Objects in Oz

Martin Henz

Dissertation

zur Erlangung des Grades
des Doktors der Naturwissenschaften
der Technischen Fakultät
der Universität des Saarlandes

Saarbrücken

Mai 1997

Zur Beurteilung der vorliegenden Dissertation wurden vom Promotionsausschuss der Technischen Fakultät unter der Leitung ihres Dekans Professor Dr. Alexander Koch als Berichterstatter die Professoren Dr. Gert Smolka und Dr. Seif Haridi bestellt.

Das Kolloquium zur Promotion fand am 23. Juni 1997 in Saarbrücken statt.

Zusammenfassung

Die Programmiersprache Oz verbindet die Paradigmen der imperativen, funktionalen und nebenläufigen Constraint-Programmierung in einem kohärenten Berechnungsmodell. Oz unterstützt zustandsbehaftete Programmierung, Programmierung höherer Ordnung mit lexikalischer Bindung und explizite Nebenläufigkeit, die mithilfe logischer Variablen synchronisiert werden kann.

In der Softwarepraxis hat sich mit der objekt-orientierten Programmierung ein weiteres Programmierparadigma etabliert. In der vorliegenden Arbeit beschäftige ich mich mit der Frage, wie objekt-orientierte Programmierung in geeigneter Weise in Oz unterstützt werden kann. Ich stelle ein einfaches und doch ausdrucksstarkes Objektsystem vor, belege seine Benutzbarkeit und umreiße seine effiziente Implementierung.

Ein zentraler Aspekt der Programmiersprache Oz ist ihre Unterstützung nebenläufiger Berechnung. Infolgedessen nimmt die Untersuchung des Einflusses der Nebenläufigkeit auf das Design des Objektsystems einen besonderen Rang ein. Ich untersuche die Möglichkeiten, die das Objektsystem bietet, um nebenläufige objekt-orientierte Programmiertechniken auszudrücken.

Ausführliche Zusammenfassung

Die Programmiersprache Oz verbindet die Paradigmen der imperativen, funktionalen und nebenläufigen Constraint-Programmierung in einem kohärenten Berechnungsmodell. Oz unterstützt zustandsbehaftete Programmierung, Programmierung höherer Ordnung mit lexikalischer Bindung und explizite Nebenläufigkeit, die mithilfe logischer Variablen synchronisiert werden kann.

In der Softwarepraxis hat sich mit der objekt-orientierten Programmierung ein weiteres Programmierparadigma etabliert. In der vorliegenden Arbeit beschäftige ich mich mit der Frage, wie objekt-orientierte Programmierung in geeigneter Weise in Oz unterstützt werden kann. Ich stelle ein einfaches und doch ausdrucksstarkes Objektsystem vor, belege seine Benutzbarkeit und umreiße seine effiziente Implementierung.

Ein zentraler Aspekt der Programmiersprache Oz ist ihre Unterstützung nebenläufiger Berechnung. Infolgedessen nimmt die Untersuchung des Einflusses der Nebenläufigkeit auf das Design des Objektsystems einen besonderen Rang ein. Ich untersuche die Möglichkeiten, die das Objektsystem bietet, um nebenläufige objekt-orientierte Programmiertechniken auszudrücken.

Die Dissertation bietet die erste ausführliche Behandlung objekt-orientierter Programmierung in einem Berechnungsmodell, das Zustand mit explizit nebenläufiger Constraint-Programmierung verbindet. Programmiersprache Oz eröffnet durch zustandsbehaftete Programmierung und Programmierung höherer Ordnung Möglichkeiten, die weit über die bisherigen Ansätze für Objekte in nebenläufigen Constraint-Sprachen hinausgehen. Programmiertechniken aus imperativer und funktionaler Programmierung können zur Integration objekt-orientierter Programmierung genutzt werden. Daher bestehen wesentliche Beiträge der Dissertation aus der Übertragung und Anpassung solcher Techniken in das Berechnungsmodell von Oz. Die Beiträge der Dissertation liegen in den Bereichen des Sprachdesigns, der nebenläufigen Programmierung und der Implementierung von Programmiersprachen.

