in \mathcal{F}$ is associated with one $I_i \in \mathcal{I}$. In this case, F_i implements the functions declared in I_i, which is written as $F_i :: I_i$. Each object implements a set of feature interfaces. For a feature F in this set we say that the object has feature F. Compare this to object-oriented languages, where an object is associated with a class, but is also a member of all superclasses. Whereas subtyping is conventionally based on a class hierarchy, we simply use set inclusion on features instead. In this section, we introduce the basic setting and give specifications which are independent of how features are composed.

In our approach, programs are operation sequences on an implicit object state. First, we need to explain program notation and monadic types. Consider for instance the boolean function `is_element` of a feature (class) `Container`, which characterizes the elements of a integer container. As a monadic function (defined below), it has the type

`Interface Container`
  `is_element` :: $Int \ \rightarrow \ m \ Bool$,
  ...

where m is a monad. A monad m is a type constructor for which certain laws must hold, as shown later. A *monadic function* is a function of type $b_1 \rightarrow \ldots b_n \rightarrow m\ a$, where a is the result type and b_i do not contain the used monad m. The monad m encapsulates a computation, which is in this case a state transformation. We leave this computation abstract for specifications. More precisely, a monadic function returns a computation. In contrast, programming language constructs, e.g. `if _ then _ else _`, have monadic computations as arguments.

We show that keeping the monad abstract is particularly useful for supporting the concepts of inheritance/feature combination and exceptions. To get some simplified intuition for the state modeling, we may imagine a particular object state of type *state* via a definition for m as follows: $m\ a = state \rightarrow (state, a)$. Hence `is_element` :: $Int \rightarrow state \rightarrow (state, Bool)$. The used state depends on all the features used; to speak in object-oriented style, it depends on the actual subclass at run time. For other language features such as exceptions, different monads are needed. Since the internal type should be hidden to the outside, we only use the internal state if needed. Note that we use polymorphic types to "quantify over state", as state is modeled via polymorphic state transformers.

A key idea of monads is to distinguish between computations and actually running computations, which is shown later. This is convenient for manipulation of programs.

Since monadic functions modify implicit state, they are also called *operations*. As evident from the type, we cannot directly compose two monadic functions. Hence we will show an operator for sequential composition. The monad is used to model the type and the state of the object, which will be exploited in later chapters. In Chapter 4, we will use m to model a set of objects and also exception handling.

If the function `is_element` is used in a program or a specification, we assume (via some type system) that the object provides (at least) for the `Container` feature (interface). In most other imperative approaches for object-oriented programming, `is_element` would have a type $Container \to Int \to Bool$, where $Container$ is the type of an object. (This parameter is often left implicit in object-oriented languages.) To be precise, the type $Container$ refers to any subtype of $Container$ in most formalisms with subtyping.

One application of this abstract, monadic model of state is subtype polymorphism of object-oriented languages. Depending on the actual subclass at run time, different state is actually used (as well as different function definitions via subtype polymorphism and overloading). In our setting, the actual function will depend on the monad used at run time. Furthermore, we sometimes mix purely functional computations with monad computations. This shows the integration of imperative and functional style via monads. From the type it is easy to see whether a function is a monadic state transformer or not.

To define a feature, we have to declare the syntactic interfaces (also called signature) of a feature. This is visible outside and may be annotated with specifications (which do not use the implicit state.) Using monadic types, we fix the functions and types as follows:

```
Interface Stack
  push     :: Int → m ()
  pop      :: m()
  top      :: m Int
  empty    :: m()
  is_empty :: m Bool
```

Interfaces are like Java interfaces or Haskell type classes [NP95]. In most object-oriented languages, a subclass inherits the interfaces and definitions of its superclasses. In addition, an element of a subclass is automatically an element of its superclasses. As we aim for a more flexible model, we use a separate interface of each feature (subclass) and allow an object to provide for several interfaces. This suffices to give an abstract view of subclassing, where the subclass relation is just set inclusion on interface sets. (See also [NP95].) Hence we say an object has feature X, if it provides for the interface of X. As in Haskell (but unlike Java), we assume that a function name is declared in only one interface.

Concrete feature constructors for an interface are defined as follows:

```
Feature SF implements Stack
```

`state list` :: $[Int]$

This defines a constructor `CF` for a concrete implementation of the stack feature with state of type $[Int]$ (lists over integer). By convention, we will access the local state with monadic functions $\texttt{get}_{list} :: m([Int])$ and $\texttt{put}_{list} :: [Int] \to m()$. Later, we will use generic functions for state access which solely depend on the constructor name. In this way variable names, here `list`, are syntactic sugar and hence may be omitted.

For monadic computations, we use the syntax for monad computations of functional programming languages [PHA+96], which resembles imperative programs. An operation sequence on a stack may look as follows:

```
push 1; push 2; push 3; x <- top; push x; pop
```

Note that `_ ; _` $:: m\ a \to (a \to m\ b)$ is a monad operator which concatenates monad computations, i.e. state transforming operations in our case. The last operation, here `pop`, determines the result of the sequence. The result of an operation can be bound to a local variable (via `<-`), as in a let construct. More precisely, the notation

```
x <- top; push x
```

is an abbreviation for

`top;` λ`x.push x`

The arrow `<-` is omitted in case an operation returns the empty (void) type ().

To work with implicit state, there are the following common monadic operations. The function `unit` is the empty or null operation; in case of state monads it is the identity on the state.

`unit` $:: m\ ()$
`result` $:: m\ a$

The function `result` in addition to `unit` returns a result. Note that

```
unit = result ()
```

The functions `unit` and `result` must obey the following monad laws, which are easy to verify for state monads. The first two are called right unit and left unit, respectively, and the third expresses associativity.

`result x;` λ`y.f = (`λ`y.f) x`
`f;` λ`y.result y = f`
`f;` λ`y.(g y ; h) = (f;` λ`y.g y); h)`

For more details on monads see e.g. [Wad92, Mog91]. Monads have been advocated as a clean and simple way of using imperative concepts in functional languages. Here we show that they are also useful for specifications.

The above monad functions are useful for specification, as e.g. shown by the following examples regarding stacks:

```
push x; pop = unit
top = x <- top ; result x
```

These equations state that the operations on both sides compute the same value and have the same effect on the state.

For concise specifications, we introduce a few abbreviations. If `op, op1, op2` are operations, we use context equations of the form

```
op => op1 = op2
```

which stand for

```
op; op1 = op; op2
```

Thus `op` is the context for an equality. As an example consider

```
push x => top = result x
```

Context equations can be viewed as a special case of conditional equations, as they put conditions on the state. We often use a particular form of conditional equations which we write in the following, similar notation. We write

```
op == b => op1 = op2
```

where `b` is some value to stand for

```
x <- op; if x==b then op1   =   x <- op; if x==b then op2
```

where `x` is a fresh variable. Recall that the function `==` $:: a \to a \to Bool$ denotes equality on data objects. Note that the above `if _ then` construct is not a monadic function, as it returns a computation. As an example for a conditional equation consider the following:

```
is_empty == True => pop = unit
```

For monadic operation sequences, we can define a sequential if-construct based on the functional one as follows:

$\texttt{if}_{seq}$ `_ then _ else _ ::` $m(Bool) \to m(a) \to m(a) \to m(a)$
$\texttt{if}_{seq}$ `op then op1 else op2 = b <- op; if b then op1 else op2`

For the last equation it is assumed that `b` is a fresh variable, as we may otherwise capture another bound occurrence of `b`.

In case of conditions which do not read or write the state, we also write

```
b => op = op'
```

instead of `if b then op = if b then op'`.

3.3 Feature Specifications

We show in this section how the individual behavior of a feature can be specified. Although many features are intended to cooperate with others, it is important — for reuse and structuring — to specify their "isolated" behavior first. The specifications for combinations will be presented later. Another goal of this section is to demonstrate that different specification styles can be used within our setting. Recall that we assume that the laws presented below hold for all constructors, if no state is referred to. Note that some feature specifications have to be refined (i.e. may only hold under some abstraction), for feature composition. In the remainder of this section, we ignore this issue, which will be discussed later.

3.3.1 Behavioral Specifications

It is often possible to specify the behavior of operations just by looking at a sequence of operations. We use equality on operations to show their effects. These can be used to simplify (or evaluate) the program. Since no state is mentioned, we can work on the interface level, as in this specification of stacks:

```
push x; pop = unit
push x; top = push x; result x
push x; empty = empty
```

This specification is sufficient for the basic stack operations. It does not mention any state at all and is hence implementation independent. This style resembles usual algebraic specifications [Wir90, EM85], but there are some more subtle issues due to state. Consider the following, slightly different version:

```
push x; pop; top = y <- top; push x; pop; result y
push x; top = push x; result x
push x; empty = empty
```

Although this specification seems equivalent to the former, the latter makes fewer assumptions on the manipulated state. The latter does not prescribe, as the former specification in the first law, that the effects on the state of `push` and `pop` cancel each other. This may not hold for some implementations and, more importantly, if other features are added. We will compare these two alternative specifications at several occasions.

3.3.2 State-Oriented Specifications

Whereas behavioral specifications relate operation sequences of a feature, state-oriented specifications specify the state change of a specification (and the result) using the

underlaying state. Typically, one specifies each operation by comparing the change of the state before and after the operation.

To discuss this style and for comparison to other formalisms, consider a simple specification of `push` via pre- and postconditions on the state with Hoare triples [Hoa69, Apt81].

$$\{\texttt{s == list}\}\ \texttt{push x}\ \{\texttt{x:s == list}\},$$

where `_ : _` is a function of type $a \to list\ a \to list\ a$ and `x:s` appends `x` to `s` and `list` is the local stateful variable which is modified.

The problem with such specifications is which variables are not affected by some statement. This common problem in specifications is often called the frame problem (originating from artificial intelligence research [MH69]) and discussed for specifying object-oriented programs in Hoare style in [BMR95]. In a subclass (or feature combination), it can be the case that more variables are affected by a method. For instance, consider adding the counter where `push` and `pop` maintain the counter variable. Hence this specification style is of limited use when inheritance is used. As stressed in [BMR95], the frame problem is often quite delicate for object-oriented languages, for methods can be redefined in subclasses. It is argued in [BMR95] that simple equality conditions on variables are not suitable in the presence of inheritance. The article presents a solution for stating in pre- and post-conditions which variables are affected and which are not, which is however quite involved in their formalism.

For specifying stacks for the feature constructor `SF`, we use the function $\texttt{get}_{list} :: m([Int])$. Hence the following laws are limited to this constructor. Since the direct formalization of the state-effect of a monadic function with equations is quite tedious, the above abbreviations are used. The idea is to "evaluate" a $\texttt{get}_{list}$ call after a `push` invocation as follows:

$$\texttt{s <- get}_{list}\ \texttt{; push x => get}_{list}\ \texttt{= result x::s}$$

This equation states that the implicit state of the SF constructor is affected as desired, but not more. In this style, we can specify this feature implementation as follows:

```
Feature SF implements Stack
   s <- get_list; push x  => get_list = result x::s
   x::s <- get_list; pop  => get_list = result (rest s)
   x::s <- get_list => top = result x
   empty => get_list = result []
```

The above specification uses the function `rest`:: $list\ a \to list\ a$, which removes the first element of a list. Observe that context equations are also useful for operations which produce only output.

Functional Specifications of State Change

State-oriented specifications as above often need auxiliary functions to describe the state change. This technique is useful in cases where behavioral specifications are not suitable. For instance, if the behavior depends on a larger history of operations, behavioral specifications can be quite involved and difficult to understand. Also, if the state changes are algorithmically more complex, the state oriented techniques have to be enhanced. Since we integrate object-oriented programs into a functional setting, it is easy to use the techniques of functional languages to describe the change of the underlying state. Thus we use auxiliary functions, which by construction do not affect the state, to describe the resulting state.

A common example is that of a bank account:

```
Interface Account
  deposit      :: Int  →  m ()
  withdraw     :: Int  →  m ()
  get_balance  :: m(Int)

Feature QF implements Queue state balance
  s <- get_balance ; deposit x => get_balance = result s+x
  ...
```

In this case, the state oriented view is more natural since the notion of a balance is generally understood. In contrast, a specification of an account by algebraic laws like

```
  deposit x; withdraw y = deposit x-y
```

(possibly assuming $x > y$) is less intuitive. Furthermore, there are good reasons against this specification. The left hand side includes two transactions, whereas the right hand side has just one. In case we extend the functionality for the account, e.g. to transaction reporting, the latter specification is inappropriate.

As another example consider the specification of a queue. The behavior of dequeuing may depend on all previous queue operations. Modeling this via reordering operations is a bit more delicate and may obscure the intuition. The following (incomplete) specification of a queue specifies `enque` in by comparing the state before and after the operation using the auxiliary function `append`.

```
Interface Queue
  enque      :: Int  →  m ()
  deque      :: m(Int)
  qempty     :: m()
  is_empty_q :: m(Bool)

Feature QF implements Queue
  state qlist :: [Int]
```

```
s <- get_qlist ; enque x => get_qlist = result append(s,x)
...
```

Note that we use a function `append` of type $Int \rightarrow [Int] \rightarrow [Int]$, which does not affect the state, as evident from the type. Hence the specification uses functional programming techniques which do not use implicit state.

3.4 Feature Interaction Specifications

In addition to the feature specifications, we must specify their interactions. We show in the following that in many cases the specification for a feature combination can be done at the interface level. This has the advantage that they are not state dependent and can be used for any corresponding constructor. Interaction specifications which require more details about specific constructors and combinations are shown later in Section 3.6.

For the combination of two features, we have to specify the behavior of "mixed" operation sequences. In order to reduce complexity, we consider two features at a time. On the programming level, this is a basic assumption (see Chapters 5 and 6). In our specification this need not be the case here, as it is possible to speak about several features in one specification.

In addition to the interaction specification, we have to assure that the specifications for the individual features hold. This is however a more subtle task, as these may only hold under extra conditions. This issue will be examined later in Section 3.7 on refinements. How we actually construct such a feature combination will be examined in the next section.

We show in the following a few typical examples where the interaction can be specified on the interface level only. From the used operations it is evident which features are concerned. Hence we do not introduce any extra syntax for interface level interactions and just state the laws.

In the examples below, only combinations with the stack feature are presented, as the queue interactions are specified in the same fashion. For more advanced interaction specifications, which rely on particular feature combinations, this is more involved and is discussed in Sections 3.6 and 3.11. In the examples below, only combinations with the stack feature are presented, as the queue interactions are specified in the same fashion.

To show feature interactions in the following, we briefly discuss two basic features.

Counter

For the counter, several styles are appropriate. We chose the more abstract interface specification:

```
Interface Counter:
```

```
size     :: m Int
inc      :: m ()
dec      :: m ()
reset    :: m ()

inc; dec    = unit
reset; size = reset; result 0
inc; size   = x <- size; inc; result x+1
```

```
Feature CF implements Counter state c :: Int
```

This declares the constructor CF, which will be used later. Note that the last equation reshuffles operations in order to specify their behavior.

Bound

The bound feature is intended to bound the size of elements of a data structure. Hence its individual specification is rather dull. It can be specified as follows:

```
Interface Bound
  set_bound   :: Int → m()
  check_bound :: m(Bool)
  set_bound x; check_bound y = set_bound x; result (y < x)

Feature BF implements Bound state b :: Int
```

3.4.1 Stack with Counter

For the combination of counter and stack, we want that the counter indicates the number of the current stack elements. Hence we only have to "modify" the counter laws to specify this combination:

```
empty; size  = empty; result 0
push a; size = x <- size; push a; result x+1
pop a; size  = x <- size; pop a; result x-1
```

By the used operations we can infer that the above laws refer to two features. Note that the above specifies previously unspecified behavior. The laws must hold for any combination which includes both features. In addition, we have to assure that the original behavior still holds. As we will see in the next section, specifications must be refined in some cases.

An issue not considered here is hiding. As it is not sensible to use the `inc` and `dec` functions of the counter feature, they should be hidden to the outside. (In case they are used, the specification produces unintended behavior. This may indicate that the specification is too loose.)

Note that there are choices for the interaction specification between two features. For instance, an alternative usage for the counter is to count the number of used operations. This alternative will be explored later. For clarity, we allow only one interaction specification for two features on the interface level. Thus, for a different usage of the counter, the counter has to be copied.[1]

3.4.2 Bounded Stack

The interesting part of the bound feature is linking it to another one, here the stack feature:

```
check_bound a == True => push a = unit
```

3.5 Feature Combination via Type Composition

So far, we have specified abstract behavior of an object which has a set of features. However, more details about the combination method are needed later for more advanced specifications and for refinement concepts. Therefore, we introduce our model of a layered feature combination via feature constructors in the following. In the layer model, features are added in a particular order using the feature constructors. These also serve as type constructors which build a new object type from another one.

Formally, we have the following constructions:

- A set of feature constructors $\mathcal{F}$, and, correspondingly, a set of feature interfaces $\mathcal{I}$, where each $F_i \in \mathcal{F}$ is associated with one $I_i \in \mathcal{I}$.
- A set of object types $\mathcal{O}types$, which is generated by feature constructors as follows: $F_n(F_{n-1}(\ldots F_1)\ldots) \in \mathcal{O}types$, where $F_i \in \mathcal{F}$, $i > 0$, and all F_i are distinct. A type $F_n(F_{n-1}(\ldots F_1)\ldots)$ implements all corresponding interfaces I_i, i.e. $F_n(F_{n-1}(\ldots F_1)\ldots) :: I_i$ for all $i = 1, \ldots, n$.
- A particular set of types $\mathcal{T}$ for instance variables of objects, which is generated by base types $(Int, Bool, \ldots)$, and the usual tuple constructors $(\_, \ldots, \_)$ and projections π_i.
- A mapping $\mathcal{T}ype :: \mathcal{F} \to \mathcal{T}$ which describes the state used by a feature. As the type of the state used by objects is represented as the tuple of the individual feature states, this mapping extends to object types in the canonical way: $\mathcal{T}ype :: \mathcal{O}types \to \mathcal{T}$ with

$$\mathcal{T}ype(F_n(F_{n-1}(\ldots F_1)\ldots)) = (\mathcal{T}ype(F_n), \ldots, \mathcal{T}ype(F_1)).$$

[1] To ease such redefinitions, it may be convenient to import the signatures and to rename the functions. The latter is needed since we do not allow names to be used in several features.

By convention, we will use δ, σ etc as variables standing for object types. Note that the above definitions do not permit functions as variables stored by features. (We use sum types as in functional languages occasionally, which can be encoded via tuples.)

As an example for the above definitions consider for instance the stack feature, for which there is a feature constructor `SF`. By construction, applying `SF` to any type adds the stack feature, i.e. the constructed type provides for the Stack feature. For instance, we construct a type for two features via `CF(SF(Id))`, where `CF` adds the counter feature and `Id` is the empty object type with no features. (Note that we will often omit `Id` for convenience.)

In object-oriented programming, functions (methods) can be redefined for each different subclass. This corresponds to redefinitions in case of feature combinations here. In our model, `push` for `SF(Id)` is different from `push` for `CF(SF(Id))`. To distinguish these, we annotate functions by the type of the object they work on as subscripts. For instance, $\texttt{push}_{SF(Id)} \neq \texttt{push}_{CF(SF(Id))}$. Note that this overloading depends on the type of a function. More precisely, it depends on the monad m. As we use implicit types, it is convenient and equally expressive to annotate the functions and not the types. The latter is used in the functional implementation in Chapter 5 and also below for illustrations.

Notice that only functions of the same object type can be composed to an operation sequence. Hence all operations in a sequence must have the same type subscript. For this reason, we also mark operations sequences with type subscripts. Integrating methods with different subscripts will be shown below.

As mentioned earlier, some specifications are stable under composition (similar to inheritance), some must be restricted. More precisely, their application has to be restricted. It is useful and often sufficient to use more precise object types for equations. We can simply restrict the operations in equations to particular feature combinations by adding types. The following are typical examples which illustrate the scope of equations:

- $\texttt{push}_{\sigma}$; $\texttt{pop}_{\sigma}$ = $\texttt{unit}_{\sigma}$ holds for any combination where the stack feature is present. (Recall that by the type system, we automatically assume that σ must include the stack feature.) This is the default in the sequel if no annotations are used.

- $\texttt{push}_{SF(\sigma)}$; $\texttt{pop}_{SF(\sigma)}$ = $\texttt{unit}_{SF(\sigma)}$ holds for the stack feature (without redefinitions), added to some feature combination σ.

- $\texttt{push}_{CF(SF(\sigma))}$; $\texttt{pop}_{CF(SF(\sigma))}$ = $\texttt{unit}_{CF(SF(\sigma))}$ holds for stack plus counter combination only.

We will later on use equations where the equality is restricted to the state of certain features. This allows one to abstract from (or ignore) any changes to the state of all other features. Here we only quantify over the feature combinations of the used operations. Note that annotations are only needed for constraining specifications to

(yet unknown) feature combinations. They are not needed for programming, where type inference usually does this job.

3.5.1 State and Feature Combinations

We explain in the following how the layer model supports the implicit state. Informally speaking, each feature constructor adds some piece to the full object state of a feature combination. More precisely, this is done via monad transformers, which transform monads into new ones, in Chapter 5. For specifications in our context, we use a simpler model tailored for state transformers.

Recall that the actually used monad depends on the used features. To be more precise, we can annotate the monad by the used object type, since the used monad depends on the used features. For instance, $m_{SF} \neq m_{CF(SF)}$. Recall that we use the type annotation on functions, as explicit types (with monads) are often omitted.

The following table shows the appropriate types for some feature combinations.

Features	Constructor	Used Monad
Stack	`SF`	$m_{SF} = Int \to [Int] \to ([Int], ())$
Stack + Counter	`CF(SF)`	$m_{CF(SF)} = Int \to Int \to [Int] \to ((Int, [Int]), ()$

In general, a function of a composed object transforms the state of all features. For modular specifications, it is often needed to embed a computation on a smaller state (with fewer features) into a computation on a larger state. The idea is to perform the identity operation on the extra state. We use the generic function `lift`:: $m_\sigma \to m_{F(\sigma)}$ to lift functions to the extended state. This embedding of computations via `lift` is essentially what happens in object-oriented languages with a call to "super", which (formally) constructs a function of the sub-class from a superclass function. For monad compositions via monad transformers, this is known as lifting, as discussed in Chapter 5.

The function `lift` must fulfill the following properties:

```
lift (result_σ x) = result_F(σ) x
lift ( f ; g) = lift f ; lift g
```

The above laws state that lifting `result` yields the function `result` for a different type and that composed functions can be lifted individually. The construction of the function `lift` is straightforward (for our case here) and shown in Chapter 5.

Recall that $\mathcal{T}ype(F)$ is the type of the state declared for feature constructor F, and similarly for feature combinations. We have for instance

$$\texttt{lift}_{F(\sigma)}\ \texttt{f}_\sigma :: \mathcal{T}ype(F) \to \mathcal{T}ype(\sigma) \to ((\mathcal{T}ype(F), \mathcal{T}ype(\sigma)), \alpha)$$

for lifting a function f of type $\mathcal{T}ype(\sigma) \to ((\mathcal{T}ype(\sigma), \alpha)$. A more general version in terms of monad transformers can be found in Chapter 5.

As an example, we can define `push` for the Stack plus Counter combination as an interaction resolution. The definition of $\texttt{push}_{CF(SF)}$ is shown in the first line of the table below. The two rows show the state used by the two features and illustrate how the state is affected by the operations. Observe the use of `lift` to embed the computation on SF into $CF(SF)$.

Operations:	$\texttt{push}_{CF(SF)}$ a =	$\texttt{inc}_{CF(SF)}$;	`lift` $\texttt{push}_{SF}$ a
$\mathcal{T}ype(SF)$	$\boxed{[Int]}$	$\boxed{[Int]}$	$\boxed{[Int]}$
$\mathcal{T}ype(CF)$	$\boxed{Int}$	$\boxed{Int}$	$\boxed{Int}$

We often use a generalized form of the above with the type $\texttt{push}_{CF(SF(\sigma))}$ instead. This is clearly more generic, as it can be added to any other feature combination σ.

To execute a monadic computation, we assume a function `run`, which invokes an operation with a given initial value for the state and returns a pair of the computed value and the updated object store. Assuming $m(a) = \sigma \rightarrow (\sigma, a)$, we can give the type of `run` as follows:

```
run ::  m(a) → σ → (σ, a)
run op_seq (o1,..,on) = op_seq o1 ... on
```

The last definition of `run` is based on the state transformer view and has to be adapted if other language effects are modeled. It assumes that the state of an object is represented by a tuple `(o1,..,on)`. An invocation `run op_seq os` thus gives a pair of the form `(os', val)`, where `os'` is the new state and `val` is the computed result value.

If operations are state transformers, we can apply the principle of extensionality, stating that `op_seq` and `op_seq'` are equal if

$$\forall\ \texttt{x} : \mathcal{T}ype(\sigma).\texttt{run (op\_seq}_{\sigma}\ \texttt{x)} == \texttt{run (op\_seq'}_{\sigma}\ \texttt{x)}$$

In our setting, types fulfill two purposes simultaneously: they determine the type of an object (i.e. the features it provides) and also model the state of an object directly. This close link will allow for generic access to the state of an object which is convenient for specifications. Furthermore, we can define abstraction functions on state using object types. As we represent an object type by a term of feature constructors, we can construct precise abstraction functions based on the syntactic structure of an object type. In contrast, usual type systems for object-oriented languages just use a particular class and not the set of superclasses as the type. This is the main difference to other type systems [AC96, PT94, Cas97], which focus on many subtle aspects of polymorphic subtyping. Although some of them are designed for imperative programs, they are not directly useful for our purpose. As we use type variables in such compositions, it is possible to reason about feature combinations. As an example consider the type term $CF(\sigma)$, where σ stands for some feature composition.

3.5.2 Generic State Access and Liftings

An essential ingredient for specifications of stateful features is generic access to state. In our setting, we can assume generic state access functions $\texttt{get}_F$ and $\texttt{put}_F$, which are parameterized by a feature constructor `F`. For instance, $\texttt{get}_{list}$ as used in the last sections is just a gentle version of the function $\texttt{get}_{SF}$.

- $\text{get}_F :: m(\mathcal{T}ype(F))$ reads state of feature constructor F.
- $\text{put}_F :: \mathcal{T}ype(F) \to m()$ writes state of feature constructor F.

It is useful to generalize the above functions for feature combinations:

- $\text{get}_\sigma :: m(\mathcal{T}ype(\sigma))$ reads state of any feature combination σ.
- $\text{put}_\sigma :: \mathcal{T}ype(\sigma) \to m()$ writes state of any feature combination σ.

These functions are essential for the specification and implementation of generic features, as shown later. A programming level version of $\texttt{get}_\sigma$ can be found in Chapter 5.

With feature combination (or inheritance) methods are redefined for feature combinations (or subclasses). Hence operations of one feature may work on different parts of the state, depending on the feature combination. On the other hand, it is important to reason abstractly about the state which is possibly affected by a function call. For this purpose, our "frame axiom" is essential for specification and verification: each function/operation of a feature can be presented as a read operation plus some computation followed by a write operation on just the feature state. Formally, for all operations `f` of a feature `F` there exists a `f'` such that

```
f = x <- get_F;  put_F y; result r    where (y,r) = f' x
```

Assuming `f`:: $\alpha \to m(\beta)$, then `f'`:: $\alpha \to (\beta, \mathcal{T}ype(F))$. This assures by construction that `f'` is side effect free. Any effect on the state must be returned via the variable y whose value is assigned to the state. Note that the frame axiom is more involved when several features are used. This law can be formally derived using the polymorphism in our feature definitions and specifications by the results developed in [Wad89]. Since this requires considerable technical overhead, it is not treated here.

In other formalisms, the frame axiom is stated on the meta level or by explicit statements. For instance, in the Larch language [CL94b, DL97], a "modifies" clause is used to restrict the effect of methods.

3.6 Specification of Generic Features

Some of the features mentioned in the introduction are very generic in the way that they refer to the full behavior of an object. For instance, the lock feature is intended to disable all other features. This informal specification refers to all other present features.

(More precisely, this refers to all "inner" features onto which lock is added.) We show in the following how to write such generic specifications in our framework. With the techniques considered so far, we have specified the behavior for two (or more) concrete features at a time. Although this is in general sufficient, it is tedious for features whose interaction can be described in a schematic way.

Generic features are features which can be specified in a schematic way referring to the state of other features, which are however kept abstract. Hence generic features can only refer to state, but not to other particularities of other features. This enables high-level formalizations. Note that we do not talk about the functions of other features directly, which means quantification over the syntactic entities. This clearly has the drawback that we would need some reflection principle in the logic to talk about its syntactic entities, which we want to avoid here.

3.6.1 A Generic Undo Feature

For the undo feature to work, we assume that every other feature must set undo markers, fixing the states to which we may backtrack via undo. Hence this feature provides for two functions which are specified in detail in the interaction specification below.

To specify which operations are undone upon one invocation of undo, we need to use the function `set_mark`. This function is typically used for the interaction specification of a feature with the undo feature.

```
Interface Undo
  undo      ::  m()
  set_mark  ::  m()

Feature UF implements Undo
  set_mark_UF(σ) ; undo_UF(σ)  = unit
  lift (put_σ x) ; undo_UF(σ)  = undo_UF(σ)
  y <- result_σ x; undo_UF(σ)  = undo_UF(σ); y <- result_σ x
  lift get_σ ; undo_UF(σ)      = undo_UF(σ); lift get_σ
```

The specification of undo requires some explanation. The first law states that `set_mark` stops the undo process. As the marker is eliminated as well, undo can be applied repeatedly. State changes, which are solely possible via `put` are simply undone, as specified in the next law. The relation of undo and other operations is more delicate, since we cannot undo a computed result. Thus, a concrete computed value remains unchanged by undo, as specified by the third law. In case the result of a `get` operation is not used, the last law states that the operation can be swapped. (Note that it is equally possible to remove such a `get` operation.) Another important detail is that the state access operations in the above specification are lifted. This is only to assure that `put` and `get` to not refer to the undo feature itself. Note also that we have not specified the effect of undo on an initial state.

This specification strongly relies on the above frame axiom stating that all operations can be reduced to the base operations `put`, `get` and `result`. Furthermore, `get` operations can be evaluated to a concrete `result` before the undo specification applies. Consider for instance a more delicate usage of undo in following program for $UF(CF)$:

$$\texttt{set\_mark}_{UF(CF)}\texttt{; put}_{CF}\texttt{ 5; y <- get}_{CF}\texttt{; undo}_{UF(CF)}\texttt{; result}_{UF(CF)}\texttt{ y}$$

The main point of this example is that the rules for undo cannot be applied directly. First, the computation to the left has to be evaluated appropriately. The resulting, equivalent program can be simplified easily with the above specification:

$$\begin{aligned}
&\texttt{set_mark; put}_{CF}\texttt{ 5; y <- result 5; undo ; result y}\\
=\ &\texttt{set_mark; put}_{CF}\texttt{ 5; undo; y <- result 5; result y}\\
=\ &\texttt{set_mark; undo; y <- result 5; result y}\\
=\ &\texttt{y <- result 5; result 5}
\end{aligned}$$

Observe that the above abstract specification of undo does not depend on any concrete state. It is essential that `put` does not compute a result, which is evident from the type of $\texttt{put}:: a \to m()$, as we cannot undo a computed result.

For the interaction specification with undo, we have to set undo markers with `set_mark`. This leads to another design question of undo. Shall we undo functions which do not affect any state? For instance, consider the following operation sequence:

$$\texttt{push x; y <- top; undo}$$

We may specify the interaction between stack and undo via these laws:

$$\begin{aligned}
&\texttt{push}_{UF(\sigma)} = \texttt{set_mark}_{UF(\sigma)}\texttt{ ; lift push}_{\sigma}\\
&\texttt{pop}_{UF(\sigma)} = \texttt{set_mark}_{UF(\sigma)}\texttt{ ; lift pop}_{\sigma}\\
&\texttt{top}_{UF(\sigma)} = \texttt{lift top}_{\sigma}\\
&\ldots
\end{aligned}$$

This specification sets an undo marker before each operation which changes the state, but not before the state readers. A specification which adheres more strictly to the informal one would also set undo markers before state readers.

It is instructive to show how we can verify the effect of undo. For the proof, we need the frame axiom for each other operation `f` we want to undo. This law expresses that the function `f` can be represented as a state change plus some output. Only this state change, which is done via `gets`, is undone, as an output cannot be revoked. To be more precise, the inner part of `f` is undone. Its specification for the undo context may include a call to `set_mark`, which is not undone by this law. (But bounds the effect of undo.)

Another question is whether this specification is actually consistent. We must assume for consistency that $\texttt{set\_mark}_{UF(\sigma)}$ and $\texttt{lift}\ f_{\sigma}$ are distinct state transformers. This is easy to achieve by assuming that `set_mark` modifies the local state of the undo feature.

3.6.2 A Generic Lock

With similar techniques as used in the last feature, we can now tackle the lock feature. The lock feature is intended to disallow any state change to the inner features. Using generic state transformers, this can be specified as follows:

```
Interface Lock
  lock   :: m()
  unlock :: m()

Feature LF implements Lock
  lock_LF(σ); lift (put_σ x) = lock_LF(σ)
  lock_LF(σ); lift (get_σ x) = lift (get_σ x) ; lock_LF(σ)
  lock_LF(σ); result x       = result x ; lock_LF(σ)
  lock_LF(σ); unlock_LF(σ)   = unit
```

As with the undo feature, this highly generic specification can only be explained using the frame axiom. This states that each operation of a feature can be represented using basic state transformers and state readers. (This argument still holds for functions which both write the state and compute a result.)

A drawback of this version of the lock operation is that it affects all inner features, with no exception. Alternatively, we could specify the effect for each feature individually. Note however, that this may easily lead to inconsistencies if one function of a feature can be modeled by another one and its behavior wrt lock is different.

3.7 Feature Combination and Refinement

The main objective of modular specifications is that specifications for individual features and interactions hold for feature combinations as well. Unfortunately, it is not always the case that the laws hold directly. Since functions can be redefined for feature composition and since the affected state varies, laws may not hold for some implementations. While interaction specifications add extra requirements, feature combination may also require refinements of feature specifications. For these reasons, we often need to use an abstraction of the state or to put extra conditions on specifications.

As an example, consider a specification for stack with a counter, i.e. the type $CF(SF)$. We have two feature specifications plus the interaction specification as follows:

- The specification of stacks must be shown for $CF(SF)$. For the redefinitions presented earlier, this is shown in Section 3.8.
- The interaction specification, given as

```
empty; size   = empty; result 0
push a; size  = x <- size; push a; result x+1
pop a; size   = x <- size; pop a; result x-1
```

can be verified in a similar fashion.

- The specification for the counter holds by construction. However, it is desirable to hide `inc` and `dec` to the outside. As hiding is an orthogonal issue, it is not considered here.

The specification of a composed object is obtained from the individual specifications plus the interaction specifications. Assume that some type σ has a feature F and that the laws for F hold. (In general, they may only be a refinement of the original specification of F.) Then, we add a feature F', which gives the type $F'(\sigma)$. By assumption, the laws for F' do hold, as specifications for feature constructors are schematic wrt the underlying features (here σ) and must hold for any implementation. As the functions of F are redefined for the type $F'(\sigma)$, we have to show that the laws for F hold (possibly in some refined way) for this type. Furthermore, we have to show that the interaction specification holds.

We can define refinements via logical implication. A specification R is a refinement of a specification S, if R implies S. A typical example of refinement is an implementation of stacks, given via equations as shown in Chapter 5. For refinement, we must show that these equations imply the stack specification. Such data type refinements are well examined problems [Hoa72, Slo97]. For an abstract view of refinement see e.g. [BS].

In our case, laws using implicit state hold under feature combination in many cases. This advantage of our specification technique is due to careful avoidance of overspecification. For instance, the specification for `push` and `pop` of the stack feature holds under adding the counter CF, although the stack operations have been redefined. Some cases where this is not the case will be examined and classified below.

As we aim for feature composition, we want to show that a specification which holds for some feature combination still holds in case another feature is added. Since the additional feature is in general unrelated, it is often required to abstract from the details of other features and establish the specification under an abstraction of the object state. Since we use equational specifications with implicit state, it is convenient to restrict the equality instead of applying abstraction functions to the object state. Furthermore, our type system can be used to determine the needed abstractions in a simple way. This is possible if the abstraction function ignores the state of particular features.

More concretely, specifications consist of equations of the form `op_seq = op_seq'`. Since this equality depends on the actual type, we can write the types explicitly as follows:

$$\texttt{op\_seq}_\sigma =_\sigma \texttt{op\_seq'}_\sigma$$

We use in the following a restricted equality, which only compares the state changes for selected features. For instance, for $\sigma = F(\delta)$, we can ignore the state used for feature F via the equality

$$\texttt{op\_seq}_{F(\delta)} =_\delta \texttt{op\_seq1}_{F(\delta)}$$

We can illustrate the above equations by making the implicit state explicit via the function `run`:

$$\forall\ \texttt{x} :: \mathcal{T}ype(\delta).(\texttt{run op\_seq}_\delta\ \texttt{x}) == (\texttt{run op\_seq'}_\delta\ \texttt{x})$$

As the two operation sequences are state transformers, we can just apply appropriate parameters (initial values of the state) and quantify over them. With explicit state it is possible to define abstractions which pick only certain features. For instance, with an appropriate abstraction function `abs` of type $\mathcal{T}ype(F(\delta)) \to \mathcal{T}ype(\delta)$, we can write the above equation with $=_\delta$ in a more explicit form:

$$\forall\ \texttt{x} :: \mathcal{T}ype(F(\delta)).\texttt{abs(run op\_seq}_{F(\delta)}\ \texttt{x}) == \texttt{abs(run op\_seq1}_{F(\delta)}\ \texttt{x)}$$

Furthermore, we can put conditions on the the initial state or on the (outside visible) parameters of the operations. This leads to the issue of conditional refinements.

Whereas refinement usually means reducing underspecification, conditional refinement may also entail restrictions on the usage of some object. As an abstract notion of conditional refinement (see e.g. [Stø96]), we can write $C \Rightarrow (R \Rightarrow S)$. We will see later that it is often difficult to formalize the conditions. For instance, how can we require that a (black-box) procedure does not invoke an internal message more than ten times? Since we do not know the internals of the procedure, we cannot talk about its internal behavior. A solution to this problem will be presented in Section 4.3 on exceptions.

In the following sections, we present some typical classes of refinements in more details.

3.7.1 Behavioral Refinement

In the simplest case, the laws of one feature hold for the particular combination. Consider for instance the stack with counter combination with an implementation where the stack operations increment or decrement the counter. Then even the law

$$\texttt{push}_{CF(SF(\sigma))}\ \texttt{x; pop}_{CF(SF(\sigma))}\ \texttt{= unit}_{CF(SF(\sigma))}$$

holds, although the manipulated state is different. As only the equality of the resulting states is required, some extensions may conflict with this specification. The problem is that this law for stacks requires that a sequence of `push` and `pop` is the identity on the state. Such strong statements about the state only hold under redefinitions if two features are independent or behave in a synchronized way, like the counter. If this is not the case, we show another common refinement in the next section which is more flexible wrt the used state.

In general, behavioral refinement is a quite strong property. It depends heavily on the specification style. For instance, the specification of the stack via

```
push x; pop; top = y <- top; push x; pop; result y
```

is tailored for behavioral refinement. As an example consider a feature counting the number of invoked operations:

```
Interface OpCounter:
  ops      ::  m Int
  inc_op   ::  m ()
  reset; ops   = reset; result 0
  inc_op; ops  = x <- ops; inc_op; result x+1

Feature OP implements OpCounter state o :: Int

  push_OP(σ) x = inc_op_OP(σ) ; lift push_σ x
  pop_OP(σ) x  = inc_op_OP(σ) ; lift pop_σ x
  ...
```

For this new feature, behavioral refinement holds for the latter specification (if the laws hold for σ):

```
push_OP(σ) x; pop_OP(σ); top_OP(σ) =
y <- top_OP(σ); push_OP(σ) x; pop_OP(σ); result y
```

Note that there are still feature combinations which do not fulfill this law (e.g. the lock). These will be discussed in the following sections.

In general, behavioral specifications have the advantage that they do not refer directly to a particular notion of state. Since behavioral specifications only relate operations of one feature, they do not state anything about extra operations.

Many other specification styles [DL97, MS97] (Hoare calculi and predicate transformers) refer to the state change in a direct way. Such specifications have to be "updated" in case extra features add state which may also be manipulated if an operation is redefined.

3.7.2 Weak Behavioral Refinement

Weak refinement covers the case when laws only hold with a restricted equality, which ignores the state of some additional features. This is a particular case of an abstraction, which is easy to handle via types (as shown above) and is quite typical in object-oriented systems. For instance, assume an implementation for undo which copies the local state before each operation. Then we have for stack with undo:

```
push_UF(SF(σ)) x; pop_UF(SF(σ)) =_SF(σ) unit_UF(SF(σ))
```

This means that the equation only holds for the state change of the stack variables (the integer list). Another, very similar example is the same law with $OP(SF)$ instead of $UF(SF)$.

In general, weak refinement applies when features add extra functionality which may invalidate old laws on the newly added state. Weak refinement is important for object-oriented design, as it allows for semantic subtyping where behavioral refinement does not hold (as discussed in Section 3.9). The goal is that an object with the new features can be viewed as a subtype of stack and can be used instead.

3.7.3 Conditional Refinement: A Bounded Stack

Conditional refinements cover cases where an abstraction on the state does not suffice. Instead, some conditions on the environment must be fulfilled for the laws to hold. By environment we mean the user of the operations. For instance, in the case of the bounded stack, we must assume that the elements we push into the stack are below the size in order for the laws to hold. Hence we need extra conditions on the function parameters.

An alternative case is a small stack, which is bounded to a fixed number of elements. Then we need extra constraints on the initial state for the equations to hold. Conditional refinement is often used for the transition of a more abstract model to a concrete one with limited resources or capabilities [Stø96].

In general, conditional refinements may constrain the following:

- The state before the execution of an operation of some feature. For instance, for the lock feature, the other feature specifications hold if the initial state is unlocked and the lock is not used. In general, this leads to invariants, as discussed in Section 4.1.1.

- The parameters with which the functions of the feature are invoked. For instance, a bounded stack may only be able to insert elements below a certain size.

- The allowed function calls may be restricted. For instance, the small stack discussed above may only be able to hold ten elements. Hence a program must not add more that ten elements in order to work properly. In the general case, conditions may disallow certain operation sequences.

Consider for instance the bounded stack, where the interaction requires to add conditions to the stack rules as follows.

```
check_bound x == True => push x; pop = unit
check_bound x == True => push x; top = push x; result x
check_bound x == True => push x; empty = empty
```

We will discuss this in detail in Section 4.3 and show that it is often easier to model conditional refinement via abstraction over exceptions.

Another typical case are conditions on the state. Consider for instance the example of a small stack with just ten elements. This is more complicated to formulate. Similar to the above feature, we assume a feature `SizeBound` with an operation `check_size` to check the size of the stack. A simple approach which reflects the form of the above conditional refinement rules is the following:

```
check_size == True => push x; pop = unit
check_size == True => push x; top = push x; result x
check_size == True => push x; empty = empty
```

These laws model a constraint on the implicit state, whereas the above is a constraint on the parameters. The general form of such a refinement is to read the local feature state and then to check the condition on the state. This however obscures the relation to the original specification. We show in Section 4.3 that exceptions give a simpler and technically more elegant treatment of conditions.

The general disadvantage of conditional refinement is that it does not create a subtype relation in general, only a conditional subtype, which has to be verified for each usage. For instance, if some program works correctly on stacks, we cannot conclude anything about its behavior on a bounded stack. This can be relieved in many cases by the use of exceptions for specifications, as discussed in Section 4.3. The idea is to raise an exception in case the bound check does not hold. In this case, the above equations hold if no exception occurs.

Note that there are cases where we need more than one of the above refinement principles. For instance, we may need conditional refinement with additional state abstractions. Such combinations are quite straightforward as the principles of conditional and (weak) behavioral refinements complement each other.

3.7.4 Structural Refinements which Require Abstractions

There are cases where an added feature changes the internal structure of a state-based implementation, on which a specification relies upon. This usually requires abstraction functions to relate two state-based specifications. An example is a special-purpose counter based on two integer variables designed for the stack feature. Via the interaction specifications, it counts the number of `push` and `pop` operations, respectively. The actual size of the stack is obtained by subtracting the two counters. Hence the operations `push` and `pop` do not cancel each other (wrt the counter).

In general, we need abstraction functions on state-based specifications to account for a new implementation. This is not formalized here, as such refinements are well examined [DL97, LW93, MS97]. Abstraction functions are a well developed and powerful tool for showing an implementation relation, typically for some abstract data types [Hoa72, GTW79, Ehr82, Slo97].

We argue however that this expressiveness to model structural change is not desirable for object composition. There are several reasons for this. Often, state-independent specifications on the interface level can be used instead, which avoid these

problems. Secondly, we claim that for inheritance or feature combination the need for structural abstractions is not an indication of good design. In most of these examples in the literature, ad-hoc redefinitions are used in order to reuse some object. In addition, abstraction functions require some technical overhead, particularly if refinements are used repeatedly. For these reasons, we restrict our attention to cases where our simpler techniques suffice.

Note further that the need for abstraction functions can also be an indication for an inappropriate specification. Consider the example of a counter `CostCount` which is used to count the cost of the used operations. For instance, operations like `push`, `pop` or `check_bound` have cost one. In case we specify that an operation `pop` increases the cost by one, then this does not hold if the bound check is added. The reason is that we only abstract over feature states and cannot "undo" this extra cost. This could be remedied by an abstraction function which recalculates appropriately. We consider this however as a specification problem and not a limitation of our techniques. A solution is to specify the cost increase in a more flexible way by a cost function which can be redefined in other features as well.

3.7.5 Combination of Refinements

When building complex objects, features are added repeatedly. In this case, it laws of one feature have to be refined repeatedly. Since refinement is just logical implication here, repeated refinements are obtained via logical conjunction. This applies also for conditional refinements, where conditions have to be conjoined.

For instance, consider the equation $\mathtt{op} = \mathtt{op'}$ which holds for some feature F. In order to add features F' and F'' independent of each other, we may show the law of feature F for the following types: for $\mathtt{op}_{F'(F)} = \mathtt{op'}_{F'(F)}$ and $\mathtt{op}_{F''(F)} = \mathtt{op'}_{F''(F)}$. Now assume we want to add both features at the same time. However, from the last two equations, we cannot conclude anything about $\mathtt{op}_{F''(F'(F))} = \mathtt{op'}_{F''(F'(F))}$.

To avoid this problem, it is important that refinement relations are established in a slightly more general way. The equation must be shown for two more general types, i.e.

$$\mathtt{op}_{F'(\sigma)} = \mathtt{op'}_{F'(\sigma)}$$

and

$$\mathtt{op}_{F''(\sigma)} = \mathtt{op'}_{F''(\sigma)}.$$

Then we can conclude

$$\mathtt{op}_{F''(F'(\sigma))} = \mathtt{op'}_{F''(F'(\sigma))}$$

as well. Since the above two equations hold for any σ, we can obtain the result by appropriate instantiation of σ. (In general, unification of type terms is needed.) The same line of reasoning holds for weak behavioral refinement, where the equality is restricted. This abstract way of representing interactions is also important for the programming level (see Chapters 5 and Chapters 6) as the interaction treatment is composed in the same way. This is essential for flexible feature combination.

3.8 Formal Reasoning about Monadic Programs

In this section, we discuss formal reasoning about program sequences in our setting. This includes proving properties (or refinements) of composed objects and of implementations. Our specification style builds upon equational reasoning about operation sequences. For this to be practical, it is essential that we can reshuffle operations which do not affect each other. As we partition objects into features, this is often simple, but can be involved in general.

Consider a simple but typical example: counter plus stack. Assume the `push` function of the counter is redefined as shown in Section 3.5, and `pop` accordingly. For this implementation, we show the law `push x; pop = unit` for $CF(SF)$ via equational reasoning and reshuffling of operations. The general proof strategy is to rearrange the commands such that laws apply. In the first step, we unfold the definitions and then reorder operations of different features. Next, we apply the rule for the counter. To simplify the stack operations, we first use the law for lift and push the lift operation outside.

$$
\begin{aligned}
&\texttt{push}_{CF(SF)}\ \texttt{x; pop}_{CF(SF)} \\
=\ &\texttt{inc}_{CF(SF)}\texttt{; lift push}_{SF}\ \texttt{x; dec}_{CF(SF)}\ \texttt{; lift pop}_{SF} \\
=\ &\texttt{inc}_{CF(SF)}\texttt{; dec}_{CF(SF)}\ \texttt{; lift push}_{SF}\ \texttt{x; lift pop}_{SF} \\
=\ &\texttt{lift push}_{SF}\ \texttt{x; lift pop}_{SF} \\
=\ &\texttt{lift (push}_{SF}\ \texttt{x; pop}_{SF}\texttt{)} \\
=\ &\texttt{lift unit}_{SF} \\
=\ &\texttt{unit}_{CF(SF)}
\end{aligned}
$$

The reordering of stack and counter operations in the above proof is possible since the two operations may only affect different parts of the state. The important observation about this proof is that we do not need to look at state. For reordering, we only use abstract results about which state may be affected. In this sense, we hope that this proof strategy is more abstract and hence scalable to larger and more involved systems.

The general design is to shift the state readers, which produce values, to the left which allows concrete evaluation. This should of course be reflected in the specification rules/styles, not just in the verification techniques. In general, there are however some obstacles to reshuffling (which clearly appear in other approaches as well). Note that it is necessary for several rules to insert extra state readers. For instance, most state-oriented specification start with an operation reading the state. Since state readers do not change the state this causes no problems. As a consequence, it is advisable for feature specification to separate state readers and modifiers.

3.9 Semantic Subtyping

In the treatment on refinement, we have aimed at lifting properties of features to feature combinations. Semantic subtyping is a related, but slightly stronger property. In our setting, adding new features to a particular combination of features creates a subtype. The idea of subtyping is that a subtype object can be used instead of an object of the supertype.[2] In object-oriented programming, subclasses usually create subtypes, which is a main ingredient of many object-oriented languages. In addition to the syntactic subtype relation, it is desirable to establish semantic properties as well. If we can show that a program has certain properties, we want the same result in case it is used with an object of a subtype.[3] Thus the difference to the last section on refinement is that we are not only interested in properties of an extended object, but also of arbitrary programs which use the extended object. In other words, we not only establish properties for an extended object, but any properties shown via these laws. We will see that in many cases only weaker results are possible, similar to weak behavioral refinement.

Consider for instance an operation sequence on a stack object for which some properties have been shown using the stack specification. If an object with additional features is used instead, weak behavioral refinement allows one to recover the properties for a restricted equality. More formally, assume a procedure `f`, of which we only know a certain property. Internally, the function consists of some operation sequence $\texttt{f = op}_{SF}$ on an object of type SF. Assume we have the following property: $\texttt{f} =_{SF} \texttt{op1}_{SF}$. If `f` is run on a subtype object of type $OP(SF)$, we want to infer $\texttt{op}_{OP(SF)} =_{SF} \texttt{op1}_{OP(SF)}$. This is all we can expect for this case, as the internals of the operation `op` are not known. In this case, the above equation for `f` specifies its effect on the stack, but we do not know how many operations were used. Thus, we cannot conclude anything about the operation counter `OP` from the abstract specification.

In the following, we will establish a meta-result about "lifting" properties of an operation sequence for the form

$$\texttt{op}_{\sigma} =_{\sigma} \texttt{op1}_{\sigma}$$

to

$$\texttt{op}_{F(\sigma)} =_{\sigma} \texttt{op1}_{F(\sigma)}.$$

For formal reasoning, it is important to realize that we do not use a dedicated, special logical calculus but instead embed feature-oriented programs into (higher-order) logic. For this reason, we cannot give such a generic result without further assumptions, as a proof of the former result may use details of the particular setting about which we cannot reason generically. Thus we will assume that the operation sequences `op` and `op1` only use the declared operations of the features. We must assume that the proof of the initial property consists of a chain of equalities of the form $\texttt{op} =_{\sigma} \ldots =_{\sigma} \texttt{op1}$.

[2] This is discussed for parameter passing in Section 4.4.

[3] We do not talk about procedures with objects as parameters here, which is discussed in Section 4.2.

With this restriction, it seems at first glance that all we need for lifting the property of `f` is (weak) behavioral refinement for the used laws. There is however a further problem. In a proof, as for instance shown in Section 3.8, one often has to reorder operations in order to apply laws. This is in general not possible if a feature is added which redefines operations. As an example consider for instance a stack with counter and undo, i.e. `UF(CF(SF))`. For this object, the reorderings shown in Section 3.8 do not apply as

$$\texttt{inc}_{UF(CF(SF))}\texttt{;}\ \texttt{pop}_{UF(CF(SF))} \neq \texttt{pop}_{UF(CF(SF))}\texttt{;}\ \texttt{inc}_{UF(CF(SF))}$$

The reason is that undo behaves differently after these two operation sequences. Note that this reordering is possible for `CF(SF)`. Therefore, we cannot simply establish a generic result for semantic subtyping from (weak) behavioral refinement. In this example, we can still establish the result, since the reordering holds under abstraction to $CF(SF)$, but this does not hold in general.

Thus we follow a similar, but different line of reasoning as in the existing results [DL97, LW93]. The idea is to use a simple restriction on method redefinitions which is quite reasonable for practical purposes.

Definition 3.9.1 A method redefinition of a function $\texttt{f}_\sigma$ for $\texttt{f}_{F(\sigma)}$ is called *conservative*, if

$$\texttt{f}_{F(\sigma)} =_\sigma \texttt{lift f}_\sigma.$$

A conservative redefinition can add new behavior, but cannot alter the inner behavior. Almost all redefinitions considered so far are conservative. Clearly, there are cases of non-conservative redefinitions, where behavioral refinement holds. If we redefine the stack operations push and pop to add/remove an element twice (similar to Section 3.7.4), then the abstract specification of stacks still holds. This means that the original data structure is used in a different, but compatible way. Since such redefinitions neither appear to be frequent nor seem to be advisable, we limit our attention to conservative redefinitions. There is however one reasonable exception to conservative redefinitions. A method redefinition can use its pendant in the superclass in a black box fashion as is or not at all. For instance, in case of the bounded stack, a pop operation is not executed in some cases. Such cases can however be handled with exceptions, as shown later in Section 4.3.

Theorem 3.9.2 *Assume two operation sequences* `op` *and* `op1` *and assume further*

$$\texttt{op}_\sigma =_\sigma \texttt{op1}_\sigma$$

holds. If the sequences only use declared operations of the features in σ which are conservatively redefined in $F(\sigma)$, then

$$\texttt{op}_{F(\sigma)} =_\sigma \texttt{op1}_{F(\sigma)}$$

Proof Since by assumption `op = op1`, we can conclude from the conservative redefinitions that both operations terminate for the type $F(\sigma)$. We assume

$$\texttt{op}_{F(\sigma)} = \texttt{op}^1_{F(\sigma)}\,;\ldots;\ \texttt{op}^n_{F(\sigma)}$$

and

$$\texttt{op}_{F(\sigma)} = \texttt{op1}^1_{F(\sigma)}\,;\ldots;\ \texttt{op1}^n_{F(\sigma)}.$$

With the property of conservative extension, we can show:

$$\begin{array}{lll}
\texttt{op}_{F(\sigma)} & =_\sigma & \\
\texttt{op}^1_{F(\sigma)};\ldots;\ \texttt{op}^n_{F(\sigma)} & =_\sigma & \\
\texttt{op}^1_{F(\sigma)};\ldots;\ \texttt{lift op}^n_\sigma & =_\sigma & \\
 & & \text{(by induction)} \\
\texttt{lift op}^1_\sigma;\ldots;\ \texttt{lift op}^n_\sigma & =_\sigma & \\
 & & \text{(properties of lift, induction)} \\
\texttt{lift (op}^1_\sigma;\ldots;\ \texttt{op}^n_\sigma\texttt{)} & =_\sigma & \\
\texttt{lift (op1}^1_\sigma;\ldots;\ \texttt{op1}^n_\sigma\texttt{)} & =_\sigma & \\
\texttt{lift op1}^1_\sigma;\ldots;\ \texttt{lift op1}^n_\sigma & =_\sigma & \\
\texttt{op1}_{F(\sigma)} & &
\end{array}$$

□

Note that this result also entails weak behavioral refinement as a special case. For weak behavioral refinement, we only have to apply the theorem to lift the laws of σ. It is straightforward to extend the last definition and the above result to combinations of several features. For instance, instead of just F, we may want to add two features F_1 and F_2. The technical treatment for this case proceeds similar to the above. We will give extended versions of these results which address the handling of object networks in Section 4.2. A further novel extension to subtyping under exceptions is shown in Section 4.3.1.

The above result is similar to the condition for semantic subtyping in [DL97, LW93, Ame91], where it is expressed with an explicit abstraction function in terms of pre- and postconditions of methods. In this version, a method redefinition in a superclass may not follow the superclass implementation, it only has to fulfill the postcondition assuming the precondition holds. Our requirement is stronger, as we require the effect on the (part of the) state to be identical. If it is only required that a postcondition holds, this may leave choices for an implementation. With this definition, a subtype may produce a different resulting state, as long as it fulfills the postcondition. In addition, we do not consider arbitrary abstractions but construct them from the type system.

The main advantage is that our definition is independent of actual specifications. Hence, as we show below, the generated subtype relation does not depend on the

specification. The notion of conservative redefinitions only depends on the redefinition. Other approaches build a relation between specifications for some abstraction function. For this reason, we claim that our notion of conservative redefinitions is practical and easy to use.

Another difference is that our setting is more flexible, as will be evident when we treat subtyping for object models in Section 4.2.2 and exceptions as refinements in Section 4.3. We may also argue that the technical treatment is simpler, as the development in [DL97] is quite technical.

3.10 Dependencies between Features

In the previous examples, each feature could be used independently. In many examples it is however useful to write a feature under the assumption that some other feature is available. For this purpose, a feature declaration may require other features. As an example consider a display adapter `DisplayAdapter` feature which adapts the output of an `AsciiAdapter` to a graphic display. In the following example, we show the `uses` condition for a constructor `DA` of the former feature:

```
Interface AsciiAdapter
  get_text ::  m(String)

Feature AA implements AsciiAdapter
  get_text = ...

Interface DisplayAdapter
  show_in_window :: m()

Feature DA implements DisplayAdapter uses AsciiAdapter
  show_in_window = ...; s <- get_text; ...
```

The `uses` clause states that we can only add the constructor `DA` to feature combinations which already provide for the `AsciiAdapter` feature. Thus, $DA(SF)$ is an invalid combination, in contrast to $DA(AA(SF))$. In this way, an implementation may use the operations provided by the feature called `AsciiAdapter` in order to produce output on a window system.