Sprachdesign. Der zentrale Beitrag der Dissertation besteht in der Entwicklung eines einfachen und doch ausdrucksmächtigen Modells zur objekt-orientierten Programmierung in einer Constraint-Sprache höherer Ordnung mit expliziter Nebenläufigkeit. Durch zustandsbehaftete Programmierung eröffnet sich die Möglichkeit, konventionelle objekt-orientierte Programmierung in ein solches Programmiermodell zu integrieren. Objekt-orientierte Programmiertechniken aus zustandsbehafteter funktionaler Programmierung werden an die Kontroll- und Datenstrukturen von Oz angepaßt.

Die direkte Unterstützung von Namen in Oz bietet - zusammen mit der lexikalischen Bindung von Programmbezeichnern - die Möglichkeit, wichtige objekt-orientierte Konzepte wie private Attribute und Methoden direkt auszudrücken. Bisher wurden diese Konzepte in objekt-orientierten Sprachen durch ad-hoc Konstruktionen realiziert.

Nebenläufige Programmierung. Ich zeige, daß die Kombination von logischen Variablen mit Zustand mächtige Ausdrucksmittel zur nebenläufigen Programmierung bietet. Diese Mittel benutze ich, um hohe Programmierabstraktionen wie etwa "thread-reentrant locking" auszudrücken.

Allen bisher benutzten Modellen zur objekt-orientierten Programmierung in nebenläufigen Constraint-Sprachen liegt das Konzept des aktiven Objektes zugrunde. Ich stelle diesem das Konzept des passiven Objektes gegenüber und biete starke Evidenz für die Überlegenheit des letzteren als Basis für nebenläufige Objekte.

Praktisch keine konventionelle objekt-orientierte Sprache bietet Botschaften als emanzipierte Datenstrukturen. Ich zeige, daß emanzipierte Botschaften eine einfache Integration aktiver Objekte auf der Basis passiver Objekte erlaubt und daher eine wichtige Komponente für nebenläufige objekt-orientierte Programmierung darstellt.

Ich stelle ein Meta-Objekt-Protokoll für Oz vor, das eine flexible Experimentierplattform zur nebenläufigen objekt-orientierten Programmierung bietet.

Implementierung von Programmiersprachen. Ich gebe die erste detaillierte Beschreibung an, wie objekt-orientierte Programmierung in eine existierende Abstrakte Maschine einer nicht-objekt-orientierten Sprache effizient integriert werden kann. Ich zeige, daß die Performanz moderner objekt-orientierter Programmiersysteme durch einige chirurgische Eingriffe in eine solche Abstrakte Maschine erreicht werden kann.

Eine neue Technik wird vorgestellt, mit deren Hilfe emanzipierte Botschaften implementiert werden können, ohne daß ein Performanzverlust entsteht, wenn diese nicht benutzt werden. Diese Technik ist wesentlich für die Praktikalität der Darstellung aktiver Objekte auf der Basis passiver Objekte.

Abstract

The programming language Oz integrates the paradigms of imperative, functional and concurrent constraint programming in a computational framework of unprecedented breadth, featuring stateful programming through cells, lexically scoped higher-order programming, and explicit concurrency synchronized by logic variables.

Object-oriented programming is another paradigm that provides a set of concepts useful in software practice. In this thesis we address the question how object-oriented programming can be suitably supported in Oz. As a lexically scoped higher-order language, Oz can express a wide range of object-oriented concepts. We present a simple yet expressive object system, demonstrate its usability and outline an efficient implementation. A central aspect of Oz is its support for concurrent computation. We examine the impact of concurrency on the design of an object system and explore the use of objects in concurrent programming.

To Kelly

Preface

Research is a dynamic and interactive activity which thrives in an environment that fosters exchange of ideas and intense collaboration between researchers. I found such an environment in the Programming Systems Lab in Saarbrücken. It was the cooperation with the enthusiastic, knowledgeable and cooperative researchers of this lab that lead to the findings reported in this thesis. To say that without them this work would not have been possible would miss the point. This is *their work* as well as it is mine. Such a research environment ridicules the stereotypical notion of the lone searcher for truth who entrenches himself in his ivory tower and comes back with a thesis. I gladly present the results of a collaborative effort to the public.

I was fortunate to be supervised by Gert Smolka, whose work laid the base for this thesis, and whose tireless striving for simplicity as a chief goal of scientific endeavor was truly inspiring. Ralf Scheidhauer implemented the emulator support and contributed several ideas to the implementation of the object system. Christian Schulte contributed countless ideas to the design of Objects in Oz. Jörg Würtz helped lay the base for the initial design of Objects in Oz. Christian Schulte and Konstantin Popov were the first object-oriented programmers in Oz and shared their programming experience with me. Martin Müller and Michael Mehl contributed concurrent programming examples. Michael Mehl helped with comments and suggestions and shared his experience with object-oriented programming with me. Martin Müller and Joachim Niehren shared their knowledge concerning a few fundamental aspects that I discuss in passing. Denys Duchier contributed an elegant syntactic detail. I thank Seif Haridi for several fruitful discussions on the design of Objects in Oz. My office mates Jörg Würtz and Joachim Walser sent out good vibes and were fun to work with. Leif Kornstaedt, Kelly Reedy, Ralf Scheidhauer, Christian Schulte, Gert Smolka and Joachim Walser commented on earlier versions of this thesis. Of course any mistakes that are still in it have been added by me *after* they looked at it and thus are entirely due to my ignorance. The following people provided advice and assistance for the performance measurements in Section 8.6: Hubert Baumeister, Seif Haridi, Michael Mehl, Tobias Müller, Jérôme Vouillon, Peter Van Roy, and Joachim Walser.

This thesis is a self-contained monograph on object-oriented programming in the programming language Oz. Previous papers on this topic [SHW93, HSW93, HS94, SHW95] document various intermediate stages of development. Their technical content underwent heavy revision and thus they are now of interest mostly as precursors of the present work.

Central reported techniques rely on several features of the underlying programming language that are not unique by themselves, but need to be *combined* in a single language. These features include higher-order programming, stateful programming, thread-level concurrency and synchronization with logic variables. Obviously it would have been hard to make all these buzzwords fit in the title of this dissertation. On the other hand, Oz is currently the only language that combines these features. So at the moment the title "Objects in Oz" is quite fitting. I do hope, however, that this work can help motivate the integration of these features in other languages. It would not be the worst fate of this dissertation if it could contribute to making its own title obsolete.

This thesis reports on work I carried out at the Programming Systems Lab in Saarbrücken from January 1992 to May 1997. From January 1992 to March 1996, I was employed by DFKI (German Research Center for Artificial Intelligence) and funded by the Bundesminister für Bildung, Wissenschaft, Forschung und Technologie (Hydra, ITW 9105). From April 1996 to July 1997, I was employed by the Sonderforschungsbereich 318, Ressourcenadaptive kognitive Prozesse (Special Research Division Resource-Adaptive Cognitive Processes) of the Universität des Saarlandes, Saarbrücken, Germany.

Martin Henz
Saarbrücken, Germany
May 1997

Contents

List of Figures

List of Programs

List of Tables

"Then you must go to the City of Emeralds. Perhaps Oz will help you."
"Where is this city?" asked Dorothy.
"It is exactly in the center of the country, and is ruled by Oz, the Great Wizard I told you of."
"Is he a good man?" inquired the girl anxiously.
"He is a good Wizard. Whether he is a man or not I cannot tell, for I have never seen him."
"How can I get there?" asked Dorothy.
"You must walk. It is a long journey, through a country that is sometimes pleasant and sometimes dark and terrible. However, I will use all the magic arts I know of to keep you from harm."

Chapter: The Council with the Munchkins

Chapter 1

Introduction

1.1 Area of Research

Software construction is an inherently complex task. Sources for this complexity are the complexity of the application domains and of the software development process on one hand, and the flexibility that programming provides on the other hand. Complex software can only be constructed successfully if the process is guided by powerful abstractions in all phases of the software development process. Our focus is on programming rather than analysis, design or maintenance, and thus our central aim is to provide powerful *programming* abstractions. High-level programming languages have been and are being developed that aim at providing such abstractions. Object-oriented programming languages aim at mastering complexity by centering computation around data items and operations on them. For many applications, object-oriented programming languages provide an attractive set of programming abstractions.