In general, the base functionality of a new feature can rely on the functionality of the required ones. This idea of assuming other features is similar to import relations of module concepts. It is however novel for class or object composition concepts.

We have shown the `uses` construct on the feature constructor level. In some cases, it may be interesting to define this on the interface level. This is possible in the same fashion and establishes the uses relation for each implementing constructor.

It is straightforward to extend the frame assumption for the `uses` construction. In this case, we have to state that functions of the new feature can also modify the state

of the other. We will later show a more refined model, where it is also possible to express that a feature only reads from the the other state, but does not write on it.

3.11 Parametric Features

For reusability, it is often desirable to parameterize a feature by a type. In this section, we introduce parametric features and parameterized specifications. Parameterization is a well examined concept which is present in many programming languages. In object-oriented ones, one often uses parametric classes, as for instance in Pizza [OW97], which is used in Chapter 6. A similar mechanism for type parameters, but without elaborate subtyping concepts, are C++ templates [Str91]. Due to the flexible composition concepts for features, we also need expressive type concepts for composition. This problem is more pronounced on the programming language level, as shown in Chapter 6.

For parametric features, we simply add a type parameter to the interface and the constructor, which can then be used to assign appropriate types to the member functions. An example specification for stacks with a type parameter `a` is as follows.

```
Interface Stack a
  push     :: a → m ()
  pop      :: m ()
  top      :: m(a)
  empty    :: m ()
  is_empty :: m(Bool)
```

Similarly, we introduce parametric feature constructors:

```
Feature SF a implements Stack a
  state list :: [a]
  ...
```

In this way, we gain flexibility in a different dimension, compared to feature composition. Data types such as stacks can be used with arbitrary element types. It is straightforward to extend this concept to several type parameters. For instance, an interface for an object which stores a pair of two elements can be defined as follows:

```
Interface Pair a b
  pair   :: a → b → m ()
  first  :: m(a)
  snd    :: m(b)
```

For consistency, we require that an interface and the corresponding constructor must have the same number of type arguments, and, in addition, the two argument lists must be identical for an implementation statement as above. Our notation for object types extends in the canonical way to parametric constructors, e.g. $CT(SF\ Int\ (Id))$.

For brevity, we sometimes omit type parameters if not needed in the context. For instance, we write just $\texttt{f} =_{SF} \texttt{f'}$.

For specifying interactions between parametric features we may have to be careful about the parameter types. As shown in Sections 5.5 and 6.3, the type parameters have to be specified carefully for feature interactions. Often the type parameters have to be compatible in some way. In some cases, type parameters may be subject of an interaction specification.

3.12 Virtual Functions and Self Type

We present in this section a simple model of virtual functions and show modular specifications for virtual functions. Virtual functions are a main ingredient of most object-oriented languages. The idea of virtual functions is to bind calls to a function within a function definition of some (sub-)class dynamically. Such a call refers to the function associated to the composed object at run time, not to the one of the local feature/subclass. The problem is hence that we want to specify an individual feature, but do not know anything about virtual functions bound at run time. Only for a concrete object whose type we know, a full specification is possible. Since this impedes our goal of modular feature specifications, we show in the following specification methods for virtuals.

To account for virtuals, we just add another type parameter to each feature and to each of its functions which stands for the type of the full object.[4] When creating an object, this type variable must be instantiated appropriately. Note that obviously the self type parameter has to be identical for all feature (constructors) of one object. Thus we index every function with two parameters in the following.

Consider extending the stack feature by a function `push2`, which pushes an element twice on the stack. This allows for the following implementation of a "virtual" function `push2`:

$$\texttt{push2}_{SF(\sigma),\delta}\ \texttt{a} = \texttt{push}_{\delta,\delta}\ \texttt{a;}\ \texttt{push}_{\delta,\delta}\ \texttt{a}$$

The motivation for the use of a virtual function is reuse, as `push2` does not have to be redefined in the presence of other features. Further examples, such as an equality function, will be presented in the following chapter, as they require more than one object. Observe that the second type parameter δ, standing for the type of the global object, is interpreted for refinements like any other type parameters.

The novel point regarding specifications is that a feature can only be specified by using some assumptions on the (still unknown) object of type δ. In this example, we need to argue about an unknown function $\texttt{push}_{\delta,\delta}$ in order to specify `push2`. In our framework this is possible by requiring that laws also hold for type δ. This assumption can be discharged when composing an object.

[4] In Section 4.1, a slightly different formalism will be used for this, as typed objects identifiers are used.

As an example for modular specifications, we show how to specify `push2` in terms of $\texttt{push}_{SF(\sigma),SF(\sigma)}$, which is local to the feature. Our line of reasoning is instructive: to specify the stack feature separately, we need to assume

$$\texttt{push2}_{SF(\sigma),\delta}\ \texttt{a} =_{SF(\sigma)} \texttt{push}_{SF(\sigma),SF(\sigma)}\ \texttt{a;}\ \texttt{push}_{SF(\sigma),SF(\sigma)}\ \texttt{a}$$

This allows one to argue about a virtual function $\texttt{push2}_{SF(\sigma),\delta}$ in terms of $SF(\sigma)$. Informally, the above assumption states that no subclass can redefine the basic effect of `push2`; only extra behavior can be added.

Although virtual functions are generally acclaimed for fostering reuse, there are many cases where simple inheritance of virtual functions without adaptation does no suffice. For instance, it is unclear if `push2` behaves as expected if undo functionality is added. In case of undo, an inherited `push2` operation is not undone as expected, only one of its push operation is.

Another disadvantage is that capturing the full effect of virtual functions is more involved. Since a virtual function depends on all used features at run time, all other features may be used. Hence it is only possible to limit the state affected for particular feature combinations. This is shown in more detail in the following sections.

Interestingly, our model for virtual functions easily accommodates a convenient solution to one of the major typing problems caused by virtual functions. It is known as the problem of binary methods [BCC+96] and is related to the problem whether inheritance should create subtypes [LW90, DL97, Ame90, LW93]. The solution to the problem which we model here was suggested by Bruce [BSG95] (see also [AC95]). The idea is to use δ for what is usually called MyType or "This type" (see for instance Eiffel [Mey92], which uses the name "like Current"). We will investigate this issue in the following chapter, as we need at least two objects for it to occur.

Chapter 4

Specification of Object Networks

Whereas the last chapter has focused on the specification of a single object with several services, we now consider the case of a larger system with a set of objects. The reason is two-fold. First, we want to specify objects which interact with others and also components which consist of a set of objects. Furthermore, we want to show that the techniques developed so far can be extended in order to specify object networks. Although this is in general significantly more difficult than specifying individual components, we can claim several new contributions.

In the following, we add object identifiers and talk about object networks. As well known, reasoning about structures with object references is quite involved, see for instance [PH97]. We show that the extension to reference structures is quite canonical. An interesting contribution is the viewpoint of a two-dimensional object store, spawned by object identifiers and features. We show abstract reasoning about homogenous object structures. The main results of this chapter include a result for semantic subtyping for homogeneous structures and the idea of abstracting over exceptions as refinement. The latter is possible for the single object case (as shown in Chapter 5) in a similar fashion.

We pursue in the following our specification style with equations between program statements. As this can be tedious in some cases, particularly for specifying invariants on pointer structures, we employ expressive specification techniques in the usual pre- and postcondition style with quantified formulas later. This includes a discussion on how the two styles integrate technically and methodically. Our treatment of (non-executable) pre- and postconditions, which may include quantifiers, is largely in the lines of current literature [PH97], but in a different setting.

Another contribution is a simple type system for the basic constructs of object-oriented languages. In particular, we do not need an impredicative type system such as f-bounded polymorphism [BTCGS91, OW97] to provide for the common features of object-oriented systems. (To be precise, we only use matching, not full subtyping, as discussed in Section 4.4. This is also used in languages like Java [GJS96].)

Although the feature model is more general, it gives a functional model of a (core) object-oriented language, based on monadic state transforming functions. This includes

inheritance, virtual functions, object identifiers with references and exceptions. We also model so-called binary functions, which, roughly speaking, can only be applied to two objects of the same type. We do however assume that functions have a type which does not change under inheritance, except for some late binding of type parameters used for virtual and binary functions. As argued in [BPF97], this is sufficient in practice. Hence we do not cover the well examined problems of arbitrary co- or contravariance (as discussed in Section 4.4), which means that the signature of a function may change under inheritance/subtyping. (More formally, the problem is to extend the subtype relation to function types.) Although semantically problematic [Cas95] wrt subtyping, in some languages (e.g.[Mey92]) functions can require parameters of more specific type in subtypes.

The following presentation is in the lines of the earlier treatment. The object model is tailored towards existing object-oriented languages. We include object references, but do not include null pointers for simplicity.

4.1 A Functional Object Model

We will show in the following the extensions for a functional model of object-oriented systems. We extend and revise the basic definitions as follows:

- A set of feature constructors $\mathcal{F}$, and, correspondingly, a set of feature interfaces $\mathcal{I}$, where each $F_i \in \mathcal{F}$ is associated with one $I_i \in \mathcal{I}$.
- A set of object types $\mathcal{O}types$, which is generated by feature constructors as follows: $F_n(F_{n-1}(\ldots F_1)\ldots) \in \mathcal{O}types$, where $F_i \in \mathcal{F}$, $i > 0$, and all F_i are distinct. A type $F_n(F_{n-1}(\ldots F_1)\ldots)$ implements all corresponding interfaces I_i, i.e. $F_n(F_{n-1}(\ldots F_1)\ldots) :: I_i$ for all $i = 1, \ldots, n$.
- A set of typed object ids oid_σ, where $\sigma \in \mathcal{O}types$. We write $oid_\sigma :: I$ if $\sigma :: I$ and $oid_{\sigma:I}$ to stand for any oid_σ with $oid_\sigma :: I$.
- A set of types $\mathcal{T}$ generated by base types $(Int, Bool, \ldots)$, oid_σ, and the usual tuple constructors $((\_, \ldots, \_))$ and projections π_i.
- A mapping $\mathcal{T}ype :: \mathcal{F} \to \mathcal{T}$ which describes the state used by a feature. As we assume that the type of the state used by objects is the tuple of the individual feature state, this mapping extends to object types and typed ids in the canonical way: $\mathcal{T}ype :: oid_\sigma \to \mathcal{T}$ with

$$\mathcal{T}ype(id_{F_n(F_{n-1}(\ldots F_1)\ldots)}) = (\mathcal{T}ype(F_n), \ldots, \mathcal{T}ype(F_1)).$$

Formalizing the last mapping requires dependent types or other expressive type systems, as the type of the object is an implicit parameter which determines the result. Note that this is generally the case for other imperative models with object identifiers.

Whereas in the last section operations were modeled as state transformers on appropriate values, we simply assume there is a global object store which is transformed. The object store includes a finite set of used objects and can be extended via the function `new`. As we aim for abstract specifications, we do not give a concrete model for such a store, which can be found in Chapter 5. Roughly speaking, we can assume an underlying monad of the form $m\ a = ostore \to (ostore, a)$, where *ostore* is the type of the global object store. Thus the function `run` invokes an operation with an initial object store *os* and returns a pair of the computed value `val` and the updated object store.

Next, we have to clarify the meaning of equality for an object model. The point is that equality between two computations refers to all objects which exist after the execution of the two computations. If two computations result with two different sets of objects, their states are incomparable. Clearly, equality assumes that both computations are invoked with the same set of objects. For instance, `x <- new(SF) = unit` does not hold, since the number of alive objects on both sides differ. Note that locally bound variables, here x, are immutable, which often simplifies reasoning about programs.

The definition of the object store is based on the idea that we can treat the access to the state via base access features. Therefore, the operations `get`, `put`, and `new` are supplied by some dedicated features. Unlike user defined features, these use dependent types as the type of `get` and `put` depends on the (type of) the used identifier. Furthermore, these operations are the only ones allowed to access the state. Other features must state that they use the base feature. To allow a fine grained control, we separate these access functions into three base features. The main advantage of this viewpoint is that we can use these features to give other features access rights to state. Furthermore, we can distinguish features which create new objects and those which do not.

The base features of the object store are as follows:

```
Interface StoreReader
   get  :: oid_σ  →  m(Type(σ))

Feature SR implements StoreReader

Interface StoreWriter uses StoreReader
   put  :: oid_σ  →  m(Type(σ))

   put x d           => get x = result d
   x ≠ y             => put x c; put y d = put y d; put x c
   put x c; put x d  =  put x d

Feature SW implements StoreWriter
```

```
Interface StoreCreater
   new :: σ → m(oid_σ)
   y <- new_σ => result x ≠ y = result True
   x <- new_σ; y <- new_σ = y <- new_σ; x <- new_σ

Feature SC implements StoreCreater
```

Concrete implementations of the above functionality can be found in Chapter 5.

For the interaction between the `StoreCreator` and the reader/writer, we have the following equations:

```
   y <- new_σ; put x c     =  put x c; y <- new_σ
   y <- new_σ; z <- get x  =  z <- get x; y <- new_σ
```

In the above, `y` is new bound variable and is hence different from `x`. We assume that the feature combination $SR(SW(SC))$ is omnipresent as the base feature in any feature combination, similar to the base feature Id in earlier sections. For convenience, we generally omit these and write for instance SF instead of $SF(SR(SW(SC))$. Note that we do not explicitly specify valid object references. We generally assume that all used object identifiers are valid. This is possible, as `new` is the only way to create elements of the type *oid*.

In contrast to the last section on virtual functions, we assume that object ids are typed and do not mark operations with two type parameters. By convention, the first parameter of every function is the object it works on (which is often called `self`.) In general, our model is similar to multi-methods, as described in [CL94a].

Furthermore, we parameterize not only object identifiers but also features with the type of self. These have to be instantiated appropriately when creating objects via `new`. This is needed to formalize the possible effect of virtual functions in a feature on the level of features. In case a feature does not use virtuals or does not have other type parameters, we may omit this type for brevity.

For instance, the stack feature with a virtual function `push2` is formalized with δ, standing for the type of self, as follows:

```
Interface Stack δ
  push     :: oid_δ → Int → m ()
  pop      :: oid_δ → m(Int)
  empty    :: oid_δ → m()
  is_empty :: oid_δ → m(Bool)
  push2    :: oid_δ → Int → m ()

  push2 self_δ a = push self_δ a; push self_δ a

Feature SF δ implements Stack δ
  push self_δ ; pop self_δ = unit
  ...
```

Similarly, as shown above, we parameterize feature constructors. We annotate the functions provided by features with an additional type for the purpose of overloading as before. For a statement like $\mathtt{f}_\delta\ \mathtt{obj}_{F(\sigma)}$, we assume that δ is a subterm of $F(\sigma)$. Since F is unique in σ, we often write $\mathtt{f}_F$ instead of $\mathtt{f}_{F(\delta)}$. For virtual functions, we define

$$\mathtt{f\ self}_\delta = \mathtt{f}_\delta\ \mathtt{self}_\delta$$

For instance, $\mathtt{push\ self}_{CF(SF)} = \mathtt{push}_{CF(SF)}\ \mathtt{self}_{CF(SF)}$.

The following examples illustrate how the type of self and other parameters can be used for assigning precise types to functions. Consider for instance a function `add_stack`, which adds all elements of another stack to itself. Its type is `add_stack` :: $oid_\delta \to oid_{Stack} \to m()$ since all we need to assume about the second parameter is that it provides for the stack feature. A variation of `add_stack` is `add_stacks`, which adds the elements of self and an object passed as parameter to a new object. Its type is either `add_stacks` :: $oid_\delta \to oid_{\sigma::Stack} \to m(oid_\delta)$, if the resulting object is of the type of self. Alternatively, it can be the type of the other parameter, as specified via `add_stacks'` :: $oid_\delta \to oid_{\sigma::Stack} \to m(oid_\sigma)$.

These typing techniques yield a simple solution to the problem of binary functions. The typing problem for binary functions has led to an extensive discussion, see [BCC+96]. As another typical example of a "binary" function, consider the function `is_equal` (in an appropriate feature). Clearly, this function should only compare objects of the same type. Hence it can only accept an object with the same type (as the full object at run time) as parameter. To account for this, we use the type

$$\mathtt{is\_equal}\ ::\ \ oid_\delta\ \to\ oid_\delta\ \to\ m(Bool)$$

This type assures that we can only compare two objects of the same type δ.

Specification of State Effects

A common problem of object networks is the specification of the effect of some operation on the global store. As in the last chapter, we use restricted versions of equality. In this case, we can restrict both the features and the object to be compared. We index `=` by two parameters, a type and a set of objects (where both can also be single features or objects, respectively.) Formally, a parameterized equality predicate $=_{F,o}$, where $o \in oid$, only compares the state of the feature F for the object o. This extends component-wise to $=_{\{F_1,\ldots,F_n\},O}$, where $O \subset oid$. Similarly, $\mathtt{a} =_\sigma \mathtt{b}$ is the conjunction of all $\mathtt{a} =_F \mathtt{b}$ with $\sigma :: F$. With equality restricted to particular objects it is possible to compare object stores which differ in the number of created objects.

As with equality, we can generalize the functions `get` and `put` by indexing them by a set of object ids. Note that specifications using these functions must also account for newly created objects.

The last definitions can be illustrated by a two dimensional object store in Figure 4.1, spawned by features and object ids. With the above base features, we can

	object 1	object 2	object 3	object 4	...
SF					
CF					
BF					
UF					
⋮					

Figure 4.1: 2-Dimensional Object Store

formalize the possible state effect of some functions. Instead of an assumes condition, which permits to use operations of other features, we can be more precise with the above functions. In this way, we reduce `uses` clauses referring to other features to appropriate clauses referring to `SR` and `SW` and similarly for `SC`. The general assumption is that a feature can access its own state and can create objects of its own type. In case a feature uses another feature F, this is equivalent to assuming the access rights of F. For read access to a feature F, we can assume $\mathtt{SR}_F$. This in turn means that we can only use functions of F which do not write the state.

As an example, consider the display adapter of Section 3.10, which reads information from the constructor `AA` of the `AsciiAdapter` feature. As it does not write onto the state of the latter, we can write:

```
Feature DA implements DisplayAdapter uses SR_AA
  show_in_window x = ...; s <- get_text x; ...
```

Such specifications are also possible for generic specifications which affect a set of features. For instance, the undo feature reads the state of all inner features (wrt the layered composition). It does however not write on the state of these nor create objects. This is expressed precisely via

```
Interface Undo δ
  undo       :: oid_δ  →  m()
  set_mark   :: oid_δ  →  m()

Feature UF δ implements Undo δ uses SR_UF(σ),SW_UF
  set_mark_UF(σ) x = ...
```

Note that we can use such generic access constraints only via the type used for annotating the feature functions.

For a function on a composed object, the possible state effect is determined by all available features. In the presence of virtual functions the full effect of a function in a particular feature depends on the features present at run time. Hence only approximations are possible.

As we deal with a network of linked objects, the access restriction via `uses` allows the specified access to the state of a feature in all (reachable and new) objects. This is clearly too coarse in some cases, but on the level of features we cannot talk precisely about the state effect, since object identifiers can be passed as parameters of functions. The only known objects at the feature level are the self object and possibly referenced objects via instance variables. Hence we extend our formalism in the following to a more precise specification of the objects which may be affected. For an operation `f` we write

$$\texttt{f uses } \mathtt{SR}_{F,Oid}, \mathtt{SW}_{F',Oid'}, \mathtt{SC}_{\delta}$$

where F, F' are features, Oid, Oid' are sets of object identifiers and δ is an object type, if the (maximal) effect of `f` is limited to

- read access to the state of the feature F of the objects in Oid,
- write access to the state of the feature F' of the objects in Oid' and
- write access to newly created objects of type δ.

Since this is in general insufficient, we generalize it to sequences of $\mathtt{SR}_{F,Oid}$, $\mathtt{SW}_{F',Oid'}$, and $\mathtt{SC}_{\delta}$. For convenience, we often use sets of features (also given via an object type) instead of a single feature. Also, singleton object sets are abbreviated by the single object.

We can now translate a clause `uses F` into assumptions on base features. As an example consider

$$(\mathtt{f}_F\ \mathtt{self}_\delta\ \mathtt{id}_{\delta'})\ \texttt{uses}\ \ \mathtt{SR}_{F,self},\ \mathtt{SR}_{\delta',id},\ \mathtt{SW}_{F',id},\ \mathtt{SC}_{\delta}$$

where (by definition) $\delta :: F$ and $\delta' :: F'$. Thus, the operation $\mathtt{f}_F\ \mathtt{self}_\delta\ \mathtt{id}_{\delta'}$ may read from the feature F of (it)self and all features of object `id`, write to feature F' of object `id` and create objects of type δ. Observe that this state access description is confined to one object. For linked objects, we often have to talk about other objects as well, including newly created objects. More examples will be presented later with lists represented via linked objects, where e.g. appending a new element may have a similar specification as the above.

Depending on the (cumulative) access to `SR`, `SW`, we can formalize the effect of a function as a single read access to the readable feature state, followed by a corresponding write access to the accessible features. Note that this is more involved if objects are created, i.e. the `SC` feature is used. In this case, the number of created objects may depend on the read values. Furthermore, write access to the new objects must be granted. For instance, for the above definition of $\mathtt{f}_F :: oid_\delta \to oid_{\delta'} \to a$, we can formalize the effect as follows, assuming for simplicity that exactly one object is created. Again, we use a side-effect free function `f'`$:: \mathcal{T}ype(F) \to \mathcal{T}ype(\delta') \to (\mathcal{T}ype(F'), \mathcal{T}ype(\delta'), a)$ to describe this as

$$\exists \texttt{f'}.\mathtt{f}_F\ \mathtt{self}_\delta\ \mathtt{id}\ =\ \ \texttt{x <- } \mathtt{get}_{F,self};\ \texttt{y <- } \mathtt{get}_{\delta',id};$$

```
(iv,nv,res) <- result f'(x,y);
put_{F',id} iv; n <- new_δ; put_δ n nv; result res
```

In case an operation creates several objects, the above has to be generalized appropriately. This is more involved, since the number of created objects depends in general on the read values. A fully precise formalization of the effect is not pursued here, since we aim for abstract reasoning in terms of the store access features.[1]

Apart from specification and documentation purposes, bounding the state effect of an operation is important for formal reasoning. We often have to reorder operations when proving properties of operations (see also Sec. 3.8). This abstract reasoning technique aims at reordering two operations. In particular, we can swap the operations, i.e.

$$\mathtt{f}_F;\ \mathtt{g}_G = \mathtt{g}_G;\ \mathtt{f}_F$$

if the following conditions hold:

- $\mathtt{f}_F$ does not write on any state which is read or written by $\mathtt{g}_G$. Formally, if `F uses` $\mathtt{SW}_{H,id}$, then `G uses` $\mathtt{SW}_{H,id}$ or `G uses` $\mathtt{SR}_{H,id}$ must not hold.
- $\mathtt{f}_F$ does not read any state which is written by $\mathtt{g}_G$. Formally, if `F uses` $\mathtt{SR}_{H,id}$, then `G uses` $\mathtt{SW}_{H,id}$ must not hold.

We will discuss examples of reorderings in Section 4.1.3.

4.1.1 Specification of Invariants

So far, we have specified features by equalities on program constructs. When working with linked object structures, it is frequently needed to specify an invariant over object references. We will present an example with linked lists below; for further examples, we refer to [PH97, MPH97]. We will see that we must assume invariants for some types. The need for this in an object-oriented setting was already recognized [MPH97]. In case of feature combinations, the invariant depends on the used features. We discuss in this section techniques for specifying invariants.

In predicate logic specifications, it is convenient to use quantifiers. For instance, to express that some object is reachable from some other one, an existential quantifier over object identifiers immediately solves the problem. Such high level specifications with existential quantifiers have the drawback that they have no computational content. For instance, if invariants are specified with quantifiers, they cannot be executed for run time checks. Also, they can in general not be used to simplify program fragments directly, as with equations on programs.

[1] Specifying the effect on newly created objects appears to be simpler with the techniques developed in the next section, where we talk about the set of alive objects.

In the following, we first introduce the formal model for predicative specifications and then discuss some typical applications. We discuss how such specifications integrate with our program-level specifications. Considering only the pre- and postconditions, the following treatment resembles existing approaches like [PH97]. The main difference is that we still use the specification techniques developed earlier, complemented by pre- and postconditions for invariants.

We will use such non-executable specifications for invariants only. Then the integration with the previous specification techniques is straightforward. It yields a framework for structured proofs, where the reasoning about invariants is performed separately from the reasoning about the result or effect of a computation. As an example for this methodology we show linked lists in the following section.

Usually, pre- and postconditions express properties of the current, global state, but do not modify the state. We can characterize pre- and postconditions as predicate logic formulas which use state-reading monadic functions. This entails that the order of the operation is irrelevant and the operations can be used freely. For convenience, we often use programming level expressions of type $m\ a$ in such conditions simply as type a, e.g. x instead of `result x`. We use the same syntax with implications `=>` as before, but write non-executable predicates in italics.

Furthermore, our setting allows one to quantify over objects, types and individual features. For instance, we can define a predicate $refers\_to$ on a pair of object identifiers (x, y) to express that `y` is reachable by a direct link from `x` via

$$refers\_to(x, y) \Leftrightarrow \exists F \in \mathcal{F}.\mathtt{get}_F\ x = y$$

Observe that we need to quantify over features. To be more precise, projections as in $\exists i, F \in \mathcal{F}.\pi_i(\mathtt{get}_F\ x) = y$ have to be considered as well if features use tuples for the state.

With this predicate, we can specify that some operation preserves this property:

$refers\_to(x,\ y)$ `=> op x y =>` $refers\_to(x,\ y)$

The meaning of this formula can be explained by making the global object store explicit. In addition, we parameterize predicates by a variable for the global object store *os* and assume that `get` behaves on *os* as expected. Then the above translates to an extended predicate with $refers\_to'$ instead of $refers\_to$ and makes the implicit store explicit.

```
refers_to'(x, y, store) =>
       (val, store') <- run (op x y) store => refers_to'(x, y, store')
```

Note that we store the result of `run` in local variables by using monad notation. As this translation is straightforward and similar to techniques in [PH97], we do not go into details of translating all operations into equivalent ones with explicit store.

It is instructive to see how we can link the programming level and quantified predicates. For instance, consider the following easy consequence:

$\texttt{put}_F$ `x y ==>` $refers\_to$`(x,y)`

A similar example is the following:

$y \neq z \wedge \neg refers\_to$`(x,z) => put`$_F$ `x y =>` $\neg refers\_to$`(x,y)`

In many examples of linked structures it is useful to talk about the objects which are reachable from some object.

The following non-executable predicate $Reach(x, y)$ determines if y is reachable from x.

$$\begin{aligned} Reach(x,y) &= \exists F :: \mathcal{F}.y = get_F\ x \vee \\ &\quad \exists z :: Oid.Reach(x,z) \wedge Reach(z,y) \end{aligned}$$

We can also define a version $Reach_F(x, y)$ of $Reach$ which only considers references via feature F:

$$\begin{aligned} Reach_F(x,y) &= (y = get_F\ x) \vee \\ &\quad \exists z.Reach_F(x,z) \wedge Reach_F(z,y) \end{aligned}$$

A common application of $Reach$ is alias avoidance. For instance, to require that two different objects are unrelated by referencing, we can use a precondition as follows:

$\neg Reach(x,y)$ *and* $\neg Reach(y,x)$ `=> s(x,y) = t(x,y)`

A typical example is to bound the number of objects affected by an operation on some object. In case of linked objects, it is difficult to bound the effect of some operation. This problem is also known as the frame problem, as examined for pre- and post-condition style specifications of object-oriented programs in [BMR95].

It is not sufficient to consider the objects reachable from the object, neither before nor after the operation. We have to quantify over the objects reachable before the operation plus all new objects. It is not sufficient to consider the objects which are reachable afterwards, since this may be smaller than the corresponding set before the operation. For this purpose, it is convenient to talk about all active or alive objects. Since this set varies, we sometimes have to be more explicit about alive object identifiers in predicates. For instance, if operations create new objects of a particular type, we want to identify the new objects which have not been present before the execution of the operation.

To specify the set of alive objects, we assume a function

$$alive :: oid_\sigma \to Bool$$

A concrete implementation of *alive* is straightforward in a concrete implementation of an object store, as for instance in Chapter 5. A specification of the properties of *alive* can be found in [PH97].

We can specify some of the interesting properties as follows:

$$\texttt{x <- new}_\sigma \texttt{ => } \mathit{alive}\ \texttt{x}$$
$$\mathit{alive}\ \texttt{x => op => } \mathit{alive}\ \texttt{x}$$

This states that new objects are alive and that the set of active objects does not change under any other operation. Hence, an object can only be created via `new` and cannot be deleted.

The predicate `alive` is used for invariant specifications only. It is not needed otherwise, as we do not allow undefined objects in the language. For convenience, we write $x \in \mathit{Alive}$. As an example for *alive*, assume we want to compare the objects which were present at the start of a computation. This is possible via the following scheme:

$$\forall\ x\ ::\ \mathit{Oid}.\ \mathit{alive}\ \texttt{x => s =}_x\ \texttt{t}$$

We can restrict equality to new objects as follows:

$$\forall\ x\ ::\ \mathit{Oid}.\ \neg\ \mathit{alive}\ \texttt{x => s =}_x\ \texttt{t}$$

Further examples and detailed verification examples can be found in [PH97].