Many applications are naturally composed of autonomous entities that progress concurrently towards performing an overall task. Instead of leaving the programmer with the tedious task of splitting up sequential control to serve these autonomous entities, modern programming languages provide the programmer with high-level concurrent abstractions that allow to spawn and synchronize concurrent computation.

The programming language Oz supports concurrent computation in a programming framework of unprecedented breadth. Oz integrates (dynamically typed) functional programming, central aspects of logic and concurrent constraint programming with thread-level concurrency. The questions that we are addressing in this thesis are how object-oriented programming can be supported in Oz, how object-oriented programming can be integrated in an implementation of Oz, and how concurrent programming interacts with objects.

1.2 Programming Language Design

> Any programming language should be simple, expressive, and efficient... *Sverker Janson in [Jan94]*

Programming languages are tools used by programmers to solve problems on a computer. The fact that they are used by humans implies that they should be *simple*; as with any good tool, the programmer should be able to concentrate on the problem rather than be distracted by a complicated tool. The language should be *expressive* enough to offer a range of programming abstractions to support the modeling of the application area at hand. Simplicity and expressivity are often in conflict with each other. A language designer striving for expressivity may add more and more features, thus sacrificing simplicity, and a designer striving for simplicity may deliberately leave out features that would come in handy to express problem solutions.

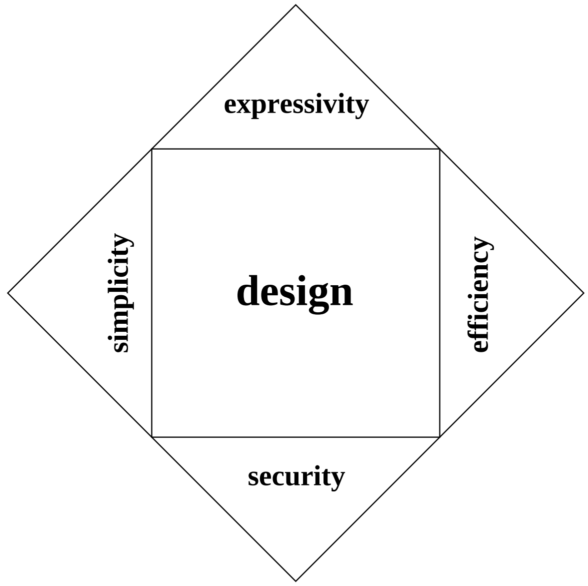
A third dimension that a language designer needs to keep in mind is *efficient* execution of programs on the target computers. There are potential conflicts of efficiency with both simplicity and expressivity. For example, assembly languages allow to write efficient programs but fail to provide expressive programming abstractions, and languages with automatic memory management provide simplicity potentially at the expense of efficiency-compromising garbage collections.

As systems become more and more complex, a fourth dimension comes into play. Large programs can only be manageable if the impact of local changes can be limited. The language designer is faced with the question to what extent *security* can be guaranteed without negative impact on the other design issues. For example, safe static type systems enforce a security from runtime errors potentially at the expense of expressivity in a sense that meaningful programs might be rejected. Dynamic type checking supports security by allowing to localize runtime errors at the expense of efficiency.

To summarize, we depict the situation of the language designer in Figure 1.1 as a extension of Janson's triangle [Jan94]. During this presentation, we will frequently be forced to take our stand in this area of conflict.