4.1.2 Example: Linked Lists

In the following, we discuss an example with a common pointer structure, linked lists. We show that singly linked and double linked lists can be created in modular fashion by individual features (and via binary functions). For modularity, we split the list nodes into three parts. First, there is the contents of the list elements, for which we use the class `Cell` with a parameter indicating the type of the cell content.

$$\texttt{Interface Cell } \alpha$$
$$\texttt{put\_cell} :: \mathit{oid}_\delta \rightarrow \alpha \rightarrow m\ ()$$
$$\texttt{get\_cell} :: \mathit{oid}_\delta \rightarrow m(\alpha)$$

$$\texttt{Feature Cl } \alpha \texttt{ implements Cell } \alpha$$
$$\ldots$$

The second feature is used to created singly linked lists. This feature introduces the known functions on lists, such as insert (at the front) and append. It also employs a function `abs_list` which returns the full list in a functional data type. More precisely, it returns a list of the object ids of the elements. This function can be seen as an abstraction function in the sense of [PH97], which are used to specify object networks.

We first specify the forward link:

$$\texttt{Interface Fd\_link } \delta$$
$$\texttt{empty} :: \mathit{oid}_\delta \rightarrow m()$$
$$\texttt{is\_empty} :: \mathit{oid}_\delta \rightarrow m(\mathit{Bool})$$

```
  set_next  :: oid_δ  →  oid_δ  →  m ()
  next      :: oid_δ  →  m (oid_δ)
  append    :: oid_δ  →  oid_δ  →  m ()
  abs_list  :: oid_δ  →  m ([oid_δ])

data Option a = Some a | None

Feature Fd δ implements Fd_link δ state link : Option oid_δ
  empty self o    = put_Fd None
  is_empty self   = l <- get_Fd; result (l==None)
  set_next self o = put_Fd (Some o)
  next self       = (Some n) <- get_Fd; result n
  append self o   = l <- get_Fd;
                       case l of None:     set_next self o
                                 (Some n): append n o

  wf_Fd => l <- abs_list y; set_next x y =>
                              abs_list x = result (x::l)
  wf_Fd => l <- abs_list y; append y x =>
                              abs_list x = result append_list(x,l)
  wf_Fd => x:l <- abs_list x; y <- next x => abs_list y = result l
  wf_Fd => empty x            =>  abs_list x = result []
  wf_Fd => is_empty x == True =>  abs_list x = result []
  wf_Fd => empty x      => wf_Fd
  wf_Fd => is_empty x   => wf_Fd
  wf_Fd => set_next y x => wf_Fd
  wf_Fd => next x       => wf_Fd
  wf_Fd => append x y   => wf_Fd
  wf_Fd => abs_list x   => wf_Fd
```

Note that the above code declares a data type *Option* and uses functional operations like `append_list`:: $[a] \to [a] \to [a]$. The predicate wf assures that all lists with this feature are well formed. According to our methodology, we specify the effect of the operations separately from the preservation of this invariance. The well-formedness predicate wf_{Fd} can be formalized as follows:

$$wf_{Fd} = \forall x :: oid_{\sigma::Fd}.\neg Reach_{Fd}(x, x)$$

where $Reach_{FDd\_link}$ is specified as above, with the restriction that only links via the feature `Fd` are considered. Note that we cannot specify this invariant on the interface level. Therefore, we cannot write any specification on the interface level, as we need to assume that the invariant holds.

Observe that this specification permits that a list element is reachable from itself via other objects. This occurs in doubly linked lists and also in case it is attached to

a feature with an object reference.

The following feature adds a backward link to linked lists to create what is usually called doubly linked lists.

```
Interface Bk_link δ uses Fd_link δ
  set_prev :: oid_δ  →  oid_δ  →  m ()
  prev     :: oid_δ  →  m(oid_δ)

Feature Bk implements Bk_link state :: (Option oid_δ)
  set_prev self o = put_Bk o
  prev self       = (Some p) <- get_Bk; result p

 wf_Bk => set_pref self x => wf_Bk
 wf_Bk => prev self       => wf_Bk

  -- Interaction handling
  empty_Bk             = lift empty
  is_empty_Bk          = lift is_empty
  set_next_Bk self o   = lift set_next self o; set_prev o self
  next_Bk              = lift next
  append _Bk           = lift append
```

In this example it is instructive to see that, using our concepts for binary functions, only homogeneous lists are possible. Thus lists are either singly or doubly linked, but a mixture is not possible.

The well-formedness condition for linked lists is slightly more involved, since backward references have to be the inverse pointers of the forward references. It can be formalized as follows:

$$wf_{Bk} = wf_{Fd} \ \wedge \forall x :: oid_{\sigma::Bk}.\neg is\_empty\ x \ \texttt{=>}\ prev(next(x)) = x$$

This assumes that *pref* and *next* only read state. We will show later in Section 4.2 that a doubly linked list is a conservative extension of a singly typed one which preserves the specification of singly linked lists. Additional properties of doubly linked lists follow immediately from the invariant:

```
 wf_Bk => n <- next o => prev n = result o
```

There is an alternative version of `Fd_link`, which allows to define `abs_list` in a more interesting way. The idea is to assume the store feature to access the inner state. Hence the function `abs_list` can return a list of values, not just *oids*.

```
Interface Fd_link1 δ α uses Store α
  set_next :: oid_δ  →  oid_δ  →  m()
  next     :: oid_δ  →  m(oid_δ)
```

```
append   :: oid_δ → oid_δ → m()
value    :: m(α)
abs_list :: oid_δ → m([α])
```

The specification and implementation of `Fd_link1` is similar to the above and can use the store feature to access the local state of the inner features. Thus, it is for instance possible to link any objects of the same type which support the store feature.

4.1.3 Formal Reasoning in Object Networks

We show in the following some of the problems of proving properties about object networks. A particular emphasis is put on abstract and reusable proofs. Since we do not reason explicitly about how state is affected, we hope that the techniques are scalable to larger systems.

Our proof methodology is to swap the operations such that the state readers are moved towards the left and hence the other operations can be executed symbolically. In case one operation uses on the output of another one, this dependency disallows swapping the two operations. As an example, consider extending the counter by a method `add`, which adds a value to it. One rule for add may be the following

```
add self x; add self y = add self x+y
```

With this rule, we want to simplify

```
add self x; y <- size self; add self y
```

Clearly, the last operation depends on the second one. Note that we cannot swap these. Thus, we have to swap the first two operations and adapt y in the last operation as follows:

```
y <- size self; add self x; add self (y+x)
```

Note that the last operation not only uses the newly computed value, but also modifies the state used by the state reader function `size`.

In general, reasoning about the effect of a method on the global state can be quite involved if references to objects are used. The idea is to argue that some functions, including their effects on other objects, can only touch certain features. This can be used to reshuffle operations of different features. Observe that the variations of equality are nicely illustrated as horizontal or vertical slices as shown in the last section. Thus, restricting equality to certain features just compares the appropriate lines in the object store, whereas restricting to object identifiers compares only certain columns. This essentially yields an upper bound for the state affected by some methods. Although this is just an approximation, it is sufficient in many cases. In particular, the approximation is not affected if the globally used state is extended. Hence proofs may be reused under extensions.

Let us first summarize the state effect of the linked list functions:

```
next_Fd j       uses RS_Fd j
append_Fd j k   uses RS_Fd, Reach_Fd k , WS_Fd, Reach_Fd k
get_cell_Cl j   uses RS_Cl j
```

With the above, we can for instance prove that

```
i <- next j; append j k ; x <- get_cell j
```

is equal to

```
i <- next j;  x <- get_cell j ; append j k.
```

Note that i≠j follows from the invariant for lists, which obviously has to hold initially. As the object identifiers i and j are not mutable variables, i≠j for the full operation sequence. This simplifies the proof in this case.

In general, reasoning about linked structures with sharing is quite difficult (see also [PH97, Moe82]).

4.2 Refinements and Subtyping in Object Networks

We discuss in the following the adaption of the refinements discussed in Section 3.7 to object networks. Since specification of object networks is considerably more difficult, we cannot expect the same simple results. We will need additional requirements and identify cases which are still tractable by our techniques.

The first observation is that invariants must be preserved when operations or programs are lifted to a new feature combination. For instance, we have seen in the linked list example that doubly linked lists require an extra invariant (plus the one for singly linked lists). For behavioral refinement it is clearly required to establish the added invariants. Even for weak behavioral refinement, it is not sufficient to establish the extra invariants under an abstraction.

We associate invariants with object types and generally require that the invariants are preserved globally for all involved objects of the appropriate type. (This is similar to [PH97], where however semantic subtypes are not considered.) It it is not the case that a method which works correctly on a feature, say linked lists, works correctly on a subtype with extra invariants. For instance, linked lists are often used with different entry points. A common problem is that sharing may cause the violation of invariants in structures, which may go unnoticed with simple behavioral subtyping as discussed in the last chapter.

Consider as an example a lock feature on linked lists, which works as the lock feature presented earlier but locks all objects in the linked list. Hence the invariant is that either all objects are locked or none. (A similar, common example are colored lists with the invariant that all list elements have the same color.) Hence locking a list which is a (shared) sublist of another one breaks this invariance. In this case, we need a global invariant which ensures that for all lists the elements have the same state.

We will present a generic result similar to Theorem 3.9.2 in the next section, which covers the case of linked lists. We will show that doubly linked lists are a conservative extension, since singly linked lists are only extended, but not modified.

Behavioral refinement works as in the case of one object, except that invariants have to be considered. For instance, we want to show that doubly linked lists do fulfill the laws for linked lists. Behavioral refinement holds in this case, since both features use the same object network. For instance, the equalities (on all objects) of the linked list specification hold for this extension as well. Note that we also have to show that the invariants of both features are preserved.

4.2.1 Weak Behavioral Refinement

Weak behavioral refinement is significantly more difficult for object networks. The idea of weak refinement is to abstract from some feature and to show that laws hold under this abstraction. The problem is that the effect of a function call is not limited to one object. In particular, other objects of different type can be involved. Hence it is difficult to abstract from a feature with our earlier techniques and to look only at the effect of particular features. This is particularly difficult in case new objects are created, as we cannot fix the number of modified objects before the method. Thus we largely focus on simpler cases where no objects of different types are used or are ignored.

Assume we want to lift a property for a function `f` for an object of type σ to $F(\sigma)$. The problem is to separate the effect of `f` $\mathtt{x}_\sigma$ and the new feature F in `f` $\mathtt{x}_{F(\sigma)}$. In the case of a single object, this is possible in a schematic way using types in many cases. In this setting, it is useful to distinguish the following cases:

- The object has no object references. Then we proceed as in the case of one object and specify via equality on a single object, e.g. `f x` $=_x$ ....
- Another typical case are homogeneous structures like linked lists. Here, we can abstract from the new feature for all objects of this type. Thus refinement specifications are of the form `f` $\mathtt{x}_{F(\sigma)} =_\sigma \ldots$ This case will be considered below with schematic abstractions.
- In the remaining case, objects of other types are involved. These can also appear as function parameters. Hence in general we need to abstract from the effect of the added feature. This is not generaly possible with schematic abstraction techniques.

We present a result for semantic subtyping for the first two cases in the next section. In our setting, specifications which only talk about one object or objects of one type can be handled generically. The reason is that the used abstractions are constructed via the object types. Thus, other specifications can only be lifted partially with this result.

Since specifications for the last case cannot be treated schematically in our setting, we do not pursue this case further. Note that it is clearly possible to use our specification techniques for lifting individual specifications. For more complex object networks, it seems more promising to work on the level of functional abstractions. Compare for instance the representation of linked lists with the functional representation of lists used for stacks. A formal verification of an algorithm using linked lists can be found in [PH97]. Since we have shown the essential concepts of the formalism of [PH97], it is possible to model these results in our setting.

4.2.2 Semantic Subtyping with Object Networks

In the context of object references we can rephrase the problem of semantic subtyping for parameter passing. Assume we define a function which uses an object of some type. In a concrete application, an object of a subtype of the expected type is used instead, e.g. when passed as parameter. The question is under which conditions specifications or proofs about the function still hold for the more specific parameter.

Consider a typical example for weak refinement. Assume the following function definition.

$\texttt{f(x}_{Stack}\texttt{) = push x 1; pop x}$

For the parameter `x`, all we can assume are the stack laws. (Recall that $\texttt{x}_{Stack}$ stands for some σ with $\texttt{x}_{\sigma::Stack}$.) In case of an invocation of `f` with an object `y` of type $UF(SF)$, we can show the following:

$\texttt{f(y}_{UF(SF)}\texttt{)} =_{SF} \texttt{unit}$

This equation does not hold for $UF(SF)$. Hence we have to abstract from the undo state recording. Therefore, $UF(SF)$ is only a weak refinement of SF.

For reasoning about procedures, a typical problem is aliasing, which is even possible for variables of different type. We need to use case distinctions for reasoning in the presence of aliasing. Consider the following definition:

$\texttt{f'(x}_{Stack}\texttt{, y}_{Stack,Undo}\texttt{) = push x 1; lock y; push x 2; unlock y; pop x}$

In case of an invocation with distinct parameters x and y we can show the equation

$\texttt{x}_{SF} \neq \texttt{y}_{UF(SF)} \texttt{ => f'(x, y) = push x 1}$

In case of aliasing, we can prove:

$\texttt{x == y}_{UF(SF)} \texttt{ => f'(x, y)} =_{SF,x} \texttt{unit}$

This only holds since we can infer that `x` has also type $UF(SF)$, and hence the refinement rules for this type also hold for for `x`. Thus, to reason about object references with aliasing, we need to consider appropriate case distinctions.

Aliasing is a special case of sharing. To ensure consistency for shared reference structures, we need invariants. For instance, we have seen in the linked list example that doubly linked lists require an extra invariant, compared to linked lists. As shown above, this extra invariant is not automatically preserved by an operation of a supertype. For instance, to use a procedure which uses a linked list as parameter with a doubly linked list, we must require that the procedure maintains the invariant of doubly linked lists. In case of sharing of data structures, we therefore have to guarantee the subtype invariant for all objects.

Assuming that invariants are preserved, we can extend the definition of conservative redefinitions to the case of object nets. To simplify the presentation, we only consider functions with one argument. Since objects may store references, this is not a general limitation.

Definition 4.2.1 A method redefinition of a function $\mathtt{f}_\sigma$ for $\mathtt{f}_{F(\sigma)}$ is called *conservative*, if

$$\mathtt{f}\ \mathtt{x}_{F(\sigma)} =_\sigma \mathtt{f}_\sigma\ \mathtt{x}_{F(\sigma)}$$

and furthermore $\mathtt{f}\ \mathtt{x}_{F(\sigma)}$ preserves the invariance properties of the feature in $F(\sigma)$.

Note that the equality in the above definition must hold for all objects which have these features.

Theorem 4.2.2 *Assume an operation sequence* op *and a function* f *such that*

$$\mathtt{f}\ \mathtt{x}_\sigma =_\sigma \mathtt{op}$$

holds. If op *and* f *only use operations which do not affect the features in $F(\sigma)$ (including invariants) or functions of σ which are conservatively redefined from σ to $F(\sigma)$, then*

$$\mathtt{f}\ \mathtt{x}_{F(\sigma)} =_\sigma \mathtt{op}$$

Proof The proof proceeds as the one for Theorem 3.9.2. The difference is that we use equality on the features in σ for all objects. Hence the functions used by $\mathtt{f}\ \mathtt{x}_{F(\sigma)}$ either do not affect these or are reducible to the ones in $\mathtt{f}\ \mathtt{x}_\sigma$. In addition we have to establish the invariants here, which follow from the assumption that it is guaranteed by each individual operation. □

This result goes beyond the corresponding results for semantic subtyping in [DL97, LW93], as we cover object references. (More precisely, abstraction functions on single object, as used in [DL97, LW93], are not sufficient for this purpose.) Note that problems with aliasing were already recognized in [DL97, LW93], but not wrt linked structures.

Note that our definition above only considers invariants for one feature wrt all objects. In the general case, all invariants of all features have to be considered, as shown for object-oriented systems in a recent report [MPH97].

It is easy to verify that this theorem applies to doubly linked lists, which can hence be used as a subtype of linked lists. Another example are lists with the undo or lock features. As with Theorem 3.9.2, it is easy to extend the result to the combination of several features. This theorem also entails weak behavioral refinement for a specification on a single object with no references as a special case.

It would be desirable to extend this result for non-homogeneous structures, in order to account for the effect of other objects of different types. Our generic abstractions are in general not sufficient for this case, since the newly added behavior must be "separated" from the old one. As we cannot use schematic abstractions, this has to be done on an individual basis using restricted equalities. Furthermore, the combination of such individual refinements, which is essential for composing several features, does not work schematically as well.

4.3 Exceptions

We show in the following that another common ingredient of programming languages can be modeled easily in our setting. Exceptions are a technique to treat program errors in a convenient and also systematic way. For instance, the `pop` operation on an empty stack is unspecified so far; it could either do nothing or raise an exception, since no sensible return value exists. In many languages, for instance in Java [GJS96] or SML [MTH90], exceptions are provided. Exceptions are raised either by the run-time system or by applications. For simplicity, we limit our attention to the latter. In case an exception occurs, all computation is stopped and control is passed to the next error handler in the call stack. For a detailed treatment for exceptions, we refer to [Fla89].

It superficially appears that exceptions are similar to the lock feature, which disables other features. However, the lock feature only disallows write access to the state of an object. It cannot disable the return values of computations, as it is still on the level of the programming language. In contrast, the usual model for exceptions introduces error cases which abort any computation. This is usually modeled via monads as follows. A computation of type

$$m(a)$$

is enhanced by error cases by instantiating m to

$$m(a) = m'(Err\ a),$$

where Err is a datatype constructor which adds an error element to any type a via the following definition of a sum type where $|$ separates the type options.

$$Err\ a = Error \mid Data\ a$$

As we use pattern matching, the two constructors `Error` and `Data` suffice and selectors are not needed. For simplicity, we do not model different error or exception cases. In

Section 5.4 it is shown that adding an extra parameter for more detailed exception information is straightfoward.

The "partially" instantiated monad m' can be further instantiated as done earlier for state transformers. With $m'(a) = ostore \to (ostore, a)$ we obtain

$$m(a) = ostore \to (ostore, Err\ a)$$

With this monad construction, we can add error values to any specification if needed. (We will however do the instantiation in the reversed order.) Recall that adding exception handling later is one motivation for keeping the monad m abstract.

Assuming the above monad construction, we can define the following additional base features used for exceptions. We define the functions for raising and handling exceptions as follows:

```
Interface Error where
 raise_err  ::  m ()

Feature ER implements Error

Interface ErrorHandler where
 read_err :: m a  →  m Bool

Feature EH implements ErrorHandler
```

As for the store features, we split the exception treatment into two features to give a more fine grained control via features.

The function `raise_err` raises an exception. Detecting an exception can however not been done at the programming language level with monadic functions. The reason is that by definition all operations are fully disabled. Therefore, we need a function `read_err` for this purpose, which maps a computation over `a` to $m\ Bool$, where the boolean value indicates if a exception has occurred. This simple version discards the value of type a if no exceptions has occurred. It is a simple extension to pass the computed value, if it exits.

The function `read_err` maps computations to computations of different type; the expression `read_err op` returns a new computation which indicates if an error has occurred. Note that the global state is passed on to the resulting computation, which remains unchanged after an error has occurred. For instance, assuming that `pop` raises an error for an empty stack, we can guard such a command via `read_err` as follows:

```
  err <- read_err(x <- pop id; y <- pop id); if err then ....
```

By construction, the actual types of values `x` and `y` are lifted over error cases. Hence they can only be used inside the computation passed to `read_err`, which corresponds to the scope of the local definition.

This construction is similar to other programming languages, where usually an extra syntactic construct is provided for raising and handling exceptions. For instance, in Java [GJS96] one uses the keywords `try` and `catch` to write the above as

```
try { x = id.pop(); y = id.pop(); }
   catch(Error) {  ...  // error handling code }
```

Note that a Java statement calling a method of an object `id` with no parameter is denoted as `id.pop();`.

As with state access, we model exceptions as a predefined feature. Similar to the global state and object references, exceptions do not affect just a single object. Hence we add the underlying monad construction for exceptions similar to the state access features.

To accommodate exceptions, we have to redefine the basic monad operations. These vary for different monads and have so far been tailored for state transforming monads. In particular, the sequencing operator `_ ; _` $:: m\ a \to (a \to m\ b) \to m\ b$ has to check for errors. For a sequence `f ; g` the second operation is bypassed if an error occurs. If not, the (internal) resulting value of type $Err\ a$ is of the form `Data x`, and `x` is passed on as parameter to `g`, which is of type $a \to m\ b$. Furthermore, the `result` function has to embed the value into the new data type. We show these redefinitions in detail in Section 5.8.2 and give a more abstract specification of exceptions here. Before this, we have to define equality on operations which may use exceptions. The canonical answer is to compare the resulting values of type $Err\ a$ instead of just a. Hence an equation

```
op = op'
```

compares the resulting state, the error condition, and in case of no error, the resulting value. For the restriction to the state of some feature F of the form

$$\texttt{op} =_F \texttt{op'}$$

the equality holds if the state of the feature and the computed value are equal, unless an exception occurs.

Similar to the restriction of equality to state, we define that

$$\texttt{op} =_{ER} \texttt{op'}$$

holds exactly if `op` and `op'` either both raise an error or produce the same result. Thus the implicit state change is ignored.

In many cases, the opposite of the above is interesting: Define

$$\texttt{op} =_{\neg ER} \texttt{op'}$$

as `op` and `op'` produce the same result if none raises an error. The subscript $_{\neg ER}$ can be seen as a shorthand for omitting ER in the list of the compared features.

As computations are modeled as data objects, we can define this equality simply by comparing the resulting pair of a computation of type $(Store, Err\ a)$. Hence

```
op =¬ER op'
```

holds iff `run op = run op' = (ostore, res)` or `run op = (ostore, Error)` or `run op' = (ostore, Error)`.

In comparision to generic specifications like the one for the lock feature, we cannot determine the behavior solely via types. Hence our specification is more operational as we must assure that `raise_err` is not invoked in the operation sequence `op`, which is done via a conditional rule below.

```
raise_err; f          =  raise_err
read_err (raise_err)  =  result True
read_err (result x)   =  result False
read_err (f; op)      =  f; read_err (op) <= f ≠ER raise_err
```

The important point of this specification is the `f` $\neq_{ER}$ `raise_err` condition. The equality expresses that f does not raise an exception, since it is unequal to the function which raises an error. Note that there is only one function of this type which injects an error. To understand the scope of the third rule, recall that `unit = result ()` holds.

An easy consequence of the above is the following:

```
read_err (f; raise_err) = f; result True if  f ≠ raise_err
```

As an example, we use the exception mechanism to redefine the stack operations. An important difference to the base storage features is that we want to add exceptions as an extra feature. In this way, we can use the exception mechanisms only if needed. In case an application can exclude stack underflow (or cannot react appropriately), then the exception mechanism should not be used. Our technique is to introduce a new feature, here `ErrStack`, which uses `ER`, and to lift other features to the new feature `ER`. In particular, the `pop` operation has to be redefined:

```
Interface Error where
   stack_err  ::  m a

Feature ES implements Error uses ER
   stack_err = raise_err

popER  self = ifseq (is_empty self) then stack_err else lift (pop self)
pushER self = lift (push self)
...
```

Note that the lift operation also has to be redefined for error monads, as shown in the following chapter. Clearly, the usual laws for lifters must hold.

It is instructive to evaluate the following code with the above rules. We fist unfold the definitions of `pop` and `if` and then push `is_empty` out of the `read_err` scope.

```
  empty id_ER; er <- read_err( x <- pop id_ER)
= empty id_ER; er <- read_err( x <- if_seq (is_empty id_ER)
                                        then stack_err
                                        else lift (pop self))
= empty id_ER; b <- is_empty id_ER; er <- read_err(x <- if b
                                        then stack_err
                                        else lift (pop self))
= empty id_ER; er <- read_err( x <- if True then stack_err else lift pop)
= empty id_ER; er <- read_err( stack_err )
= empty id_ER; result True
```

In case some object uses exceptions, we assume that its object type is of the following form:

$$F_1(...F_n(EH(ER(SR(SW(SC)))))\ldots)$$

As before, we often omit the base features $EH(ER(SR(SW(SC))))$ and indicate the usage via `uses` statements.

4.3.1 Exceptions and Refinements

We discuss in this section the relation between refinements and exceptions. Our particular execution model, in which we specify features, treats exceptions as data objects. In this way, we can abstract over exceptions similar to other abstractions. Hence we can accomodate the abstraction over the occurrence of exceptions as a refinement. This is similar to the usual notion of partial correctness, which states that a particular property holds if the program terminates. We develop this idea below and show that it provides for a gentle approach to behavioral subtyping. With this refinement it is possible to handle many practical cases in a simple way.

We first discuss the typical example of the stack with underflow handling, followed by stacks with a (modified) bound feature. In the former, only the `pop` operation may raise an exception for an empty stack. Hence we can show behavioral refinement, as

$$\texttt{push}_{ER(\sigma)}\ \texttt{self x; pop}_{ER(\sigma)}\ \texttt{self} = \texttt{unit}_{ER(\sigma)}$$

holds, if it holds for σ. Thus adding exceptions poses no problem for behavioral refinement, if only extra behavior is added. Recall that the stack underflow was underspecified so far.

To accommodate exceptions for the bounded stack, we modify the interaction specification as shown below. Alternatively, we can lift the bound feature to ER and redefine `check_bound` to raise an error.

```
Feature BF implements Bound uses ER state b :: Int
  check_bound x = ....

 --- Stack Interaction
  push self a = if_seq (check_bound a) then raise_error else (push self a)
```

We can now show conditional refinement for the following equation:

```
push self_BF(σ) x; pop self_BF(σ) = unit
      <= check_bound x == True
```

Interestingly, the following holds without condition:

```
push self_BF(σ) x; top self_BF(σ) = push self_BF(σ) x; result x
```

The point is that in case `push` triggers an exception, both sides yield the same state and an error condition. This shows that exceptions are an interesting alternative to conditional refinements. We can however go a step further and develop a notion of weak behavioral refinement for exceptions such that the above equation holds without condition. The idea of this extension is to use restricted equalities to specify exception behavior. For instance, for the bounded stack, this yields the following refinement:

```
push self_BF(σ) x; pop self_BF(σ) =_¬ER unit_BF(σ)
```

We view this as more abstract than conditional refinement, since the condition is unspecified. Furthermore, we can reason that upon a successful computation, the desired equations do hold. With conditional refinement, we cannot conclude anything about a successful computation. Thus exceptions provide for a useful and operational check of conditional refinement. In some cases, we have to be a bit more careful. If an operation locally checks for exceptions, we must explicitly reason about exceptions. In our setting, we can assure that this does not happen by assuming that the code does not use the error handler feature `EH`.

This refinement notion is similar to partial correctness, which states that a property of a program holds if it terminates. With exceptions, the property holds if the program terminates without raising an exception.

Note that our interpretation of exceptions is easily possible with monads, since exceptions are modeled via a data type. Hence we need little additional specification techniques and can in particular abstract over exceptions similar to other data values.

On the other hand, exceptions delay the burden of showing that a program terminates successfully, if this is required. At some point, we usually have to show the condition. However, in most cases the formalization via exceptions is more convenient. Consider again the small stack, which can only hold a fixed, small number of elements. Then, when passing such a stack object to a procedure, we must assume that the procedure does not insert too many elements. This is difficult to formalize and our partial result is appropriate if the details of the procedure are not under consideration.

Note that in our approach, we distinguish exceptions from other reasons for partiality, such as non-termination. This last point is an advantage over other techniques which use partial functions to model conditional refinement [Stø96]. Furthermore, our model of exceptions can be used in a flexible add-on fashion and is compatible with current programming languages.

Finally, note that there are cases where methods of inner features are disabled, but raising an exception is inappropriate in this case. Recall the lock feature, which disables operations if the object is locked. In this case, an exception should not be raised if a disabled operations is invoked. Hence we need to use conditional refinement and need to assure the invariant that the object is not locked. From this we can conclude a subtype relation.

4.4 Comparison to Object-Oriented Type Systems

In the following, we discuss how our model for typing features relates to approaches for typing object-oriented languages. We claim that our model, which is inspired by the type systems of functional languages [Jon95, NP95], is a simple but expressive approach. We contribute in the next chapter an embedding of our typing techniques into the language Gofer [Jon91], for which formal typing rules and decidable type inference exist. For an object model, a formal description of the state access via object identifiers requires dependent types [Bar91], as the type of the object depends on the object identifier. For a fixed number of features, it is however possible to model this in the Gofer type system with relations between types (see Section 5.8).

There exists an abundance of type systems for object-oriented languages, for instance see [AC96, PT94, Cas97, BCC+96]. Although we do not aim for a detailed comparison to other object-oriented type systems, which are in most cases quite complex, we show the main differences here.

Most of these approaches model classes with (mutable or non-mutable) variables and methods, as well as subtyping between classes. Furthermore, explicit data structures are mostly used to represent objects. With our abstract model of state, we can avoid explicit data structures for objects, e.g. records.

Apart from this, the main differences concern the technical treatment of the inherent typing problems of object-oriented language in conjunction with other known extensions. We summarize some of them below:

1. Formal notions of classes and objects. In our case, we have no notion of a class, only concrete objects are created. Features are similar to classes, but objects are created from a set of features and not from a single class.

2. The subtyping relation. This relation is usually generated by the subclass relation, although there are exceptions in so-called object-based languages (see [AC95] for a comprehensive treatment). Here, we use the subset relation on features, as discussed in more detail below.

3. Co- or contravariant changes of types of methods under subclassing wrt the subtype relation. As an example, assume a method `f` of a class `A` which takes a parameter of type B. When redefining `f` for a subclass `A'` of `A`, the concept of covariance allows to specialize the type of the parameter to subclass/type `B'` of

`B`. This entails that $\texttt{f}_{A'}$ `b` with `b` of type `B` is not permitted, `b` must be of type `B'`. The purpose of covariance is to specialize the type of a method in a subclass as done here via self types. This however conflicts with type soundness of the type system in many cases [Cas97]. More precisely, an unrestricted usage of covariance may result in (noticed or unnoticed) dynamic type conflicts. Semantically sound, but rarely useful, is contravariance, which permits that `B'` is a supertype of `B`. There exist several approaches which circumvent these problems with covariance [BPF97, BSG95], which we follow here by using so-called self types. A similar type system for object-oriented languages is the one developed in [OW97] for the language Pizza as used in Chapter 6.

4. Parametric classes and subtype polymorphism are often included and intermingled. As we use instantiation of type variables for (restricted) subtype polymorphism (see below), there is no interaction with parametric polymorphism in the form of type variables.

The second item of the above is instructive for the comparison of subclasses and features. In object-oriented languages, one creates a fixed hierarchy of classes which induces a transitive subtype relation. For instance, assume `ColoredPointClass` is a subclass of `PointClass`. Since class names are used as types in most languages and since subclasses generate subtypes, an object of type `ColoredPointClass` can be used wherever an object of type `PointClass` can be used.

In our setting, we model this via two features, `Color` and `Point`. An object with both feature then supports a set of interfaces, here `{Color, Point}`. Subtyping is simply modeled as set inclusion. Each feature interface corresponds to the notion of a protocol in [AC95].

The next interesting issue is how we model subtype polymorphism. In usual object oriented systems, we can write a function

$$\texttt{draw} :: \mathit{PointClass} \to ....$$

This function takes an object of type $\mathit{PointClass}$ as parameter. Due to subtyping, we can also pass objects of type $\mathit{ColoredPointClass}$ to `draw`. In our setting, we use polymorphism to model this. The function `draw` has the following polymorphic type in our setting:

$$\texttt{draw} :: \mathit{oid}_{\sigma::\mathit{Point}} \to ....$$

Hence we use (apart from an explicit type of object identifiers) a polymorphic type with the type variable σ. This variable can be instantiated by any type which implements, among others, the feature interface Point. This well known technique is also used in [OW97]. In our approach it is also possible to specify a particular feature combination (without admitting subtypes) by giving a concrete object type. This is sometimes useful, but is not possible in most object-oriented languages which generally admit subtypes.

Below the interface level, we use a layer model to compose features via feature constructors. For instance, if F_i is the constructor for feature i, we can compose three features via $F_1(F_2(F_3))$. These types fulfill two purposes simultaneously: they determine the type of an object (i.e. the features it provides) and also model the state of an object directly. This close link allows for generic access to the state of an object which is convenient for specifications. Furthermore, types are used for abstraction functions which are needed for refinement results. Using these, we easily establish results comparable to [DL97, LW93]. We argue that this simple form of abstractions is both practical and sufficient for our setting.

Chapter 5

Monadic Feature-Oriented Programming

In this chapter, we model feature-oriented programming concepts in a functional language. The constructions we use here are fully compatible with the last chapters and give concrete models for the monads and monad operations used earlier. We show in the following our feature composition architecture, which originates from results for monad compositions. While in this setting the origins of these techniques are visible, the imperative implementation language presented in the next chapter avoids these complications, as implicit state, exceptions and inheritance are available in the language. In contrast to the often more readable imperative version, we no not need any language extension here.

As we embed the feature-oriented (and object-oriented) construction into the type system of Gofer [Jon95] (which is somewhat extended over the one of Haskell [PHA+96]), we give a concrete typing model for object-oriented systems. This is indirectly used in the following chapter, as the language Pizza [OW97], which we use there, has a type system which is similar in several respects.

An advantage of this embedding is the integration in a functional language. Hence no language extension is needed and both programming styles can be used interchangeably. Furthermore, as functional programming allows one to write highly abstract programs, this embedding also provides for a prototyping language for executable specifications. The disadvantage is that the type system has some rough edges and furthermore the current implementation is slow, since the imperative part is largely interpreted. Regarding the former problem, a simple and safe modification of the type system would suffice, since it is in principle sufficient.

In this chapter, the main technical contributions towards feature-oriented programming are as follows.

- Using concepts for monad composition, we introduce a novel model for programming features in a modular and composable way which generalizes inheritance or subclassing.

- We show that some functionality (an undo function) which depends on several features can be implemented abstractly for any feature combination using type computations via type classes.

- We generalize some programming techniques used in [LHJ95] to generic classes of stateful and error monads.

In contrast to the last two chapters on specification, we also include exceptions for the single object case. Since this requires more elaborate typing and monad concepts, this was not shown earlier for simplicity. The exception model for the single object here is different from the global store with exceptions, since an exception feature can be added like any other feature.

We demonstrate our concepts by two examples, the stack example and a small example using telecommunication features, where feature interactions have attracted great attention [Zav93, CO95]. More examples can be found in the next chapter.

For implementing our concepts with monads we generalize techniques which were developed in [LHJ95]. In our model, classes correspond to monads, which can be viewed as particular abstract data types. The interesting point is that (some classes of) monads compose nicely and that we can build monad transformers, which transform an abstract data type to another. This is used to add features to objects. For instance, the mainly used monad transformers add (local) state (and extra functionality), from which we draw the comparison to inheritance. We show that implicit state via monads is essential for our abstract programming techniques. Similarly, overloading via type classes is important, as the type of polymorphic functions in feature implementations can only be determined after an object is composed from a set of features.

To compare this work with earlier results on monads, note that Moggi [Mog91] aimed at lifting monads just by their types. This was extended to liftings for particular types of monads in [LHJ95], using their specific properties. Our technique is to name concrete instances of monad classes (e.g. state monads) and to program liftings depending on the names, but using generic liftings for the class of monads. As the names are identified with features, this clearly goes along the ideas of inheritance. Furthermore, we mostly use just state monads, which compose easily.

We use the type system of the language Gofer in the following. This has the advantage that we can build on top of an existing, widely used language. Although Gofer type classes fully suffice for the single object case, there are some insufficiencies in the case of object networks. As mentioned in Section 5.8, some features cannot be made polymorphic, as the Gofer type class system requires all type variables in the parameters of a class to appear in the type declarations of member functions.

This chapter is structured as follows. After a brief introduction to the Gofer type system and monads in Section 5.2, we show the concepts of stateful features in Section 5.3 and of error features in Section 5.4. The problems of multi-feature interaction for generic features are discussed in Section 5.5, followed by examples for stack features in Section 5.6. Another example for feature interactions in telecommunications

is presented in Section 5.7. In Section 5.8, we present a functional model of the object model of the last chapter.

5.1 Programming Monadic Features

To give a first idea of how to program features with monads, we show (some of) the code for the stack and the counter features. Our concepts are provided by executable Gofer functions [Jon91] and type constructions. We use the constructor classes of Gofer [Jon95], which extend Haskell's type classes [NP95] and have been partly adopted in Haskell 1.3 [PHA+96].

We use monadic state transformers for modeling implicit state as in imperative languages, which is essential for the desired flexibility and modularity. Composing features is done by the type system of Gofer with type constructions and type classes. A type class declares certain functions for its member types. Since types can be in several classes, we can use Gofer type classes for modeling features. Observe that type classes do not correspond to classes in object-oriented programming, but determine if a type has some feature. Thus a type can be in several type classes, which resemble the idea of interfaces, as e.g. in Java [GJS96].

Note that there are some syntactic and typing differences to the treatment in the last chapter. Therefore, we do not use the same names for the example interfaces and feature constructors names.

- Operation sequences are denoted via `do{op; ... ;op}`.
- Instead of the keywords interface and feature etc, we use the notation for Gofer type classes. A feature interface is declared as a type class, feature implementations as instance declarations for type classes.
- A type is in a type class (e.g. `StackMonad` or `CountMonad`) if the corresponding functions are provided in an instance declaration, as shown below.
- We use the type constructors $StackT, CountT$ to add features to a type. For instance, if m is the type of an object (a monad), then $StackT\ s\ m$ is a new type which also supports the stack feature with a local state of type s. (In the earlier chapters, the names SF and CF were used for this purpose. The difference is that here the constructors also have a parameter indicating the state used by the feature.)
- As we use overloading provided by Gofer, we do not annotate functions with type subscripts. For object networks in Section 5.8, an extra function parameter is used to facilitate overloading.

In the following code, the first type declaration for *StackT* declares that *StackT* is a state transformer, adding implicit state to the object of type m.[1] The second statement declares that *StackT* [*Int*] m is in the class *StackMonad* of stacks of integers.[2] The type declaration and the instance correspond to feature constructors in the earlier chapters. Furthermore, we have to give implementations for the functions which the feature provides, here push and pop. Recall that we write types, type constructors and type declarations in italics. As in the last chapters, we use monadic types. First, we have to declare the interface of the stack feature:

```
class  Monad m => StackMonad m  where
  push     :: Int  →  m ()
  pop      :: m Int
  is_empty :: m Bool
```

The following is a slightly simplified version of the constructor declaration.

```
  -- add implicit state of type s to m (simplified here)
type StackT s m  =  StateTrans s m

instance StackMonad (StackT [Int] m)  where
   push a   = do{ s <- get; put (a:s) }
   pop      = do{ s <- get; put (tail s); result (head s)}
   is_empty = do{ s <- get; result (s==[]) }
```

In the above implementation, the do-notation for sequential computations in monads is used. Each statement in the do construct may compute a value and assign it to a local variable, e.g. `s <- get` assigns the result of `get` to `s`. In such a monad computation the added, implicit state can be modified via the functions `put` and `get`. Note that these access functions always refer to the implicit state of the "current" feature.

Next we show the counter feature, whose functions are also implemented via state transformers.

```
class Monad m => CountMonad m  where
  size     ::  m Int
  inc      ::  m ()
  dec      ::  m ()

type CountT Int m  =  StateTrans Int m

instance CountMonad (CountT Int m) where
```

[1] State transformers will be explained in detail later. Also, the following type declaration is shortened. The full code and the class declarations are shown later.

[2] Polymorphic stacks are possible via a binary class `StackMonad`, using the extra argument for the type of the stack. However, this leads to ambiguous types later.

```
size  = get
inc   = do{ i <- get; put (i+1) }
dec   = do{ i <- get; put (i-1) }
```

It remains to lift the functionality of stack to the context of a counter. In contrast to the last sections, we need an explicit instance declarations for this. The following instance declaration states that $(CountT\ Int\ m)$ has the stack feature, under the preconditions (stated before the `=>`) that m has the stack feature, i.e. `StackMonad` m, and that $(CountT\ Int\ m)$ is a `CountMonad`.

```
instance (StackMonad m, CountMonad (CountT Int m)) =>
               StackMonad (CountT Int m) where
  push a   =  do{ inc; lift (push a)}
  pop      =  do{ dec; lift pop}
```

The code for push first increments the counter and then via `lift (push a)` the push function of the inner object ("superclass") of type m. Roughly speaking, `lift` corresponds to the function super as e.g. in Java and is, like `get` and `put`, defined later. Alternatively, if there is no interaction, one would just write

```
pop = lift pop
```

which could also be made a default (as implicit in object-oriented programming). With the above code, an object of type

$$CountT\ Int\ (StackT\ [Int]\ m)$$

provides both features and behaves as expected. In general, liftings should preserve the functionality of the lifted features, i.e. an individual feature always behaves identically (if no others are used in between). For the standard lifting, this can be shown similar to [LHJ95].

The implementation of the undo feature is more involved and is presented in Section 5.5. The idea of the simple undo implementation is to save the local state of the object each time a function of the other features is applied (e.g. push, pop). The undo feature raises several new issues:

- The lifting of functions of the other used features is schematic: Always save the state first and then call the function to be lifted. In contrast to object-oriented programming, this can be done once and for all by a particular function

  ```
  lift_undo f =  do{ local_s <- lift gets ;
                                put (Some local_s) ;
                                (lift f) }
  ```

  which lifts any function `f` to the undo feature. Note that `lift gets` refers to the state of the inner object.

- Undo depends essentially on all "inner" features, since it has to know the internal state of the composed object. Since we work in a typed environment, the type of the state to be saved has to be known. This multi-feature interaction is solved by an extra feature, which allows one to read and write the local state.

5.2 Monads, Type Classes and Features

In the following, we explain the technical background needed for implementing the feature model in a functional language. Some of the material is repeated from Chapter 3, but with an emphasis on typing and other language issues.

The concept of monads has been introduced to programming for modeling state in functional languages [JW93] and for writing code which is easy to modify [Wad93]. Both aspects will be essential in our context. The techniques presented here are based on investigations on features in programming languages [LHJ95].

5.2.1 Type Classes

A type class in Haskell is essentially a set of types (which all happen to provide a certain set of functions). Each class declaration introduces a new class and a set of new function names, which are overloaded for each member of a class. For instance

```
class Eq a where
   eq ::  a  →  a  →  Bool
```

introduces the class `Eq` of all those types a which provide a function `eq` $:: a \to a \to Bool$. A class declaration is like a module interface: it separates declarations from implementations. Instance declarations determine the members of classes and give concrete implementations for the member functions, e.g.

```
instance Eq Int where
  eq = eq_int
```

In general we can instantiate classes not just by base types but also by type terms. For example, we may wish to express that a type $[a]$ admits equality provided a does. This is achieved by the following instance declaration, where the Haskell notation `=>` adds a list of type assumptions (here `Eq` a) to the new instance `Eq` $[a]$.

```
instance Eq a => Eq [a] where
  eq [] [] = True
  eq (a:as) (b:bs) = and [eq a b, eq as bs]
```

Note that the last two `eq` expressions refer to two different instances of `Eq`, one for a and one for $[a]$.

5.2.2 Constructor Type Classes

The extension to constructor classes of Gofer [Jon95, PHA+96] allows n-ary type classes. Furthermore, these arguments may not just be types, but can be type constructors. Constructor classes are often used when standard type classes are too coarse to describe the types of the member functions. The standard example is the binary container class, whose instances typically are lists and trees:

```
class Container c a where
  member :: a → (c a) → Bool
```

Here we can express that the type $c\ a$ depends on a. If $c\ a$ is replaced by a type s, in a `class Container' s`, then the type of `member` $:: a \to s \to Bool$ would be too general: we cannot write a sensible function which for any type a checks membership in a type s. Typical instance declarations are:

```
instance Container List a where
  member e []  = False
  member e a:s = or [eq e a,  member e s]

data Tree a  = Leaf a | Node (Tree a ) (Tree a )

instance Container Tree a where
  member e (Leaf a)   = eq e a
  member e (Node a b) = or [member e a, member e b]
```

5.2.3 Monads

Programming with monads provides a compromise between imperative languages, where statements affect an implicit, global state, and stateless functional languages, where all information flow is — sometimes tediously — explicit. Monads also separate building computation (e.g. composing state transformers) and running a computation.

A monad is a type constructor m with some operations and laws. If a is a type, then $m\ a$ is the type of a larger object which "wraps" a, often a function type (e.g. a state transformer) as shown later. In monadic style, a function from a to b is assigned the type $a \to m\ b$. There are standard functions to work with monads, defined in the type class for monads, which builds upon the functor class:

```
class Functor m where
 map :: (a → b) → (m a → m b)

class Functor m => Monad m where
 result :: a → m a
 bind   :: m a → (a → m b) → m b
```

Function `result` inserts a value into the "empty" monad and `bind` applies a monadic function to a value of type `m a`. Below we use monad transformers to construct new monads. A simple example for a state monad can be found in Section 5.8 for the constructur `St`.

Note that we use the do-notation for `bind`, defined as

`do { x <- m ; t}` $=^{def}$ `m bind` λ`x.t`

This notation extends canonically to sequences of bind applications. For the monad laws we refer to Section 3.2. Note further that `unit = result ()` is not used here.

5.2.4 Features: Monads with Operations

Features are defined as monads with additional operations. These can be viewed as predicates over types which characterize the features. For instance, for the basic stack and counter features we define:

```
class  Monad m => StackMonad m  where
  push     :: Int  →  m ()
  pop      :: m Int
  is_empty :: m Bool
```

This declares the two classes used in the introduction, `StackMonad` and `CountMonad`, with their corresponding functions. It assumes that `m` is a monad. (Note that () is the empty type.)

5.3 A Class of Stateful Monads

We show in the following the underlying machinery for features which add state to some object. The basis of state monads is a type

`type` $StateTrans\ s\ m\ a\ =\ s \rightarrow\ m(s,\ a)$

which extends any monad m to a type of a state transformer for a state of type s. This transformer can be applied repeatedly, since $StateTrans\ s\ m$ is again a monad, as shown below. For the following general model, we generalize over this type and just assume the functions `closeS` and `openS`. These access the internal structure of state monads and are only used internally.

The ternary class `StateMonadT` $c\ s\ m$, where s is the type of the added state, m a monad and c an appropriate type constructor, declares that $(c\ s\ m)$ is a stateful monad with the following functions (for some of which definitions are included):

```
class Monad m => StateMonadT c s m  where
 closeS :: (s  →  m(s,  a))  →  c  s  m  a
 openS  :: c  s  m  a  →  s  →  m(s,  a)

 get    :: c  s  m  s
 get    = closeS(λs.result(s,s))

 put    :: s  →  c  s  m  ()
 put a  = closeS(λs.result(a,()))

 lift   ::  m  a  →  c  s  m  a
 lift m = closeS(λs. do{x <- m; result(s ,x)})
```

For the functions `get`, `put` and `lift`, also definitions are provided in the class declarations. The functions `closeS` and `openS` are used to show that any state monad is a monad:

```
instance  StateMonadT c  s  m => Functor (c  s  m)  where
   map f xs = closeS (λs. (openS xs) s bind λ(s',x). result(s', f x))

instance  StateMonadT c  s  m => Monad (c  s  m)  where
    result x = closeS(λs.result(s,x))
    m bind k = closeS(λs0. (openS m) s0 bind λ(s1, a). openS (k a) s1)
```

This generic class generalizes the various stateful monads in [LHJ95], where the above definition of monads is repeated for stateful monads.

5.3.1 Defining a Stateful Feature

With the above concepts, we can show in detail the definition of basic stack features. Only the following data type declaration is needed, as well as declaring it to be a stateful monad.

```
data StackT  s  m  a = STM(StateTrans  s  m  a)

instance StateMonadT StackT  s  m where
   closeS  x       = STM x
   openS (STM x)   = x
```

Note that we use an extra constructor `STM` to define *StackT* via a data type definition. This wrapper is needed for type checking; otherwise *StackT* cannot be distinguished from other state transformers with the same type.

Similar declarations are needed for the counter feature. The instance declarations for *StackT* and *CountT* can be found in Section 5.1.

5.4 A Class of Error Monads

As for stateful monads, we can similarly define a generic monad which adds extra values to the computation. These are used to model exceptions or errors. For instance, with the above definition of stacks, stack underflow results in a program error. Using error monads, we can cope nicely with such cases. In applications it is then possible to use stacks with or without error handling as needed.

We introduce error features similar to stateful features. They can hence be added like any other stateful feature. Since this requires more complicated type constructions than used in Chapter 3, this was only shown for a global object store in Chapter 4. The advantage here is that exceptions can be supplied more selectively to only particular features in a feature combination.

Whereas stateful monads build upon a particular function type ($StateTrans$), we use a sum type here:

$$\texttt{data}\ Err\ e\ a \quad = \quad Data\ a \mid Error\ e$$

$$\texttt{type}\ ErrT\ e\ m\ a \ = \ m(Err\ e\ a)$$

Thus $ErrT$ adds error elements of type e to a monad m. Compared to 4.3, we now add a parameter for different error cases. Note that this composes with state monads. For instance, we obtain the type

$$(ErrT\ e\ (StackT\ s\ Id))\ a \ = \ STM(\ s\ \rightarrow\ Id(s,\ Err\ e\ a))$$

The class of error monads supports open and close functions as for state monads, plus generic functions to inject and check errors (`put_err, read_err`), and the canonic lifting function `lifterr`.[3]

```
class Monad m => ErrMonadT c s m where
 openE    :: c s m a  →  m(Err s a)
 closeE   :: m(Err s a)  →  c s m a

 put_err  :: s  →  c s m a
 read_err :: c s m a  →  c s m Bool
 lifterr  :: m a  →   c s m a
 lifterr c = closeE (map Data c)

 put_err  s = closeE(result(Error s))
 read_err m = closeE(map isError (openE m)) where
         isError (Error s) =  Data True
         isError (Data x)  =  Data False
```

[3]Due to the type system, the function cannot be overloaded to work under the same name as in stateful monads. Adding an extra class for monad transformer is no solution, as typing does not permit to declare instances for both classes of monads.

Showing that `ErrMonadT` $c\ s\ m$ is a monad is more complicated. It can for instance be shown if we assume that m is a StateMonad. For this we use the concepts of [JD93], which can be generalized to classes of monad transformers.

For instance, an error handler for stack underflow is written by lifting stack over Err, using Int for error values. Since we only use the base functions of `ErrMonadT`, we do not need to introduce an extra class and a type constructor for this. (An example with an explicit class is shown in Section 5.7.2.)

```
instance (StackMonad m, ErrMonadT ErrT Er m)=>
          StackMonad (ErrT Er m)  where
   pop      = do{ b <- is_empty ;
                  if b then (put_err 0)
                  else (lifterr pop)}
   push a   = lifterr (push a)
   is_empty = lifterr is_empty
```

Lifting other, independent features is canonical:

```
instance (CountMonad m, ErrMonadT ErrT s m) =>
          CountMonad (ErrT s m)  where
   size = lifterr size
   inc  = lifterr inc
   dec  = lifterr dec
```

This lifting can even be generalized to any state monad, if `CountMonad` is independent of all other stateful features.

In the current model for features, we have just provided generic monad composition for a set of stateful features with one error feature. Although it is possible to use several error features, it is easier to use one error monad transformer and to build other features on top of it. For instance, we only use the integer 0 as error message here and leave others open for other error cases. (In case several features use the same error message, we can treat this as an interaction.)

5.5 The Undo Feature: Multi-Feature Interaction

We continue the stack example by introducing the undo feature, which has interesting interactions with several other features. The problem is that the undo feature must access the local states of all (stateful) features the object already has. Since we work in a typed setting, we also need the type of all local states. Hence undo depends on several features. Recall that we have shown a highly generic specification for undo in Section 3.6. On the programming level, this corresponds to multi-feature interaction, as the generic state access is not directly available. However, the solution is similar to the specification in Section 3.6 and it is easy to see that the implementation fulfills the specification of Section 3.6.

As we work with standardized monads, it is possible to add an auxiliary feature,[4] which determines the state of an object and provides access to it. Thus undo can be added to any feature combination.

The additional class `SMonad` for stateful monads is declared via

```
class Monad m => SMonad s m where
  gets  :: m s
  puts  :: s → m s
```

This binary class declares that monad m has state s and provides access functions. Instances can be defined schematically for both classes of monads, e.g.:

```
instance (SMonad s0 m, StateMonadT c s m) =>
          SMonad (s,s0) (c s m) where
 gets       = do{s  <- lift gets; s' <- get; result (s',s) }
 puts (a,b) = do{s <- lift (puts b);put a }
```

This expresses that $c\ s\ m$ has state $(s, s0)$, if m has state $s0$. Now we can define the undo feature via `SMonad` as follows. Since there may be no saved state for undo, we use the data type *Option* for the copy of the local state in the following code:

```
data Option a  =  Some a | None

data UndoT s m a  =  UTM(StateTrans s m a)

instance StateMonadT UndoT s m where
  closeS  x     = UTM x
  openS (UTM x) = x

class Monad m => UndoMonad m where
  undo   ::  m ()

instance SMonad s m => UndoMonad (UndoT (Option s) m) where
   undo    =  do{ u <- get ; case u of
                                None    -> result ()
                                Some u1 -> lift (puts u1)}
```

The other interesting point about undo is the lifting of functions of other features. The advantage is that lifting proceeds via the following generic scheme, which first extracts the local state of the object, updates the saved state and then calls the lifted function:

```
 liftundo f = do{local_s <- lift gets ;
                 put (Some local_s) ;
                 (lift f) }
```

[4]Not shown in Figure 2.3.

Lifting for the basic stack features proceeds canonically:

```
instance (SMonad s0 m, StackMonad m ) =>
          StackMonad (UndoT (Option s0) m) where
  push a = liftundo (push a)
  pop    = liftundo pop
```

There is an interesting interaction when the counter is used. For lifting `size`, which does not affect the state, we can either overwrite the saved state or leave it unchanged (as shown in the comment in the code below). In the former case, `undo` after `size` will have no effect. With our model of feature interaction, we just have to use the appropriate lifting function for interaction resolution.

```
instance (SMonad s m, CountMonad m ) =>
          CountMonad (UndoT (Option s) m) where
    size          =  liftundo size
 -- alternative:  =  lift size
    inc  = liftundo inc
    dec  = liftundo dec
```

Currently, just one lifting between two features is possible due to the type system.

Whereas the store feature is be used here to model the generic state access functions on the programming level, it is also possible to implement the functions individually for each feature. This yields a customized version of state access. For instance, this can be used to specify and to implement a more flexible, but still generic undo feature.

5.6 Using the Stack Features

A simple example for an object (monad) with two features is the following, which uses the identity monad Id with no features as base monad. By the following type declarations features are selected.[5] Running the above state transformers requires extra machinery for injecting an initial state and for extracting the computed value.

```
 type St = Int  -- Type for StackT
 type Ct = Int  -- Type for CountT

  -- stack with counter
test1 :: (CountT Ct (StackT [St] Id)) St
test1 =   do{
  push 1 ;
  push 2 ;
```

[5]Gofer can infer the types without these declarations, but the inferred type is too general, as Gofer allows several (base) implementations for a type class.

```
  size }     -- computes 2

  -- stack with undo
test2 :: (UndoT (Option ([St], ())) (StackT [St] Id)) [St]
test2 = do{
  push 1 ;
  push 2 ;
  push 3 ;
  undo ;
  p2 <- pop ;
  undo ;
  p1 <- pop ;
  result [p1,p2]} -- computes [2, 2]

-- stack with counter + undo
test3 :: (UndoT (Option (Ct, ([St], ())))(CountT Ct (StackT [St] Id))) [St]
test3 = do{
  push 1 ;
  push 2 ;
  push 3 ;
  undo ;
  p2 <- pop ;
  s <- size ;
  p1 <- pop ;
  result [p1,p2,s]}
    -- computes [1, 2, 1]

--  counter with undo
test4 :: (UndoT (Option (Ct, ())) (CountT Ct Id)) St
test4 = do{
  inc;
  inc;
  undo;
  size }              -- computes 1
```

5.7 Feature Interaction in Telecommunications

In the area of telecommunications, feature interaction problems have led to a new research branch [Zav93, CO95] focusing on such interaction problems which hinder the rapid creation of new services. The problem in feature interaction stems from the abundance of features telephones (will) have. For instance, consider the following conflict occurring in telephone connections: B forwards calls to his phone to C. C

screens calls from A (ICS, incoming call screening). Should a call from A to B be connected to C? In this example, there is a clear interaction between forwarding (FD) and ICS, which can be resolved in several ways. For many other examples we refer to [CGN+94].

We demonstrate our techniques, including an example for virtual functions, with the following set of features for this domain of connecting calls:

- ICS (incoming call screening)
- Forwarding of calls
- Error handling for busy phones (also used for disallowed calls)

The first two of these features add local state, i.e. the origin of the call, which is not needed for the other features.

In this application, there are similar feature interactions as in the last section. The interactions mostly stem from extending the environment or from resource conflicts. The first can be handled by liftings, the second by the order on features.

Our full implementation contains another feature, called OCS (outgoing call screening), which is similar to ICS. Already with four features and several resolutions to the interactions, there are many different feature combinations.

5.7.1 Forwarding

The goal in the following is to provide functionality for connecting calls.

```
  -- type for phone numbers
type Dn = Int

class PMonad m => FWDMonad m where
  forward :: Dn → m Dn
```

Forwarding only uses two (constant) lookup functions `fd_check` and `fd` with forwarding information and adds no local state. For simplicity, we use a state transformer which adds no state.

```
data FwdT s m a = FTM (StateTrans () m a)

instance StateMonadT FwdT () m where
   closeS  x     = FTM x
   openS (FTM x) = x

instance FWDMonad (FwdT () m) where
  forward nr = if (fd_check nr) then result (fd nr)
                                else result nr
```

5.7.2 The Busy Monad

The Busy monad provides a function for raising a busy signal and is based on the error monad.

```
class Monad m => PMonad m where
  raise_busy :: m a

type PhoneT = ErrT ()

instance ErrMonadT ErrT () m => PMonad (PhoneT m) where
  raise_busy = put_err ()
```

5.7.3 Incoming Call Screening

For ICS we use a state monad with the origin of the call as local state:

```
data IcsT m a = ITM (StateTrans Dn m a)

instance StateMonadT IcsT Dn m where
  closeS  x     = ITM x
  openS (ITM x) = x

class IcsMonad m where
  check_ics  :: Dn → m Dn
```

The corresponding implementation uses a function `check_ics1`, which is not shown here. It simply checks disallowed callers.

```
  instance IcsMonad (IcsT Dn m) where
  check_ics dest   =  do{ orig <- get;
                          if (check_ics1 orig dest)
                             then result dest
                             else raise_busy }
```

This code raises an exception in case the call is disallowed, which in turn is the signal for busy.

5.7.4 Resolving the ICS/Forward-Interaction

To resolve the interaction between forwarding and ICS, we lift the forward function to ICS. If we choose the standard lifting by

```
instance (FWDMonad m, StateMonadT IcsT a m) =>
          FWDMonad (IcsT a m)  where
 forward  a = lift (forward a)
```

then the local state added by ICS is not affected by forwarding. Hence, the ICS check uses the origin of the call. If the intermediate hop is to be used, we would write

```
forward a = do{put a; lift (forward a)}
```

instead. Note that `get` and `put` refer to the ICS feature here. Again, lifting allows a modular resolution of the interaction between two features.

5.8 A Functional Object Model

In this section, we present a concrete functional model of the concepts presented in Chapter 4. In contrast to the case of one object with mutable state, we model here a supply of mutable objects.

The presentation is in the lines of Section 4.1. We show concrete implementations of the monad constructions used in Section 4.1. Furthermore, it is interesting to compare this version to the previous one with one monadic object. Whereas in Section 5.3 the used state is individually determined via the feature constructors, we must assume a single global object store. Similarly, exceptions are modeled globally. In this sense, the monad constructions are in fact simpler that the ones presented in Section 5.3. However, the encoding of objects and types is more involved. For the understanding of features and for reasoning about features, the simple case in Sections 5.3 and 5.4 is more instructive.

In general, the model presented in Chapter 4 requires dependent types. Using the techniques of explicit type markings and type relations, we are able to implement an object model for a fixed set of features. Observe that the abstract mathematical description uses dependent types: the object store maps an object identifier and an object type to an object of this type. For a fixed number of features, we can model the global object store as a mapping from an object identifier and a feature identifier to the state of the feature for this object.

In this way, we show type inference by an embedding into the Gofer type system. However, the Gofer type system is at some points unnecessarily restrictive for our applications. (In case of a type class with several parameters, every function defined in this class must use all parameters in its type. This is not needed if the actual overloading can still be resolved.) As a consequence, the embedding is possible, but quite delicate at some points.

5.8.1 Features on a Global Object Store

We show in the following the main constructions needed for features on an object store. The global store is a state monad whose implicit state is a mapping from an object index and a feature to the value of the feature for this object. Note that the store class is just an auxiliary construction; only the function `new` is used in feature definitions. State access for features and objects is shown below.

```
type Index  =  Int  -- Object number
type FIndex =  Int  -- Feature number

class Monad m => Store m  where
  rd  :: Index  →  FIndex  →  m Value
  set :: Index  →  FIndex  →  Value  →  m ()
  new :: o  →  m (o, Index)
```

The following simple implementation uses a function for the object store and a variable indicating the number of used objects. As explained in Section 5.3.1, we need a wrapper S for the following type St.

```
type Ostates =  (Index, FIndex  →  FValues)

data St a =  S( Ostates  →  (Ostates, a))
unS (S a) =  a

instance Store St where
  rd  o i   = S(λ(max, atribs). ((max, atribs), atribs o i))
  set o i a = S(λ(max, atribs).
                 ((max, λo1 i1. if (o==o1)  and (i==i1)
                                    then a else atribs o1 i1), () ))

  new obj = S( λ(max, a). ((max+1, a), (obj, max+1)) )

instance Functor St where
  map f (S a) = S (λs. let (s1, sa) = a s in (s1, f sa) )

instance Monad St where
  result a   = S(λp. (p,a))
  m `bind` f = S(λp. let (p1,a) = unS m p
                     in unS (f a) p1)
```

Next we show the encoding of objects. An object consists of an index in the global store plus some type information. Similar to the single-object case, we represent the object type via nested feature constructors. These are modeled as data type constructors. An object is modeled as a pair of a term with a data type constructor and an index in the global store.

```
class Object o where
  idx :: o  →  Index

instance Object  (o, Int) where
  idx (o,n) = n
```

```
-------------- Feature constructors
data SF a = Sf a
data CF a = Cf a
data FD a = Fd a  -- homogeneous lists
```

For instance, concrete objects are `(Sf(), 2)`, `(Cf(Sf()), 5)` and `(Lf(Sf()), 5)`, assuming `SF`, `CF` and `FD` are declared as feature. This is explained next.

The central Gofer class in our encoding is the class `Feature` m a_type f o below which declares a type constructor `f` as a feature with the appropriate state access functionality. Together with the function `new` of the store feature, this provides for the functionality of the constructors $SR(SW(SC))$ of Section 4.1. As we do not aim for specifications here, we do not split the object access functions into three features (like in Section 4.1) for brevity.

```
class (Store m, Object o, Conv a_type) => Feature m a_type f o where
  get  :: (f ax,o) → m a_type
  put  :: (f ax,o) → a_type → m ()
  get (ax,o)   = do{xs <- rd (idx o) (fnum ax); result (proj xs)}
  put (ax,o) x = set (idx o) (fnum ax) (inj x)

call (o,n) f   = f (o,(o,n))  -- virtual call of f on (o,n)
self (a,(o,n)) = (o,n)        -- remove first parameter
                              -- (used for overloading)
```

The parameters in `Feature` m a_type f o and the assumed type classes have to be explained in detail. The first one is the global monad store `m`. The type a_type is the type of the state variable associated with the feature. The class `Conv` a_type provides the functions `proj` and `inj` to store the value into the global store. The last parameter is the object type of the full object (self type). Furthermore, all functions of features have to be indexed by the feature (as done via subscripts in Chapter 3). For this purpose, we use a pair `( f ax, o)`, where `o` is the type object (again a pair) and `f ax` is subterm of the type of `o`. For instance, we will implement $\texttt{push}_{SF}$ $\texttt{oid}_{CF(SF)}$ in the syntax of Chapter 3 as

```
push (Sf,(Cf(Sf()), Id)) a = ...
```

Note that `fnum` determines the index of a feature.

Since we use an extra parameter to implement overloading, calling virtual methods requires an auxiliary function `call` to initialize this extra parameter, as shown above. Similarly, the function `self` strips this parameter off.

As an example of the above class, the feature constructor for the stack feature is declared as follows, where `[Int]` is the type of the used state.

`instance Feature` m $[Int]$ SF oo

Now we are ready to define the actual features. For instance, the counter is defined as follows:

```
instance Feature m Int CF oo

class Store m => Counter m ax o  where
  reset :: (ax,o) → m ()
  inc   :: (ax,o) → m ()
  dec   :: (ax,o) → m ()
  val   :: (ax,o) →  m  Int

instance Feature m Int CF o=> Counter m(CF id) y o where
  reset o =  put o 0
  inc o   =  do{ i <- get o; put o (i+1)}
  dec o   =  do{ i <- get o; put o (i-1)}
  val o   =  get o
```

The definition of the stack feature shows the definition of a virtual function `push2`:

```
  class Store m => Stack m ax o where
  empty :: (ax,o) → m ()
  push  :: (ax,o) → Int → m ()
  pop   :: (ax,o) → m Int
  push2 :: (ax,o) → Int → m ()

instance (Feature m [Int] SF (o, n), Stack m o (o, n)) =>
         Stack m (SF ax)  (o, n) where
  ...
  push2 o a = do{call (self o) push a ; call (self o) push a}
```

Note that we need an auxiliary function `call` for virtual function calls.

The following example of linked lists shows the usage of binary functions, as in Section 4.1.2. We only sketch the following list feature here, as more implementation details can be found in Section 4.1.2. The state variable of this feature is of type `Option o`, where `o` is of the full object (self type), in order to model a possibly empty link (`None`) to the next object of the linked list. This type is then used in the functions `next` and `get_next`.

```
data Option a = Some a | None

instance Feature m (Option o) Fd o

class Store m =>  Fd_link m ax o  where
  empty      :: (ax,o) → m()
```

```
  is_empty   :: (ax, o)  →  m(Bool)
  next       :: (ax, o)  →  m(o)
  set_next   :: (ax, o)  →  o  →  m ()
  ...

instance Feature m (Option o) Fd o =>  Fd_link m (Fd ax) o where
  ....
```

A sample program using a linked list of objects with the counter feature is the following:

```
o <- new (Cf (Fd ()) ;
o1 <- new (Cf (Fd ());
o2 <- new (Fd ());
set_next o o1 ;
```

Note that replacing `o1` by `o2` in the above result in a type error.

5.8.2 Exceptions on a Global Object Store

To add exceptions to a global object store, we use the techniques of Section 4.3. The obtained monad is the result of combining the above monad with the error monad as used in Section 5.4. More precisely, the constructor combination $EH(ER(SR(SW(SC))))$ is implemented by the monad StE (), for which the corresponding feature functions are defined below. Since we have to provide for exceptions on a global and not per object basis, there is only one global error monad. Hence we do not provide the error handling as a pluggable feature of one object. Instead, the features with exceptions rely on an appropriate monad.

We proceed similar to the last section, but use a different base monad as follows. As before, we use a wrapper Se.

```
data Err a  = Data a | Error

data StE a     = Se( Ostates  →  (Ostates,   Err   a ))
unSe (Se a)    = a  -- Auxiliary functions for wrapper Se
isErr (Data a) = False
isErr Error = True

instance Store StE where
  rd  o i    = Se(λ(max, atribs). ((max, atribs), Data (atribs o i) ))
  set o i a  = Se(λ(max, atribs).
                  ((max, λo1 i1. if and [o==o1,i==i1]
                                 then a else atribs o1 i1), Data () ))
  new obj    = Se(λ(max, a). ((max+1, a), Data (obj, max+1)) )
```

```
instance Functor Err where
  map f Error    = Error
  map f (Data x) = Data (f x)

instance Functor StE where
  map f (Se a)   = Se(λs. let (s1, sa) = a s in (s1, map f sa) )

instance Monad StE where
  result a   = Se(λp. (p,Data a))
  m `bind` f = Se(λp. let (p1,a) = unSe m p in doErr f a p1) where
      doErr f (Data a) pl = unSe (f a) pl
      doErr f Error pl = (pl, Error)
```

The functions for raising and catching errors are declared in a class `ErrMonad`:

```
class Monad m => ErrMonad m where
  raise_err :: m a
  read_err  :: m a → m (Err a)

instance ErrMonad StEwhere
  raise_err  = Se(λp -> (p,Error))
  read_err m = Se(λp -> let (p1,a) = unSe m p in (p1, Data a))
```

In this setting, it is now possible to define a pluggable `StackErr` feature:

```
class (ErrMonad m, Store m) => StackErr m ax o where
  stack_err  :: (ax, o) → m a
  stack_err o = raise_err

instance Feature m () SE o => StackErr m (SE ax) o where
  stack_err o = raise_err
```

The following type expressions for lifting stack over `SE` to `Stack` $m\ o\ (o,n)$ require explanation. The object type (o,n) must be stated in this form, as o appears explicit in $(SE\ o)$. (One cannot introduce different type variables for $(SE\ o)$ and (o,n), since the instance declaration would be too general.)

```
instance (Lifter SE, ErrMonad m, StackErr m (SE o) (o, n),
          Stack m o (o, n) )
        => Stack m (SE o) (o,n) where
  pop o    =   do{ b <- is_empty o; if b then stack_err o
                                    else lift pop o}
  ...
```

An example usage is the following code, in which stack operations are protected in an exception handler.

```
o <- new (Cf sfb) ;
empty o;
e <- read_err( do{pop o; top o} );
result  (if (isErr e) then "Error detected" else "Ok")
```

In this way, the exception raised by `pop` is caught.

Chapter 6

Imperative Feature-Oriented Programming

In the following, we present feature-oriented programming in an imperative setting. In particular, we show how an existing language, namely Java [GJS96], can be extended to support feature-oriented programming. More advanced concepts are translated to the language Pizza [OW97], whose compiler in turn translates to Java.

Since some of the earlier concepts are simplified and restricted for this integration, this version of feature-oriented programming is easier to use. This gives an ideal implementation platform for feature-oriented development. Furthermore, it shows that Java is easily extended to feature-oriented programming.

Apart from the integration aspect, the other novel point here is that we give a detailed comparison to common object-oriented programming techniques. The translation of our extension delineates the relation to inheritance and aggregation. This also gives a detailed comparison between both, which reveals a few subtle differences. The semantics of our extensions can be defined via direct translation to plain Java, as shown below, or via the functional model presented earlier.

The integration in existing languages like Java and Pizza requires a few modifications and simplification of the earlier developed concepts.

- All functions are virtual and are furthermore inherited/lifted by default. (In Gofer, a function which is not lifted is not available for this subtype/feature combination.)
- The type expressions have to be compatible with Java and Pizza.
- The types permitted in feature assumptions in lifters are limited. For a simplified version of the general concepts, extra constructs are added.

We present the basic notion of features in Java; for several extensions, in the lines of Chapters 3 and 4, we use Pizza. Note that there are strong relations between the type and class concepts in the languages Gofer, Java and Pizza, as explained in [OW97].

We show that parameterized features (similar to templates) work nicely with interactions and liftings. Compared to the specifications discussed earlier, we have to be more explicit about type declarations. As we will see, there can also occur type dependencies between two features, which can be clearly specified in our setting. Another extension permits features which add exceptions and may raise exceptions for other features via lifters. Hence checking exceptional cases is viewed as resolving an interaction. As liftings for some features are generic, we model this case via higher-order functions.

The main technical contributions and results in this chapter are as follows:

- Translations of a feature-based language extension of Java into Java, one via inheritance and one via aggregation and delegation.
- A programming language version of parameterized features and type dependencies between features, followed by a translation into Pizza [OW97], an extension of Java.
- The translations lead to a detailed comparison of aggregation and inheritance. This unveils two cases where aggregation is more powerful than inheritance due to typing problems.

In the following section, we discuss the first three features of the stack example. We define the feature-oriented extension of Java via translations in Section 6.2, followed by an extension to parameterized features in Section 6.3. This section also discusses the remaining two features, undo and bound. An extension to generic liftings via higher-order functions is shown in Section 6.4. Section 6.5 discusses features which introduce exceptions and raise exception in lifters.

Many examples from several areas are shown in Section 6.6, starting with variations of design patterns in Section 6.6.1. Section 6.6.2 models common software description techniques using automata by features and analyzes interactions. An interesting application area where feature interactions have been extensively examined are multimedia and telecommunication systems, as discussed in Section 6.6.3. This application area actually triggered this research.

6.1 Examples of Feature-Oriented Programming

In this section, we introduce imperative feature-oriented programming using the example modeling variations of stacks. (The undo and bound features are shown later in Section 6.3.) For this purpose, we present an extension of Java in the following.

We first define interfaces for features. Although not strictly needed for our ideas, they are useful if there are several implementations for one interface. Furthermore, they ease translation into Java, as a class can implement several interfaces in Java.

```
interface Stack {
  void empty();
  void push(char a);
  void push2(char a);
  void pop();
  char top();
}
interface Counter {
  void reset();
  void inc();
  void dec();
  int size();
}
interface Lock {
  void lock();
  void unlock();
}
```

The code below provides base implementations of the individual features. Note that we only treat stacks over characters; parametric stacks will be considered later. The notation `feature SF` defines a new feature named `SF`, which implements stacks. Similar to class names in Java, `SF` is used as a new constructor for a feature. Using the other two feature implementations, `CF` and `LF`,

```
  new LF (CF (SF))
```

creates an object with all three features. For interaction handling, it is important that features are composed in a particular order, e.g. the above first adds `CF` to `SF` and then adds `LF`.

```
feature SF implements Stack {
   String s = new String();
                        // Use Java Strings
   void empty() {s = ""; }
   void push(char a)
         {s = String.valueOf(a).concat(s); };
   void pop() {s = s.substring(1); } ;
   char top() { return (s.charAt(0) ); } ;
   void push2(char a)
         {this.push(a) ; this.push(a); };
}
feature CF implements Counter {
   int i = 0;
   void reset() {i = 0; };
```

```
   void inc() {i = i+1; };
   void dec() {i = i-1; };
   int size() {return i; };
}
feature LF implements Lock {
   boolean l = true;
   void lock() {l = false;};
   void unlock() {l = true;};
   boolean is_unlocked() {return l;};
}
```

In addition to the base implementations, we need to provide lifters, which replace method overriding in subclasses. Such lifters are separate entities and always handle two features at a time. In the following code, features (via interfaces) are lifted over concrete feature implementations. For instance, the code below `feature CF lifts Stack` adapts the functions of `Stack` to the context of `CF`, i.e. the counter has to be updated accordingly. When composing features, this lifter is used if `CF` is added to an object (type) with a feature with interface `Stack`, and not just directly to a stack implementation. This is important for flexible composition, as shown below.

```
feature CF lifts Stack {
  void empty() {this.reset(); super.empty();};
  void push(char a)
        {this.inc(); super.push(a) ;};
  void pop() { this.dec(); super.pop() ;};
}
feature LF lifts Stack {
  void empty() {if (this.is_unlocked())
                      {super.empty();}};
  void push(char a) {if (this.is_unlocked())
                      {super.push(a);}};
  void pop() { if (this.is_unlocked())
                      {super.pop();}};
   int  size() { return super.size(); };
   char top() { return super.top(); } ;
}
feature LF lifts Counter {
  void reset() {if (this.is_unlocked())
                      {super.reset();}};
  void inc() {if (this.is_unlocked())
                       {super.inc();}};
  void dec() {if (this.is_unlocked())
                 {super.dec();}};
}
```

Methods which are unaffected by interactions are not explicitly lifted, e.g. `top` and `size`. Note that the lifting to the lock feature is schematic. Hence it is tempting to allow default lifters, as discussed in Section 6.4.

The modular specification of the three features, separated from their interactions, allows for several object compositions:

- Stack with counter
- Stack with lock
- Stack with counter and lock
- Counter with lock

For all these combinations, the three lifters shown above adapt the features appropriately to the chosen combination. The resulting objects behave as desired. In addition, we can of course use each feature individually (even lock). With the remaining two features, bound and undo (shown later), many more combinations are possible in the same way. Recall that the composition of lifters and features was shown in Figure 2.3 for an example with three features.

The composed object simply provides the functionality of all selected features to the outside, but for composition we need an additional ordering. In particular, the outermost feature is not lifted, similar to the lowest class in a class hierarchy, whose functions are not overridden.

Although inheritance can be used for such feature combinations, all needed combinations, including feature interactions, have to be assembled manually. In contrast, we can (re)use features by simply selecting the desired ones when creating an object.

In the above example, each feature can be run independently. In other examples it is often needed to write a feature assuming that some other feature is available. For this, a feature declaration may require other features, e.g. in the following example:

```
feature DisplayAdapter assumes AsciiAdapter {
  void show_window(...) { ... }
  ...  }
```

Consequently, an implementation may use the operations of the feature `AsciiPrintable` in order to produce output on a window system.

In general, the base functionality of a new feature can rely on the functionality of the required ones. This idea of assuming other features is a further difference to usual abstract subclass concepts. (Note that the extended object can obviously have more than just the required features.)

6.2 Translation to Java

To provide a precise definition of our Java extension, we show two translations into Java. The first translation uses inheritance, while the second uses aggregation with delegation. Hence this also serves to compare the feature model with both of these approaches and will highlight two cases where both differ. The equivalence of the two translations is easy to see and similar to the formal comparison of delegation (or aggregation) and inheritance presented in [Ste87].

We assume the following abstract program with

- I_i feature interfaces
- $I_i.t_{k_i}$ methods declared for interface I_i
- F_i corresponding features
- $F_i.vardecls$ declaration of instance variables
- $F_i.f_{k_i}$ code for methods $I_i.t_{k_i}$
- $F_{i,j}$ lifter for F_j to I_i
- $F_{i,j.f_{k_j}}$ code for lifting $I_j.t_{k_j}$

```
interface I_1 {
                // method declarations
    I_1.t_1;
    ⋮
    I_1.t_{k_1};
}
    ⋮
interface I_m {
    I_m.t_1;
    ⋮
    I_m.t_{k_m};
}
feature F_1 implements I_1 assumes I_1^{l_1}, ..., I_1^{l_n} {
    F_1.vardecls  // variable declarations
    I_1.t_1 F_1.f_1;  // method implementations
    ⋮
    I_1.t_{k_1} F_1.f_{k_1};
}
    ⋮
```

```
    // lifters
feature F_i lifts I_j {
    I_j.t_1 F_{i,j}.f_1;   // function redefinitions
    ⋮
    I_j.t_{k_j} F_{i,j}.f_{k_j};
}
```

We use this schematic program to translate concrete object creations into Java in two ways, inheritance and aggregation. For the translations, the feature interfaces are preserved, while the feature code is merged into concrete classes, as shown below.

For sake of presentation, the translation is simplified in order to make the obtained code as explicit as possible. Therefore, we assume the following:

1. The names of (instance) variables as well as method names are distinct for all features.

2. Assume that method calls to *this* are explicit, i.e. always $this.fct(\ldots)$ instead of $fct(\ldots)$.

3. Variable declarations have no initializations.

6.2.1 Translation via Inheritance

For this translation, we create a set of concrete classes, one extending the other, for each used feature combination $F_1(F_2(F_3(\ldots(F_n)\ldots)))$. For such a combination, we create n classes named $F_i\_F_{i+1}\_F_{i+2}\_\ldots\_F_n$ for $i = 1, \ldots, n$. These extend each other and add one feature after another. For instance, $F_1\_F_2\_F_3\_\ldots$ extends $F_2\_F_3\_\ldots$. The class $F_1\_F_2\_F_3\_\ldots$ adds the functionality for interface I_1 and lifters to F_1 for all others.

Formally, an object creation

`new` $F_1(F_2\ldots(F_n)\ldots)$

translates to

`new` $F_1\_F_2\_\ldots\_F_n$

Furthermore, we need the following Java classes for $i = 1, \ldots, n$:

```
class F_i_F_{i+1}_..._F_n extends F_{i+1}_..._F_n
    implements I_i, ..., I_n {
                // Feature i implementation
    F_i.vardecls    // variable declarations
    I_i.t_1 F_i.f_1;    // function implementations
    ⋮
    I_i.t_{k_i} F_i.f_{k_i};
```

```
                    // Lift Feature i+1 to i
    I_{i+1}.t_1 F_{i,i+1}.f_1;// function redefinitions
    ⋮
    I_{i+1}.t_{k_{i+1}} F_{i,i+1}.f_{k_{i+1}};
      ⋮             // Lift Feature n to i
    I_n.t_1 F_{i,n}.f_1;   // function redefinitions
    ⋮
    I_n.t_{k_n} F_{i,n}.f_{k_n};
}
```

The above schematic code implements feature i and, using lifters, possibly overrides the features in the extended class. A concrete example for the way lifters are composed is presented below. Recall also Figure 2.3 for this example. Observe that $n - i$ lifters are needed, which may call methods of the super class.

We have so far assumed that the features required for F_i via `assumes` are present in $F_{i+1}, \ldots, F_n$ for a feature combination $F_1(F_2 \ldots (F_i \ldots (F_n) \ldots)$. This restriction can be relaxed if we assume that the required ones are present in $F_1, \ldots, F_{i-1}$, similar to virtual functions which are only defined in a subclass. However, this does not work with this translation. The translation assumes that the features required for F_i via `assumes` are present in the extended class. In the class created for the combination $F_1, \ldots, F_i$, this however entails that undeclared identifiers occur in the translated code. This might only be allowed in a dynamically typed language with dynamic method lookup. Interestingly, this assumption is not needed for aggregation, which accounts for a small difference between the two translations. Another difference will be examined in the following section on parameterized features.

For instance, the three features from the introduction translate into the following class hierarchy, if an object of type `LF (CF (SF))` is used.

```
class SF implements Stack {
  String s = new String();
  void empty() { s = "";}
  void push( char a)
         {s = String.valueOf(a).concat(s);};
  void pop() {s = s.substring(1); } ;
  char top() { return (s.charAt(0) ); } ;
  void push2( char a)
         {this.push(a) ; this.push(a); };
}
class CF_on_SF extends SF
                implements Counter, Stack {
  int i = 0;
  void reset() {i = 0; };
```

```
  void inc() {i = i+1; };
  void dec() {i = i-1; };
  // lift SF to CF
  void empty() {this.reset(); super.empty();};
  void push( char a)
              {this.inc(); super.push(a) ;};
  void pop() {this.dec(); super.pop() ;};
}
class LF_on_CF_on_SF extends CF_on_SF
            implements Lock, Counter, Stack {
  boolean l = true;
  void lock() {l = false;};
  void unlock() {l = true;};
  boolean is_unlocked() {return l ;};
  // lift CF to LF
  void reset() {if (this.is_unlocked())
                    {super.reset(); }};
   void inc() {if (this.is_unlocked())
                    {super.inc(); }};
   void dec() {if (this.is_unlocked())
                    {super.dec(); }};
 // lift SF to LF
  void empty() {if (this.is_unlocked())
                    {super.empty(); }};
  void push( char a) {if (this.is_unlocked())
                      {super.push(a);}}
  void pop() { if (this.is_unlocked())
                      {super.pop();}};
}
```

In this example, the above code provides for most sensible combinations, except for stack with lock only or counter with lock. In general, this translation introduces intermediate classes which may be reused for other feature combinations.

6.2.2 Translation via Aggregation

Aggregation is a common technique for composing objects from different classes to a larger object. It is used in some object-based systems as a replacement for inheritance [US91].

This translation requires a set of base implementations and one new class for each feature combination. The idea of the translation is to create a class, where, for each selected feature, one instance variable of this type is used to delegate the services, similar to [JZ91]. We have to be careful with delegation and calls to `this`, which

should not be sent to the local object. Hence we have to supply the delegate object with the right pointer to the enclosing object, which "replaces" `this`. For this purpose, we create a base class for each feature implementation with an extra variable which will point to the composed object. This construction enables us to handle the extension discussed for the above translation. Here, we can check the `assumes` clauses wrt the full set of features, and not just wrt the lower ones in the composition order. With the inheritance translation we had to check these assumptions for each newly added class wrt its superclass.

Unlike the first translation, we need a few further technical assumptions. For all lifters, all methods are lifted explicitly, e.g.

```
int size() { return super.size(); };
```

is assumed to be present. Furthermore, we need to assume that instance variables which are used in lifters are declared public.[1] Also, the name self may not be used.

The main task of this translation is to compose the lifters, i.e. all lifters for one method have to be merged at once here. This can lead to more dense code, as all needed lifters are composed in one class, contrary to the inheritance translation.

An object creation

`new` $F_1(F_2 \ldots (F_n) \ldots)$

translates to

`new` $F_1\_F_2\_\ldots\_F_n$

For this, we first need the following base classes for each feature implementation F_i, $i = 1 \ldots n$. For the type of self in the code below, we use the class $F_1\_F_2\_\ldots\_F_n$. If no assumes statements are used, then just I_i is sufficient and the class can be reused for other object creations. Alternatively, one can introduce an intermediate class with just the needed interfaces $I_i, I_i^1\_\ldots\_I_i^{l_i}$.

```
class F_i implements I_i {
            // reference for delegation
    (F_1_F_2_..._F_n) self ;

            // reference for virtual functions,
            // based on required feature list
            // constructor for this class
    F_i (F_1_F_2_..._F_n  s) { self = s; } ;
    F_i.vardecls
            // function implementations
    I_i.t_1 θF_1.f_1;
```

[1] Note that public declarations are omitted throughout this presentation.

$\vdots$
$I_i.t_{k_i}\ \theta F_1.f_{k_i};$
}

For delegation to work in the above, we need to apply a substitution θ which renames this to self:

$$\theta = [\text{this} \mapsto \text{self}]$$

With the above base implementations we construct the class $F_1\_F_2\_\ldots\_F_n$ via aggregation.

```
class F_1_F_2_..._F_n implements I_1, I_2, ..., I_n {
                // delegate objects
    F_1 b_1 = new F_1(this);
    ⋮
    F_n b_n = new F_n(this);
                // now need to nest lifters
    I_2.t_1 δ_2 F_{1,2}.f_1; // lift feature 2 to 1
    ⋮
    I_2.t_{k_2} δ_2 F_{1,2}.f_{k_2};

                // lift feature 3 to 1
    I_3.t_1 δ_3 θ_{2,3} F_{1,3}.f_1;
    ⋮
    I_3.t_{k_3} δ_3 θ_{2,3} F_{1,3}.f_{k_3};
       ⋮
                // lift feature n to 1
    I_n.t_1 δ_n θ_{n-1,n} θ_{n-2,n} ... θ_{2,n} F_{1,n}.f_1;
    ⋮
    I_n.t_{k_n} δ_n θ_{n-1,n} θ_{n-2,n} ... θ_{2,n} F_{1,n}.f_{k_n};
}
```

For simplicity, we only indicate the applications of nested lifters via unfolding operators θ_i, where $\theta_{i,j}$ unfolds the lifter from j to i, sketched as

$$\theta_{i,j} = [\texttt{super}.f_1 \mapsto F_{i,j}.f_1, \ldots, \texttt{super}.f_{k_i} \mapsto F_{i,j}.f_{k_j}],$$

and also passes the actual parameters. Unlike in the examples below, unfolding is in general more involved for functions, as we cannot have local blocks with return statements. Hence we also assume for simplicity that methods return `void`.[2]

[2]This is no restriction, as in Java objects of primitive type can be "wrapped" into an object in order to be passed as variable parameters.

Furthermore, we have to delegate calls to `super` to the delegate objects. For this purpose, δ_i shall rename the instance variables and method calls of methods in I_i to `super` correctly to the corresponding b_i. For instance, `super.pop()` is translated to `sf.pop()`, where `sf` is the name of the instance variable in the following example. We show the translation for the combination of the three introductory features. First, new base classes are introduced (with suffix `_ag`):

```
class SF_ag implements Stack {
  Stack self;
  String s = new String();
  SF_ag(Stack s) {self = s;};
  void empty() { s = "";}
  void push( char a)
        {s = String.valueOf(a).concat(s);};
    // self replaces this for delegation!
  void push2( char a)
        {self.push(a); self.push(a);};
  void pop() {s = s.substring(1); };
  char top() { return (s.charAt(0)); };
}
class CF_ag implements Counter {
  Counter self;
  CF_ag (Counter s) {self = s;};
  int i = 0;
  void reset() {i = 0; };
  void inc() {i = i+1; };
  void dec() {i = i-1; };
  int size() {return i; };
}
class LF_ag implements Lock {
  Lock self;
  LF_ag (Lock s) {self = s;};
  boolean l = true;
  void lock() {l = false;};
  void unlock() {l = true;};
  boolean is_unlocked() {return l;};
}
```

A class for a composed object is shown below.

```
class LF_CF_SF implements Lock, Counter, Stack
{
           // delegate objects
  SF_ag sf = new SF_ag(this);
```

```
  CF_ag cf = new CF_ag(this);
  LF_ag lf = new LF_ag(this);
           // delegate to lock
  void lock() {lf.lock();};
  void unlock() {lf.unlock();};
  boolean is_unlocked()
    {return lf.is_unlocked();};
         // delegate to lock
  void reset()
    {if (this.is_unlocked()) {cf.reset();}};
  void inc()
    {if (this.is_unlocked()) {cf.inc();}};
  void dec()
     {if (this.is_unlocked()) {cf.dec();}};
  int  size()
     {return cf.size();};
         // delegate to stack
  void empty()
      {if (this.is_unlocked())
          {this.reset(); sf.empty();}};
  void push( char a)
        {if (this.is_unlocked())
           {this.inc(); sf.push(a);}};
  void push2( char a) {sf.push2(a);};
  void pop() {if (this.is_unlocked())
              {this.dec(); sf.pop();}};
  char top() {return sf.top();};
}
```

Compared to the first translation, we need fewer classes here, as the base classes can be reused. On the other hand, aggregation introduces another level of indirection which may affect efficiency.

6.3 Parametric Features

In order to write reusable code, it is often desirable to parameterize a class by a type. In this section, we introduce parametric features, which are very similar to parametric classes. Due to the flexible composition concepts for features, we also need expressive type concepts for composition. For Java, parametric classes have recently been proposed and implemented in the language Pizza [OW97], which will be the target language for our translations. Apart from other nice extensions, which are also used in some examples here, Pizza introduces a powerful extension for type-safe parameterization. The notation for type parameters is similar to C++ templates [Str91]. A typical

example is a stack feature parameterized by a type parameter `A`, written as `<A>`, as follows:

```
interface Stack<A> {
   void empty();
   void push( A a);
   void push2( A a);
   void pop();
   A top();
}
feature SF<A> implements Stack<A> {
           // Use Pizza's List data type
   List<A> s = List.Nil;
   void empty() { s = List.Nil;};
   void push(A a) {s = List.Cons(a,s);};
   void push2(A a)
          {this.push(a) ; this.push(a);};
   void pop() {s = s.tail();};
   A top() { return s.head();};
}
```

Stacks over type char with a counter are then created via

```
    new CF (SF<char>);
```

Note that it is sometimes useful to make assumptions on the parameter for providing operations, e.g.

```
 interface Matrix<A implements Number> {
   void multiply_matrix( ...);
   ...
 }
```

Such an assumption is different from assumptions via `assumes`, as it refers to a parameter and not to the inner feature combination. The difference is that this kind of parameterization is not subject to liftings.

For translating parameterization into Java we refer to [OW97]. Here, we only aim at translating into Pizza. As we mostly use basic concepts, it is not necessary to go into the details of the Pizza type system. We do however use another convenient, but not essential, extension provided by Pizza: algebraic data types, which are also called class cases. With algebraic data types, we can easily define basic data structures such as lists used above. The following class declaration introduces a class of list elements which are either the empty list `Nil` or a cons node.

```
 class List<A> {
    case Nil;
    case Cons(A head, List<A> tail);
```

For such a data type, an appropriate switch statement can be used, for instance for computing the length of a list:

```
int length(List<A> x) {
        switch (x) {
        case Nil: return 0;
        case Cons( A el, List<A> xs):
                  return 1 + xs.length();}
 }
```

6.3.1 Type Dependencies

For parameterized features new and interesting specification problems occur when combining features. Not only can features depend on each other, but the parameter types can also depend on each other. This gets even more complicated if more than two features are involved, as shown below. For instance, we may want to combine `Stack<A>` with a feature which only allows elements within a certain range. Its implementation maintains two variables of type `A` used for filtering. This feature `Bound` is parameterized by a numeric type:

```
interface Bound<A> {
  boolean check_bounds(A el);
}

feature BF<A implements Number>
        implements Bound<A> {
  A min, max;
  BF(A mi, A ma) { min = mi; max = ma;};
  boolean check_bounds(A el) {...};
}
```

Clearly, we can only combine the two features when both are supplied with the same type. This can be expressed by liftings:

```
feature BF<A> lifts Stack<A> { ... }
```

Another example for such a dependency will be shown in Section 6.6. Note that in feature implementations, assumes conditions can also express type dependencies in the same way.

6.3.2 Multi-Feature Interactions and Type Interactions

In the following, we discuss multi-feature interactions and type interactions using the undo example. The problem and the solution correspond to the functional version in

Section 5.5. Here, we introduce a new language construct to cope with the new aspects of lifting features, i.e. that lifting may change the type parameter.

The implementation of the undo feature is simple: save the local state of the object each time a function of the other features is applied (e.g. push, pop). Undo depends essentially on all "inner" features, since it has to know the internal state of the composed object. As we work in a typed environment, the type of the state to be saved has to be known. This multi-feature interaction is resolved by an extra feature, called `Store`, which allows to read and write the local state of a composed object. (The motivation for store is similar to that of the Memento pattern in [GHJV94].)

We introduce the following interface for `Store`:

```
interface Store<A> {
   void put_s( A a);
   A get_s();
}
```

Note that the parameter type depends on the types of all instance variables of the used features. Consider for instance adding this feature to a stack with counter. Then for both features the local variables have to be accessed.

With the `Store` feature, we reduce the multi-feature interaction to a type interaction problem. This means that the parameter type of a feature has to change when a feature is lifted. The following solution makes these type dependencies explicit. We use polymorphic pairs via the Pizza class `Pair<A, B>`, defined as

```
class Pair<A,B> {
    public case Pair(A fst, B snd);
    ...
}
```

This class is useful for type composition. In the lifter below, we state that the inner feature combination supports feature `Store<A>` for some type `A`. For this, we need a new syntactic construct, namely `assumes inner`. As feature stack `ST` adds an instance variable of type `List<B>`, we can support the store feature with parameter `Pair<List<B>,A>`.

```
feature ST<B> lifts Store<Pair<List<B>,A>>
              assumes inner Store<A> {
   Pair<List<B>,A> get_s()
      {return Pair.Pair( s,    // local state
          super.get_s()); }    // inner state
   void get_s(Pair<List<B>,A> s) { ... }
}
```

The `assumes inner` however has some constraints. The lifted feature may not have instance variables or calls to self where the changed type parameter type is used. (This

can be allowed if the type change is a specialization, which this is not the case in this example.)

This inner condition is implicit in other lifters and is only needed if the type parameters change. The lifting

```
feature F lifts F1<A> { ... }
```

can be seen as an abbreviation for

```
feature F lifts F1<A>
          assumes inner F1<A> { ... }
```

We show below how this change of parameters affects the two translation schemes of Section 6.2. Continuing with the example, we express that the counter `CF` adds an integer and `LF` a boolean variable with the following lifters:

```
feature CF lifts Store<Pair<A, int>>
           assumes inner Store<A> {
   ...
}
feature LF lifts Store<Pair<A, Boolean>>
           assumes inner Store<A> {
  ...
}
```

With the above lifters, we can assure that the store feature works correctly and with the correct type for any feature combination. All we need to add is a base implementation for store. As the base implementation cannot store anything useful, we introduce a Pizza type/class `Void`, which has just one element, `void_el`.

```
class Void { case void_el; }
                    //  base implementation
feature ST implements Store<Void> {
  void put_s(Void a ) {};
  Void get_s() {return Void.void_el; };
}
```

With the store feature, we can now write the generic undo feature, which can be plugged into any other feature combination. It is important for the undo feature that the store feature determines the type of the state of the composed object. The undo feature can then have an instance variable of this type. Recall that this is not possible for store, as the type parameter of store changes under liftings.

The undo feature consists of two parts: storing the state before every change and retrieving it upon an undo call. The latter is the core functionality of undo, whereas the former will be fixed for each function which affects the state via liftings. First consider the undo feature and its implementation, which uses a variable `backup` to

store the old state. Since there may not be an old state, we use the algebraic (Pizza) type `Option<A>`, which contains the elements `None` or `Some(a)` for all elements `a` of type `A`.

```
interface Undo<A> {
   void undo();
}
class Option<A> {
     case None;
     case Some(A value);
}
feature UF<A> implements Undo<A>
              assumes Store<A> {
  Option<A>  backup = None;
  void undo() {
        switch ((Option) backup) {
           case Some(A a):
              put_s( a );
} } }
```

An alternative version of undo may store several or all old states. Due to our flexible setup, we can just exchange such variations.

For each of the other features, we have to lift all functions which update the internal state. As for lock, this lifting is canonical, e.g. for `push`:

```
  void push(A a) {
     backup = Option.Some( get_s());
     super.push(a); };
```

Note that there is an interesting interaction between lock and undo: shall undo reverse the locking or shall lock disable undo as well? We chose the latter for simplicity and hence add lock after undo. Lifting undo to lock is canonical and not shown here. As an example, we can create an integer stack with undo and lock as follows:

```
   new LF (UF<Pair<int,Void>> (SF<int> (ST) ))
```

6.3.3 Translation into Pizza

We show in the following how to translate the above extensions into Pizza. This will reveal another difference between aggregation and inheritance: for inheritance, we cannot cope with the change of parameters. Otherwise, the translation to Pizza is quite simple.

For aggregation, additional inner statements just translate into types of the instance variables of class generated for a combination. This is shown in the following code for

a class generated for a composed object with both stack and store features. We first introduce a class `SF_ag<A>` for parametric stacks. As we do not allow calls to self for features whose type parameter changes during lifting, we do not use the usual delegation mechanism in the above class. Hence we use just `ST`. The class `SF_ST<A>` exports the interface `Store<Pair<List<A>,Void>>`, but uses a delegate object with interface `Store<Void>`.

```
class SF_ag<A> implements Stack<A> {
   Stack<A> self ;
   SF_ag(Stack<A> s) {self = s;};
   List<A> s = List.Nil;
   void empty() ...
}
class SF_ST<A> implements Stack<A>,
                   Store<Pair<List<A>,Void>>{
   SF_ag<A>    sf = new SF_ag(this);
   Store<Void> st = new ST();
   Pair<List<A>,Void> get_s()
        {return Pair.Pair(sf.s, st.get_s());};
   ...
}
```

A further detail to observe is that all type variables have to be considered for the translation. This means that for the newly introduced class, all type variables which appear as parameters in the desired set of features have to appear as parameters. For instance, for the combination `F<A>(G<B>)`, we need a class `F_G<A,B>`.

For inheritance, an inner statement is an assumption on the extended class. If the parameter changes, this amounts to specialization for parameterized classes, which is problematic in typed imperative languages, as discussed in [OW97]. In Pizza, the problem in this example is that subtyping does not extend through constructors such as `List`. For instance, we cannot translate the above feature combination to the following (illegal) code:

```
        // illegal ! Type conflict!
class ST_SF<B,A> extends ST<A>
     implements Stack<B>,
                Store<Pair<List<B>,A>> {
    Pair<List<B>,A> get_s() {
        Pair.Pair( s,          // local state
               super.get_s()); // inner state
   }
   ...
}
```

If the parameters do not change, the translation is straightforward.

6.4 Generic Liftings via Higher-order Functions

Generic liftings are possible via higher-order functions, which are available in Pizza. In the stacks example it is easy to see that lifting to lock is schematic. It is natural to express this by higher-order functions. We use the Pizza notation `()->void` for a function type. Pizza function types can be polymorphic as well and are, in general, written as

```
(A1, A1, ..., An)->A0,
```

where `Ai` are types.

As an example of a generic lifting, consider for instance the following function for lifting lock:

```
void lift_to_lock( ()->void f)
       { if (this.is_unlocked() ) { f() ;};}
```

It can be used e.g. with

```
void reset() {lift_to_lock(super.reset);};
```

which replaces

```
void reset()
  {if (this.is_unlocked()) {super.reset();}};
```

It can be useful to extend the language to allow default lifters, which are applied if no explicit lifters are provided.

Note that there is a small problem with generic lifters: We cannot write a generic lifter which suits both functions and procedures (without return values) in Java. This is of course only useful in cases where the return value of the lifter is generic as well. Consider for instance the above lifter: it cannot be applied to lift the function pop. A lifter for this case would look like:

```
void lift_fun_to_lock( ()->char f)
           {return  f() ;}
```

This code may replace the lifters for `top` and `size` to Lock. (Note that disabling these two functions via lock is more difficult. What shall these lifted functions return? The only two options are raising an exception instead or returning an arbitrary or default value.)

6.5 Features and Exceptions

So far, we have used lifters to adapt features to the context of other features, which usually meant additional state in the form of instance variables. Yet features may equally well add exceptions, as shown in Section 4.3. In the setting of Java, exceptions are already provided, but are too limited for our purpose as discussed below.

Consider for instance the stack example and adding exception handling for stack underflow. The idea is to add a feature which does nothing but raising the appropriate exception. This feature can be added or omitted as desired.

```
  // exception for underflow
class Underflow extends Exception {
}
interface UnderflowI {
   void display_exception();
}
feature Underflow implements UnderflowI {
  void display_exception()
    {System.out.println("Stack Underflow!");};
}
```

In the above example, the underflow feature only implements some auxiliary functions, for instance the function `display_exception`. (For a further example see Section 6.6.3.) The exceptions are raised in the lifters:

```
feature Underflow lifts Stack {
  public void pop() throws Underflow
   { if (is_empty()) throw new Underflow();
                     else super.pop(); }
}
```

This example shows that the feature model easily accommodates exceptions, but in Java a small language extension is useful. Java requires to declare an exception for any method whose body may raise one, and also in interfaces for which one implementation raises an exception. This rigid and fully explicit declaration of exceptions in Java can be used for feature combination, but affected methods are marked, if the exception feature is used or not. For instance, in the above example, the underflow exception has to be declared for the `pop` function at every occurrence. As we can use lifters to add exceptions, this seems too strict. Hence we argue that only the lifters should declare exceptions in this case. The translation to Java in turn has to add declarations of exceptions to function definitions appropriately. This may require creating a class twice, with and without exception handling. As the main idea is straightforward, we refrain from presenting this in detail.

6.6 Examples

In the following sections we sketch a few typical applications for feature-based programming. These range from generic application frameworks to application domains where change and flexibility are predominant.

6.6.1 Variations of Design Patterns

We show in the following that for several typical programming schemata, coined design patterns [GHJV94], feature-based implementations provide high flexibility and the desired reusability. This is particularly important if several features or design patterns are combined. The following examples are freely taken from standard literature on design patterns [GHJV94].

Proxy Pattern

Consider implementing some functional entity, e.g. sets, where caching of the results of operations is a viable option. In the lines of [GHJV94], this can be viewed as a Proxy pattern. Clearly, a cache is an independent feature, and there exist many variations of caching. For instance, considering the data structures used and the replacement strategy. And it furthermore may depend on appropriate hash functions, which could also be provided via features.

When writing a reusable set of caching modules, the various cache implementations just implement the data structures and the access functions. Interaction resolution in turn modifies the access operations for the object to be cached and determines the type dependencies.

Consider writing this with classical object-oriented languages: for each needed combination of a cached object, a cache, and a hashing function, a new (sub-)class has to be implemented.

A sketch of such an example is shown below. It shows how to add a cache to the parametric features `Set<A>` and `Dictionary<A, B>`. For caching, these data structures are viewed as mappings, from `A` to boolean and `A` to `B`, respectively. The feature implementation `CacheI<A,B>` (whose interface `Cache` is not shown here) caches mappings from `A` to `B`.

```
interface Set<A> {
   void put( A a);
   boolean contains(A a);
}
interface Dictionary<A, B> {
  Option<B> get(A key);
  void put(A key, B value);
}
```

```
feature CacheI<A,B> implements Cache<A,B> {
   ...
   void put_s( A a, B b) {...} ;
   boolean find_s(A a) {...};
   B get_s() {...};
}
feature CacheI<A,B> lifts Dictionary<A, B> {
            // adapt access functions to cache
  Option<B> get(A key)
    {if find(key) return Option.Some(get_s());
            else  return super.get();  }
  ...
}
      // second parameter is just boolean here
feature CacheI<A,boolean> lifts Set<A> {
  ...
}
```

Note that the lifters express the type dependencies.

Adaptor Patterns

The adaptor design pattern [GHJV94] glues two incompatible modules together. This design fits nicely in our setting, as adaptors should be reusable. Typically, there is some core adaption functionality, e.g. some data conversion, which we model as a feature. When adding this to another feature, we can just lift the incompatible functions with help of the core functionality.

An example is converting big endian encoding of data to little endian. For instance, if we output data on a (low-level) interface which needs big endian, but we work with little endian, such a conversion feature can easily be added. The adaptor feature provides the core functionality, here the data conversion, and interaction resolution adapts the operations of the object.

The following features and lifters sketch the solution of pluggable adaptors with features. The adaptor feature `Big_to_little_endian` adds a conversion function, which is used in the lifter to provide the put method with big endian data input.

```
feature Big_to_little_endian {
            // convert to little endian
  int big_to_little(int a) {...};
}
             // assumes little_endian
interface low_level_IO {
  void put(int a);
}
```

```
feature Big_to_little_endian
        lifts low_level_IO {
 void put(int a)
     {super.put( big_to_little(a)); };
}
```

There is an interesting optimization if the cache feature is used with the above adaptor. The interaction handling for the two feature as it is not needed to convert the cached data to big endian.

6.6.2 Automata-based Modeling Techniques

Diagrammatic modeling techniques have gained large interest in the last years. We discuss here automata-based description techniques, where we aim at composing hierarchical automata from smaller automata parts. In particular, we discuss interactions arising in such descriptions. We follow an example with was studied by Teege in [Tee94, Tee96] as a subset of a group-ware application. In this example, hierarchical automata are used to describe the status of a document. As discussed in [Tee94], there arise feature interactions, which are not easily modeled by simple automata concepts. We show here how to model automata and the interactions by feature-oriented programming. The main benefit we gain is composition of automata, where our model of feature interactions applies. Note that features may provide the basis for developing more advanced automata composition concepts, which possibly permit graphic notation. This is discussed below.

The idea of this example is to specify smaller automata as features and to compose them with the techniques of feature-oriented programming. This enables flexible composition, which is the main difference to the many other techniques for graphic description of software components.

In this example, documents are manipulated via a groupware application which admits several options or features for controlling the access to the documents. These features can be set individually for each document. They usually model the current state of each document to control the allowed operations. We model the following, very basic features as automata:

- DocAut(omaton) with states `existent` and `undef`, and a transition (or function) `init`.
- ChangeAut with states `changeable` and `fixed` and a transition `fix`. ChangeAut assumes DocAut and defines a sub-automaton of state `existent`. This can be viewed as a refinement of a state.
- ErasableDoc assumes also DocAut and defines a function `erase`.

The combination of these three features is shown in Figure 6.1. The full application contains a larger set of features. The motivation for this example is to select the desired features individually for each document.

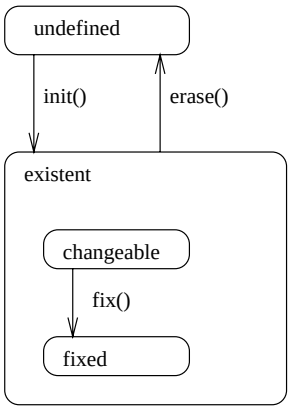

Figure 6.1: Hierarchical Automaton for the Status of a Document

The interaction problem of this example is clearly between ErasableDoc and ChangeAut. If ChangeAut is in state `fixed`, then erase should not be possible. This is achieved here by lifting ErasableDoc to ChangeAut. As shown below, we disallow erase if the local state is `fixed`.

Note also that hierarchical automata pose another problem which can be viewed as an interaction. The problem arises since a sub-automaton is only active, if the global automaton is in the corresponding, refined state. In case of a global transition to a new state, which has been refined to a sub-automaton by some other feature, it is unclear if the sub-automaton should be (re-)initialized or if its old state be preserved.

The following code defines the base document with two global states. The feature implementation requires another feature `BA`, as indicated with the assumes clause. Feature `BA` allows `DA`, the implementation of DocAut, to use some basic functionality of `BA` to construct automata. Hence the method `newstate` can be used to define a new state. Its details shall not be discussed here. In general, the base functionality of a new feature can rely on the functionality of the required ones.

```
interface DocAut {
  void init();
  final int undef;
  final int existent;
}

feature DA implements DocAut assumes BA {
  final int undef = newstate();
  final int existent = newstate();
        // default state
  DA() {state = undef;};
```

```
  void init()
     {if (state==undef) state = existent; };
}
```

The second component, ErasableDoc, just adds one transition. As it adds no states, its implementation requires `DA` in order to implement the method. The lifting of DocAut is trivial (and could be omitted).

```
interface ErasableDoc   {
   void erase();
}

feature EA implements ErasableDoc assumes DA{
   void erase()
     {if (state==existent) state = undef; };
}

feature EA lifts DocAut {
   void init() {super.init(); };
}
```

The definition of ChangeAut is more involved, since it defines a local automaton, named `local_aut`, with two states. Its constructor `CA` initializes its local state.

```
interface ChangeAut {
  void fix();
}

feature CA implements ChangeAut assumes BA {
  // CA refines a state,
  // hence uses a local automaton
   BA local_aut = new BA();
   final int changeable =
               local_aut.newstate();
   final int fixed =
               local_aut.newstate();
        // initialize with default state
   CA(){
       super();
       local_aut.state = changeable;};

   void fix() { local_aut.state = fixed;};
}
```

The interesting aspects of ChangeAut are defined in its lifters. To lift DocAut, we turn the local automaton into an initial state, if `init` is invoked. Furthermore, to solve the interaction between with ErasableDoc, we redefine `erase`.

```
feature CA lifts DocAut {
   // lift DA: initialize
   void init() {
              // changeable is default state
         local_aut.state = changeable;
         super.init(); };
}

feature CA lifts ErasableDoc {
   // lift EA:
   void erase() {
                // disable erase if fixed
         if (local_aut.state != fixed)
              super.erase(); };
}
```

For the full example with more, but similar interactions, we again refer to [Tee96]. Note that the simple implementation of automata is sufficient for our purpose. For pure automata, there clearly exist more efficient implementations. However, for many software applications, automata just pose as a skeleton of the actual program, which is completed as a further step. For this purpose, our implementation is more appropriate, as it is extensible.

We have shown that typical composition problems of automata descriptions can be modeled by our feature model. Clearly, when working with automata, a graphical notation for interaction resolution is desirable. For some of the typical interaction cases it is possible to define graphical equivalents.

Other specification concepts with automata can be found in [HN96, Tee94]. For instance, the two interactions above are supported by special purpose languages with graphical notation. The first problem of disabling a transition is possible in an object-oriented variant of Statemate [HN96] and in Hierastates [Tee96]. In both languages, the local transition has precedence over the global.

The other problem regarding the life-time of local states is resolved by particular annotations in Statemate [HN96]. A typical problem of such simple notations is that certain aspects cannot be modeled, e.g. that only particular global transitions reset the local state.

Our main point here is that common automata concepts allow for typical composition operations on automata, but do not consider interactions between components explicitly, as done here. With the concepts developed here, it is possible to compose just the wanted features/automata, where interactions are resolved as specified. It

remains for future work to extract a broad set of typical interactions and to devise graphical notation for them.

6.6.3 Feature Interaction in Telecommunications

A well known problem in feature interaction stems from the abundance of features telephones (will) have. For instance, (re-)consider the following conflict occurring in telephone connections: B forwards calls to his phone to C. C screens calls from A (ICS, incoming call screening). Should a call from A to B be connected to C? In this example, there is a clear interaction between forwarding (FD) and ICS, which can be resolved in several ways. For many other examples we refer to [CGN+94].

Similar to Section 5.7, we demonstrate our techniques with the following set of features for this domain of connecting calls:

- Forwarding of calls
- ICS (incoming call screening)
- OCS (outgoing call screening)

Although ICS and OCS look very similar, there are small differences. For instance, they may interact differently with other features, as shown below.

The Basic Phone

The basic building block is a feature `Phone`, which provides a function `connect` for computing the phone reached by some dialed number. In addition, we use a simple technique for adding the actual services to the full object. Each feature adds its functionality by extending a function `dispatch` (for feature dispatch), which is used by `connect`. As we use exceptions for modeling a busy signal, the function `dispatch` may throw an exception, as shown in the following Java interface description:

```
interface Phone {
  int connect(int dest) ;
  int dispatch(int dest) throws Busy ; }
```

The implementation provides for a trivial connect functionality, just in order to set the stage for other features. Note that we need an explicit constructor `Ph` here, which initializes the instance variable used for the origin of a call.

```
class Ph implements Phone {
    // origin of call
  int o;
    // function for creating
    // and initializing objects
```

```
  Ph(int orig) { o = orig;} ;
        // trivial dispatch
  int dispatch(int dest) throws Busy
     {return dest;};

  int connect(int dest) {
    try{        return dispatch(dest); }
    catch(Busy B)
       { println("Busy " ); return 0; } }
}
```

Forwarding

Next we add a feature for forwarding with the following interface:

```
interface Forward {
  boolean fd_check(int i);
  int forward(int dest);  }
```

The code is again as simple as possible, just forwarding selected numbers to the next number via an auxiliary function `fd_check`.

```
feature FD implements Forward {
   // aux. function
  boolean fd_check(int i)
   // naive check if forwarding is desired
     { return (i == 5 || i == 7 || i == 10 ||
                i == 11 || i == 12); };
  int forward(int i) { return i+1; };
}
```

To integrate the service, we lift the `dispatch` function in the following lifter:

```
feature FD lifts Phone {
  int dispatch(int dest) throws Busy {
     if (fd_check(dest))
                //  recursive forwarding !
          return connect(forward(dest));
     else return super.dispatch( dest);
   }
}
```

Note that the above either calls `super.dispatch` in order to invoke (possibly) other features, or calls `connect` to attempt a recursive connect attempt. (This recursive forwarding is not limited, for simplicity.)

Incoming Call Screening

For ICS we use a simple check for each call and raise the exception busy if ICS disallows a call. The structure of the following code is as above.

```
interface IcsI {
   int ics(int dest) throws Busy ;
}

feature Ics implements IcsI assumes Phone{
  Ics(int orig) {super(orig); };
     // aux. functions
  boolean ics_check(int i) {
             // no calls from 5 to 8
    return (o == 5 & i == 8);  };

  int ics(int dest) throws Busy {
    if (ics_check(dest)) throw new Busy();
    else return dest;};
}

feature Ics lifts Phone {
  // lift Phone
  int dispatch( int dest) throws Busy {
             // add ICS service
        return super.dispatch( ics(dest) ) ;};
}
```

Resolving the ICS/Forward-Interaction

To resolve the interaction between forwarding and ICS, we lift the forward function to ICS. We chose the lifting for forward as follows:

```
feature Ics lifts Forward {
  // lift forward
 int forward(int dest) {
  // update origin (also ok if not forwarded!)
       o = dest;
       return super.forward(dest); };
}
```

In case of forwarding over several hops, ICS is checked for each (intermediate) hop wrt the next hop. If only a check wrt the origin of the call is desired, one just has to adapt the lifter (and not to update the origin of the call). Thus, lifting allows for a modular resolution of the interaction between two features.

Outgoing Call Screening

OCS is quite similar to ICS, but we model interaction with forwarding differently: OCS should always be checked from the initial phone to the final destination. With this choice OCS can be modeled similar to ICS, but with a different interaction code. We only show the interface here for brevity:

```
interface OcsI {
   int ocs(int dest) throws Busy;
}
```

Examples

It is now possible to create objects with any subset of the three features ICS, OCS and FD. For instance, with the object `con` created by

```
con = new Ics (FD (Ph 5))),
```

`FD` and `ICS` are selected and the originating phone is set to 5.

With the above code and settings, we obtain the following examples for a connect call from phone 5 with all features:

```
   // just gives 6
 println( con.connect(5));
   // just gives 6
 println( con.connect(6));
   // gives 8, as forwarded to 8,
   // which is allowed by ICS
 println( con.connect(7));
   // gives Busy, as forwarded to 8,
      which is not allowed by ICS
 println( con.connect(8));
   // gives 13, as forwarded twice
 println( con.connect(11));
```

It is easy to imagine other features and also variations of the above features. With our approach such features can be composed in a flexible way. This also provides for a clear structure of their dependencies, which is needed if the number of complementary or alternative features grows.

6.7 Extensions

We discuss in the following a few extensions and issues which have not been addressed so far. As we have focused on feature composition, several interesting aspects have been neglected.

- We have not discussed visibility of attributes of features here, as this does not belong to our core ideas, but it is clearly another issue to be discussed. Currently, features can use only the interfaces of the assumed features, whereas the instance variables of concrete other features are not visible, for good reason. Lifters however lift a feature interface over a concrete feature constructor. Hence only the instance variables of the concrete feature are visible.

  Hiding some functions is clearly needed in some applications. For instance, when adding the counter to a stack, we may not want to inherit the inc and dec functions, as they may turn the object into an inconsistent state.

- In this presentation, liftings depend on one concrete object constructor and an interface, which can be impractical if the interaction resolution depends on the concrete constructor which provides the methods of the interface. It is easy to imagine an extension for this purpose, but it is currently unclear if such a violation of encapsulation is needed.

- An adaption to distributed objects should also address the problem of inheritance anomaly [MY93, LLNW96], as this addresses similar interaction problems.

- Instead of letting features assume other features, it can also be useful for specification purposes to disallow features, similar to canceling other mixins in [SCD+93].

6.8 Discussion of the Feature Composition Architecture

We discuss in the following the main assumptions and characteristics of our feature composition approach, followed by an overview of related work and some extensions. Our feature model enables the flexible composition of features and lifters, which can greatly reduce complexity. Our main claim is that there is a large class of applications for our expressive feature composition method. Recall that our approach generalizes inheritance and aggregation and integrates nicely with several common language extension.

In general, feature interactions are semantic problems of two or more features (and not a priori of their implementations). Our composition method for features is designed to clarify dependencies and to provide structure and flexibility. In this sense, we do not address the general (semantic) feature composition problem but discuss programming-level composition of feature implementations. No general-purpose technique, including ours, can solve the undecidable problem of detecting semantic feature interactions.

A premise of our approach is that any interactions can be handled for two features at a time. This means that interactions between three or more features cannot be modeled directly if they do not show up between two individual features. In our experience, such cases do not appear frequently in practice. For instance, in the literature on feature

interactions in telecom, the survey in [CGN+94] reports 22 feature interactions, but none of them with three features. (And hence none where the interactions do not show up when considering two features at a time.) When true multi-feature interactions examples have been identified (which do not indicate an inconsistency), there are two options. If the features are highly entangled, it is preferable to write code for particular combinations. Otherwise, there are usually ways to circumvent the interaction or reduce it to two-feature interactions. For instance, a feature can be split into two separate features. Recall for instance the undo feature which assumes the store feature to be added before (wrt the feature order) the features to be undone.

Consider the following example for a true three-feature interaction. Assume an email client with added features for handling priorities of emails (bulk, normal, priority) as follows:

1. Basic email client; send and receive emails
2. Downclass outgoing mails (at least for some users)
3. Discard incoming bulk mail
4. Forward incoming emails to another address

With the assumptions below we construct a true three-feature interaction:

- The basic client does not interpret priorities.
- Downclassing is destructive on the email (no copying).
- Forwarding takes place before delivery of incoming mail.
- Features are composed in the above order. (Which is drawn upside down in earlier pictures.)

With all the above features, an incoming normal message may be forwarded, and hence downclassed to bulk by the downclass feature. As the email is modified to bulk, it is discarded by the incoming bulk filter. In brief, the email is forwarded but never delivered to the client. This undesired interaction does not happen with two features at a time.

Yet, as in most other cases, it is easy to avoid (e.g. by reordering features or different feature implementations). This may incur implementation and runtime overhead in some cases, which is the price for flexibility and robustness.

Compared to the above example, a case which occurs more frequently is when performance optimizations are possible only for particular combinations. For instance, for the combination of the stack, counter, and lock feature, in a call to `push` the locking of the stack is checked twice. Clearly, this small performance loss can be avoided. Some simpler optimizations can be performed by a compiler. For complex ones, one

would need a domain-specific optimizer, as for instance pursued by aspect-oriented programming [KLM+97].

The second major assumption of our approach is that we compose features following an ordering between features. This is given by the feature lifters. Only from the outside interface it is possible to view an object as composed of a set of features. There are several reasons for this ordering. First, it is in the spirit of inheritance and it seems to be the simplest structure capturing the essential object-oriented ideas like inheritance. Second, there are problems when viewing features as unordered citizens. Clearly, when features are fully independent, it is not needed to order them. A simple syntactic extension may alleviate the problem.

Although one can imagine more complex feature composition schemes, we argue that basic language features should be as lean as possible. This favors efficient implementations and clarity to the programmer.

We show next that composition schemes without order have difficulties wrt liftings or inheritance. The problem seems to be similar to known problems with multiple inheritance.

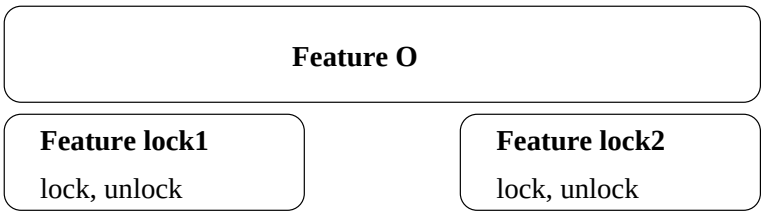

Figure 6.2: Alternative Feature Composition

Consider an example of an object integrating two unordered features, both implementing a lock. Such a configuration with lock1 and lock2, to which a feature O is added, is shown in Figure 6.2. The interaction is that closing lock1 should also close lock2 and vice versa. Hence we need liftings from the two features to feature O. The simple lifting model is to lift the functions of each feature to O, e.g. by applying all lifters corresponding to the other present features. The lifter of lock1 shall call `lock2.lock();` and similarly the lifter for lock2 calls `lock1.lock();`. The problem is which version of lock should be called, the lifted or the original of the feature? If original is called, then all other liftings are ignored, e.g. if other features are involved. Or if the lifted version is called, then the procedure diverges.

An alternative solution would be to introduce a "global" lock function and to lift each local lock to the global one. Then a flexible number of locks can be handled by one access point. This is another example where introducing extra features (and ordering) is useful and leads to a less ambigous structure.

Chapter 7

Concluding Remarks

We have presented a novel model of feature-oriented design, including new programming and specification concepts. For programming, features can be defined individually and are separated from interactions with other features. This is the main difference to other concepts of abstract subclasses or inheritance. Thus it is much more flexible and has a larger potential for reuse. The recent interest in feature interactions, mostly stemming from multimedia applications [Zav93, CO95], shows that there is a large demand for expressive composition concepts where objects with individual services can be created. It also supports that our view of inheritance as interaction is a natural concept.

Compared to classical object-oriented programming, feature-oriented programming provides much higher modularity and flexibility. Reusability is simplified, since for each feature, the functional core and the interactions are separated. This difference encourages to write independent, reusable features and to make the dependencies to other features clear. In contrast, inheritance with overwriting mixes both, which often leads to highly entangled (sub-)classes.

The modular structure of this approach allows to disassemble and to pinpoint the problems of feature interactions. Only with such a modular design, can we hope for adaptable and reusable software. Particularly if the number of features grows, design by features seems preferable over traditional object-oriented design.

Furthermore, we have shown abstract specifications of complex objects with implicit state. We claim that our specification style is suitable for refinement, as it avoids overspecification. We allow to quantify operations/laws over composition schemes, which is essential for abstract reasoning about state and feature combinations. As we deal with references, inheritance and virtual functions, we can cover the essential ingredients of object-oriented systems. Our claim is that we can use well known type systems of functional languages to model binary functions, and do not have to resort to f-bounded polymorphism, as e.g. in [OW97].

Our new contributions to programming and specification concepts open many issues for further studies. We briefly list a few ones. For programming languages, efficient compilation concepts for feature-oriented programming are to be examined. Another

extension is to permit dynamic feature change, i.e. adding and removing features of an object at run time. This may require dynamic type systems without static type safety. For both specification and programming, more extensive practical experience with features and empirical results on reuse and feature variations are desirable. Finally, mechanical proof support for specification and verification can enhance the use of specification concepts.

Bibliography

[AC95] M. Abadi and L. Cardelli. On subtyping and matching. In Walter Olthoff, editor, *ECOOP '95 - Object-Oriented Programming 9th European Conference, Aarhus, Denmark*. Springer LNCS 952, New York, N.Y., 1995.

[AC96] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer-Verlag, New York, 1996.

[AL97] M. Abadi and R. Leino. A logic of object-oriented programs. In Michel Bidoit and Max Dauchet, editors, *TAPSOFT '97: Theory and Practice of Software Development, 7th International Joint Conference CAAP/FASE, Lille, France*, volume 1214 of *Lecture Notes in Computer Science*, pages 682–696. Springer-Verlag, New York, N.Y., 1997.

[Ame90] P. America. Designing an object-oriented programming language with behavioural subtyping. In *Proc. REX/FOOLS Workshop*, Noordwijkerhout, June 1990.

[Ame91] P. America. A behavioural approach to subtyping in object-oriented programming languages. In M. Lenzerini, D. Nardi, and M. Simi, editors, *Inheritance Hierarchies in Knowledge Representation and Programming Languages*, chapter 11, pages 173–190. J. Wiley, New York, NY, 1991.

[Apt81] K. R. Apt. Ten years of Hoare's logic. *ACM Transactions on Programming Languages and Systems*, 3:431–483, 1981.

[AWV93] J. Armstrong, M. Williams, and R. Virding. *Concurrent Programming in Erlang*. Prentice-Hall, Englewood Cliffs, NJ, 1993.

[BA96] L. Bergmans and M. Akşit. Composing synchronization and real-time constraints. *Journal of Parallel and Distributed Computing*, 36(1):32–52, 1996.

[Bar84] H. Barendregt. *The Lambda Calculus, its Syntax and Semantics*. North Holland, 2nd edition, 1984.

[Bar91] H. Barendregt. Introduction to generalized type systems. *Journal of Functional Programming*, 1(2):125–154, April 1991.

[BBM96] V. R. Basili, L. C. Briand, and W. L. Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on Software Engineering*, 22(10):751–761, October 1996.

[BC90] G. Bracha and W. Cook. Mixin-based inheritance. *ACM SIGPLAN Notices*, 25(10):303–311, October 1990. *OOPSLA ECOOP '90 Proceedings*, N. Meyrowitz (editor).

[BCC+96] K. B. Bruce, L. Cardelli, G. Castagna, the Hopkins Objects Group (J. Eifrig, S. Smith, V. Trifonov), G. T. Leavens, and Benjamin Pierce. On binary methods. *Theory and Practice of Object Systems*, 1(3):221–242, 1996.

[BDC+89] T.F. Bowen, F. S. Dworack, C.H. Chow, N. Griffeth, G.E. Herman, and Y.-J. Lin. The feature interaction problem in telecommunications system. In *Software Engineering for Telecommunication Systems (SETS)*, 1989.

[BG97] D. Batory and B. J. Gerac. Composition validation and subjectivity in genvoca generators. *IEEE Transactions on Software Engineering*, February 1997.

[BMR95] A. Borgida, J. Mylopoulos, and R. Reiter. On the frame problem in procedure specifications. *IEEE Transactions on Software Engineering*, 21(10):785–798, October 1995.

[BMR+96] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern Oriented Software Architecture, A system of Patterns*. Addison-Wesley, 1996.

[BPF97] K. Bruce, L. Petersen, and A. Fiech. Subtyping is not a good "match" for object-oriented languages. In *ECOOP '97*. Springer LNCS 1241, 1997.

[Bro95] K. Brockschmidt. *Inside OLE (2nd Ed.)*. Microsoft Press, 1995.

[BS] M. Broy and K. Stølen. *Interactive System Design*. Springer-Verlag. To appear.

[BSG95] K. B. Bruce, A. Schuett, and R. Gent. A type-safe polymorphic object-oriented language. In Walter Olthoff, editor, *ECOOP '95 - Object-Oriented Programming 9th European Conference*, pages 27–51. Springer LNCS 952, 1995.

[BSST93] D. Batory, V. Singhal, M. Sirkin, and J. Thomas. Scalable Software Libraries. In *Proceedings of the ACM SIGSOFT '93 Symposium on the Foundations of Software Engineering*, pages 191–199, December 1993.

[BTCGS91] V. Breazu-Tannen, T. Coquand, C. A. Gunter, and A. Scedrov. Inheritance as explicit coercion. *Information and Computation*, 93(1):172–221, 1991.

[BV94] L. G. Bouma and Hugo Velthuijsen, editors. *Feature Interactions in Telecommunications Systems*. IOS Press, Amsterdam, 1994.

[Cas95] G. Castagna. Covariance and contravariance: Conflict without a cause. *ACM Transactions on Programming Languages and Systems*, 17(3):431–447, May 1995.

[Cas97] G. Castagna. *Object-Oriented Programming: A Unified Foundation*. Birkhäuser PTCS, 1997.

[CGN+94] E. J. Cameron, N. Griffeth, Y.-J. Linand M.E. Nilson, W.K. Schnure, and H. Velthuijsen. A feature interaction benchmark for in and beyond. In Bouma and Velthuijsen [BV94], pages 1–23.

[CL94a] C. Chambers and G. T. Leavens. Typechecking and modules for multimethods. *ACM SIGPLAN Notices*, 29(10), 1994.

[CL94b] Y. Cheon and G. T. Leavens. A quick overview of Larch/C++. *Journal of Object-Oriented Programming*, 7(6):39–49, October 1994.

[CM91] L. Cardelli and J Mitchell. Operations on records. *Mathematical Structures in Computer Science*, 1:3–48, 1991. Also in Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design* (MIT Press, 1994) ; available as DEC Systems Research Center Research Report #48, August, 1989, and in the proceedings of MFPS '89, Springer LNCS volume 442.

[CO95] K. E. Cheng and T. Ohta, editors. *Feature Interactions in Telecommunications III*. IOS Press, Tokyo, Japan, Oct 1995.

[Din97] P. Dini, editor. *Feature Specification and Refinement with State Transition Diagrams*. IOS-Press, 1997.

[DL97] K. K. Dhara and G. T. Leavens. Weak behavioral subtyping for types with mutable objects. *Electronic Notes in Theoretical Computer Science*, 1, 1997.

[Ehr82] H. D. Ehrich. On the theory of specification, implementation, and parameterization of abstract data types. In *Journal of the ACM*, volume 29, January 1982.

[Ehr96] H.-D. Ehrich. Object specification. In B. Krieg-Brückner E. Astesiano, H.-J. Kreowski, editor, *IFIP WG14.3 Book on Algebraic Foundations of Systems Specification*. Springer, 1996.

[EM85] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1*. Springer-Verlag, 1985.

[Fla89] C. Flaviu. Exception handling. In T. Anderson, editor, *Dependability of Resilient Computers*. Blackwell Scientific, UK, 1989.

[Frö96] H. J. Fröhlich. Prototype of a run-time adaptable object-oriented system. In *PSI '96 (Perspectives of System Informatics)*, Akademgorodok, 1996. Springer-LNCS.

[Gam96] E. Gamma. Design patterns, frameworks, components — elements of moderns application architectures. Presentation at CUC'96, 1996.

[GHJV94] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Micro-Architectures for Reusable Object-Oriented Design*. Addison Wesley, Reading, MA, 1994.

[GJS96] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, September 1996.

[Gor88] M. J. Gordon. HOL: A proof generating system for higher-order logic. In G. Birtwistle et al., editor, *VLSI Specification, Verification and Synthesis*. Kluwer Academic Press, 1988.

[Gro94] The Object Management Group. *Common Object Service Specification*. John Wiley & Sons, 1994.

[GS92] D. Garlan and M. Shaw. An Introduction to Software Architecture. In V. Ambriola and G. Tortora, editors, *Advances in Software Engineering and Knowledge Engineering*, pages 1–40. World Scientific Publishing Co., 1992.

[GTW79] J. Goguen, J. Thatcher, and E. Wagner. *An initial algebra approach to the specification, correctness, and implementation of abstract data types*, pages 80–149. Prentice-Hall, 1979.

[HHK+96] C. Hofmann, E. Horn, W. Keller, K. Renzel, and M. Schmidt. The field of software architecture. Technical Report TUM-I9641, Technische Univerität München, 1996.

[HN96] D. Harel and A. Naamad. The Statemate Semantics of Statecharts. *IEEE Transactions on Software Engineering Method*, 1996.

[HO93] W. Harrison and H. Ossher. Subject-oriented programming (A critique of pure objects). In Andreas Paepcke, editor, *OOPSLA 1993 Conference Proceedings*, volume 28 of *ACM SIGPLAN Notices*, pages 411–428. ACM Press, October 1993.

[Hoa69] C. A. R. Hoare. An Axiomatic Basis of Computer Programming. *Communications of the ACM*, 12:576–580, 1969.

[Hoa72] C. A. R. Hoare. Proofs of correctness of data representations. *Acta Informatica*, 1:271–281, 1972.

[Jac96a] B. Jacobs. Inheritance and cofree constructions. In P. Cointe, editor, *Proceedings ECOOP '96*, LNCS 1098, pages 210–231, Linz, Austria, July 1996. Springer-Verlag.

[Jac96b] B. Jacobs. Objects and classes, coalgebraically. In B. Freitag, C. B. Jones, C. Lengauer, and H. J. Schek, editors, *Object-Orientation with Parallelism and Persistence*, pages 83–103. Kluwer Academic Publishers, Boston, 1996.

[JD93] M. P. Jones and L. Duponcheel. Composing monads. Technical Report RR-1004, Yale University, December 1993.

[JF88] R. E. Johnson and B. Foote. Designing Reusable Classes. *Journal of Object-Oriented Programming*, 1(2):22–35, June/July 1988.

[Jon91] M. P. Jones. Introduction to Gofer 2.20. Technical report, Yale University, September 1991.

[Jon95] M. P. Jones. A system of constructor classes: overloading and implicit higher-order polymorphism. *Journal of Functional Programming*, 5(1), January 1995.

[JW93] S. L. Peyton Jones and P. Wadler. Imperative functional programming. In *Proceedings of the Twentieth Annual ACM Symposium on Principles of Programming Languages, Charleston, South Carolina*, pages 71–84, January 1993.

[JZ91] R. E. Johnson and J. M. Zweig. Delegation in C++. *J. of Object-Oriented Programming*, 4(3), November 1991.

[KCH+90] K. Kang, S. Cohen, J. Hess, W. Novak, and S. Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-21, Carnegie Mellon Software Engineering Institute, November 1990.

[KLM+97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *ECOOP '97*. Springer LNCS 1241, 1997.

[KPR97] C. Klein, C. Prehofer, and B. Rumpe. Feature specification and refinement with state transition diagrams. In P. Dini, editor, *Fourth IEEE Workshop on Feature Interactions in Telecommunications Networks and Distributed Systems*. IOS-Press, 1997.

[KR94] S. N. Kamin and U. S. Reddy. Two semantic models of object-oriented languages. In Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design*, pages 464–495. The MIT Press, 1994.

[KV95] K. Kimbler and H. Velthuijsen. Feature interaction benchmark, October 1995. Discussion paper for the panel on benchmarking at FIW'95, Kyoto, Japan.

[LA94] J. Lamping and M. Abadi. Methods as assertions. In M. Tokoro and R. Pareschi, editors, *Proceedings ECOOP '94*, LNCS 821, pages 60–80, Bologna, Italy, July 1994. Springer-Verlag.

[Lew95] T. Lewis, editor. *Object-Oriented Application Frameworks*. Manning Publications, Greenwich, USA, 1995.

[LHJ95] S. Liang, P. Hudak, and M. Jones. Monad transformers and modular interpreters. In *22nd ACM Symposium on Principles of Programming Languages*, San Francisco, California, 1995.

[LLNW96] U. Lechner, C. Lengauer, F. Nickl, and M. Wirsing. (objects + concurrency) & reusability - A proposal to circumvent the inheritance anomaly. In P. Cointe, editor, *Proceedings ECOOP '96*, LNCS 1098, pages 232–247, Linz, Austria, July 1996. Springer-Verlag.

[LM91] J. A. Lawless and M. Molly. *Understanding CLOS: the Common LISP object system*. Digital Press, Nashua, NH, 1991.

[LW90] G. T. Leavens and G. E. Weihl. Reasoning about Object-Oriented Programs that Use Subtypes. In *Proceedings of the OOPSLA/ECOOP '90 Conference on Object-oriented Programming Systems, Languages and Applications*, pages 212–223, October 1990. Published as ACM SIGPLAN Notices, volume 25, number 10.

[LW93] B. Liskov and J. M. Wing. A new definition of the subtype relation. In Oscar M. Nierstrasz, editor, *ECOOP '93 — Object-Oriented Programming, 7th European Conference, Kaiserslautern, Germany*, volume 707 of

Lecture Notes in Computer Science, pages 118–141. Springer-Verlag, New York, N.Y., July 1993.

[Mey92] B. Meyer. *Eiffel: The Language*. Object-Oriented Series. Prentice Hall, New York, N.Y., 1992.

[Mez97] M. Mezini. Dynamic object modification without name collisions. In *ECOOP '97*. Springer LNCS 1241, 1997.

[MH69] J. McCarthy and P. J. Hayes. Some philosophical problems from the standpoint of artificial intelligence. *Machine Intelligence*, 4:463–502, 1969.

[MMPN93] O. Lehrmann Madsen, B. Moller-Pedersen, and K. Nygaard. *Object-Oriented Programming in the BETA Programming Language*. Addison-Wesley, Reading, 1993.

[Moe82] B. Moeller. *Unendliche Objekte und Geflechte*. PhD thesis, Muenchen, 1982.

[Mog91] E. Moggi. Notions of computation and monads. *Information and Computation*, 93(1), 1991.

[MPH97] P. Müller and A. Poetzsch-Heffter. Preserving the correctness of object-oriented programs under extension. In R. Berghammer and F. Simon, editors, *Programming Languages and Fundamentals of Programming*. Christian Albrechts-Universität Kiel, 1997. Technical Report.

[MS97] A. Mikhajliva and E. Sekerinsky. Class refinement and interface refinment in object-oriented programs. In *FME 97*. LNCS, 1997.

[MTH90] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, 1990.

[MY93] S. Matsuoka and A. Yonezawa. Analysis of inheritance anomaly in object-oriented concurrent programming languages. In P. Wegner G. Agha and A. Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, pages 107–150. MIT Press, 1993.

[NP95] T. Nipkow and C. Prehofer. Type reconstruction for type classes. *J. Functional Programming*, 5(2):201–224, 1995. Short version appeared in POPL '93.

[OJ90] W. F. Opdyke and R. J. Johnson. Refactoring: An Aid in Designing Application Frameworks. In *Proceedings of the Symposium on Object-Oriented Programming emphasizing Practical Applications*. ACM-SIGPLAN, September 1990.

[OW97] M. Odersky and P. Wadler. Pizza into Java: Translating theory into practice. In *Proc. 24th ACM Symposium on Principles of Programming Languages*, January 1997.

[PH97] A. Poetzsch-Heffter. Specification and verification of object-oriented programs, 1997. Habilitationsschrift.

[PHA+96] J. Peterson[editor], K. Hammond[editor], L. Augustsson, B. Boutel, W. Burton, J. Fasel, A. Gordon, J. Hughes, P. Hudak, T. Johnsson, M. Jones, S. Peyton Jones, A. Reid, and P. Wadler. Haskell 1.3, A non-strict, purely functional language. Report YALEU / DCS / RR-1106, Department of Computer Science, Yale University, May 1996.

[Pie91] B. C. Pierce. *Basic Category Theory for Computer Scientists.* MIT Press, Cambridge, Mass., 1991.

[Pre97a] C. Prehofer. Feature-oriented programming: A fresh look at objects. In *ECOOP '97.* Springer LNCS 1241, 1997.

[Pre97b] C. Prehofer. From inheritance to feature interaction. In Max Mühlhäuser et al., editor, *Special Issues in Object-Oriented Programming. ECOOP 1996 Workshop on Composability Issues in Object-Orientation*, Heidelberg, 1997. dpunkt-Verlag.

[Pre97c] C. Prehofer. From inheritance to feature interaction or composing monads. In *Arbeitstagung Programmiersprachen, Tagungsband der GI-Jahrestagung.* Springer-Verlag, 1997.

[Pre97d] C. Prehofer. An object-oriented approach to feature interaction. In P. Dini, editor, *Fourth IEEE Workshop on Feature Interactions in Telecommunications networks and distributed systems.* IOS-Press, 1997.

[PT94] B. C. Pierce and D. N. Turner. Simple type-theoretic foundations for object-oriented programming. *Journal of Functional Programming*, 4(2):207–247, April 1994. A preliminary version appeared in Principles of Programming Languages, 1993, and as University of Edinburgh technical report ECS-LFCS-92-225, under the title "Object-Oriented Programming Without Recursive Types".

[RAI92] The RAISE Language Group. *The RAISE Specification Language.* BCS Practitioner Series. Prentice-Hall International, 1992.

[Rya97] M. Ryan. Fireworks project homepage, case studies, 1997. `http://www.cs.bham.ac.uk/~mcp/fireworks/casestudies.html`.

[SCD+93] P. Steyaert, W. Codenie, T. D'Hondt, K. De Hondt, C. Lucas, and M. Van Limberghen. Nested Mixin-Methods in Agora. In O. Nierstrasz, editor, *Proceedings of the ECOOP '93 European Conference on Object-oriented Programming*, LNCS 707, Kaiserslautern, Germany, July 1993. Springer-Verlag.

[SG95] R. Stata and J. V. Guttag. Modular reasoning in the presence of subclassing. In *Object-Oriented Programming Systems, Languages, and Applications*, October 1995.

[SG96] M. Shaw and D. Garlan. *Software Architecture*. Prentice Hall, 1996.

[Sim96] C. Simonyi. Intentional programming - innovation in the legacy age. IFIP WG 2.1 meeting, 1996.

[SJE91] G. Saake, R. Jungclaus, and H.-D. Ehrich. Object-oriented specification and stepwise refinement. In J. de Meer, V. Heymer, and R. Roth, editors, *International IFIP Workshop on Open Distributed Processing*, volume 1 of *IFIP Transactions C: Communication Systems*, pages 99–121, Berlin, Germany, October 1991. North-Holland.

[Slo97] O. Slotosch. *Refinements in HOLCF: Implementation of Interactive Systems*. PhD thesis, Technische Univerität München, 1997.

[SPL96] L. M. Seiter, J. Palsberg, and K. J. Lieberherr. Evolution of object behavior using context relations. In David Garlan, editor, *Symposium on Foundations of Software Engineering*, San Francisco, 1996. ACM Press.

[Ste87] L. A. Stein. Delegation is inheritance. *ACM SIGPLAN Notices*, 22(12):138–146, December 1987.

[Stø96] K. Stølen. Refinement principles supporting the transition from asychronous to synchronous communications. *Science of Computer Programming*, 1996.

[Str91] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, 1991. 2nd edition.

[Tal97] Building object-oriented frameworks. Technical report, Taligent, 1997. Avaliable at `www.taligent.com`.

[Tee94] G. Teege. Hierastates: Flexible interaction with objects. Technical report, TU München, TUM-I9441, 1994.

[Tee96] G. Teege. Hierastates: Supporting workflows which include schematic and ad-hoc aspects. In Michael Wolf and Ulrich Reimer, editors, *Proc. of 1st Int. Conf. on Practical Aspects of Knowledge Management PAKM'96*, 1996.

[Tee97] G. Teege. Gestaltung von aktivitätenunterstützung in cscw-systemen durch endbenutzer, 1997. Habilitationsschrift.

[US91] D. Ungar and R. B. Smith. Self: The power of simplicity. *Lisp and symbolic computation*, 3(3), 1991.

[Van97] M. VanHilst. *Role Oriented Programming for Software Evolution.* PhD thesis, Univ. of Washington, 1997.

[Wad89] P. Wadler. Theorems for free! In *Proc. ACM Conf. Functional Programming and Computer Architecture*, pages 347–359, 1989.

[Wad92] P. Wadler. The essence of functional programming. *19th POPL*, pages 1–14, January 1992.

[Wad93] P. Wadler. Monads and functional programming. In M. Broy, editor, *Proceedings of the 1992 Marktoberdorf international summer school on program design calculi.* Springer Verlag, 1993.

[WG94] W. Wayt-Gibbs. Software's chronic crisis. *Scientific American*, 271(3), September 1994.

[Wir90] M. Wirsing. Algebraic Specification. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science.*, chapter 13, pages 675–788. North-Holland, Amsterdam, 1990.

[WP94] Y. Wang and D. L. Parnas. Simulating the behavior of software modules by trace rewriting. *IEEE Transactions on Software Engineering*, 20(10):750–759, October 1994.

[Zav93] P. Zave. Feature interactions and formal specifications in telecommunications. *IEEE Computer*, XXVI(8), August 1993.