The language design process is often presented as a cycle. Starting with an *application* domain, a *language* is designed in which the problems in the domain can be solved. The language is implemented yielding a *system* with which the applications can be solved. Experience with these applications leads to refining the language design and so on. In practice, however, the temporal and causal dependencies in this process are complex and elusive. Feedback goes from the

Figure 1.1 Language Design Principles



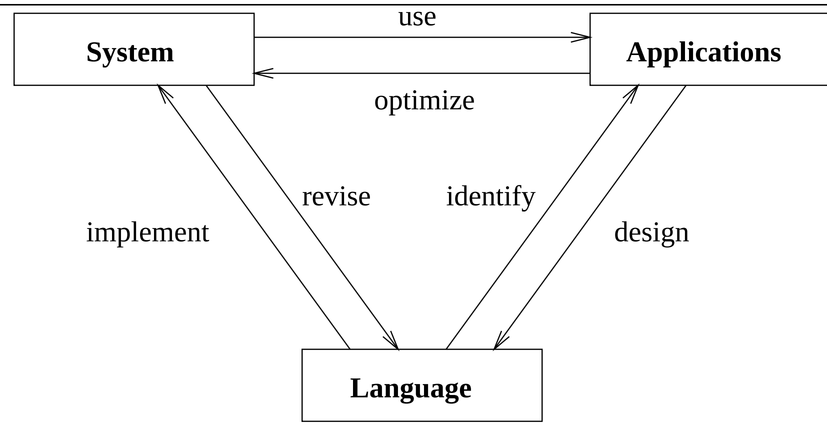
implementation to the language (e.g. revisions of constructs that cannot be implemented efficiently), from the language to the envisioned applications (e.g. new application domains become tractable or envisioned applications are recognized as intractable), and from the applications to the implementation (e.g. critical efficiency issues are detected that need new implementation techniques). Figure 1.2 depicts the language design cycle.

1.3 Object-oriented Programming

Object-oriented programming is a method of programming in which several programming concepts are combined in a particular manner. Not all of these concepts of programming have been developed in the context of object-oriented programming; rather it is their particular *integration* in a coherent programming method that is specific to object-oriented programming. This programming method prescribes organizing software into classes each of which describe the behavior of encapsulated data structures, called their *instances* or just *objects*. Classes define the set of operations called *methods* that can be *invoked* on their instances.

Often, a class contains many methods and many classes share common behavior. To structure the functionality of classes and to avoid duplication of methods, object-oriented programming allows for *inheritance* of classes such that a class can be defined as an incremental modification of one other class (single inheritance) or several other classes (multiple inheritance). These other classes are

Figure 1.2 Language Design Cycle



called *superclasses* of the newly defined *subclass*.

Many programming languages allow to program in accordance with the object-oriented programming method. What distinguishes *object-oriented programming languages* from other programming languages is that they provide semantic and syntactic facilities that make it particularly convenient to use this method, or even *force* the programmer to do so.

1.4 Concurrency

Sequential execution lies at the heart of conventional programming languages. The programmer's instructions are executed in strict sequential order. However, many applications such as interactive systems can be decomposed into autonomous entities that do not lend themselves naturally to an overall sequential order. Sequential programs for such applications typically use complex and artificial control structures to schedule and interleave activities (e.g. event loops in graphics libraries). Instead, the programmer should think of such computational activity as being naturally concurrent. To this goal modern programming languages provide for spawning new sequential threads of control that can be executed concurrently.

Concurrency adds another dimension to the complexity of software; a new range of concerns needs to be considered. For example, an I/O device may be constructed such that it must carry out a sequence of actions in a specified order to function properly. Several user processes operating on the same device concurrently will cause havoc. We call this synchronization requirement *mutual exclusion*. Many more such synchronization requirements have been identified in concurrent programming practice.

Striving for generality, different approaches to synchronization problems are compared and *abstractions* are developed that provide generic solutions. A particularly attractive concept is to provide for synchronization upon availability of data. This idea was underlying the data flow languages [Den74] and the concept of futures [BH77, Hal85] in functional programming. A more expressive and flexible idiom for data-driven synchronization is provided by the logic variable that will play a central role in our work.

1.5 Objects and Concurrency

A significant part of software development today uses object-oriented analysis, design and programming methods. Concurrency is crucial in many of these applications. Thus the question how object-oriented programming and concurrency interact is an important one.

It is often claimed that the success of object-oriented programming lies in the fact that concepts familiar in modeling physical systems can be fruitfully put to work in developing software. The analogy of software objects to physical objects is often helpful to the programmer. If we carry this analogy further, it is tempting to attribute certain patterns of concurrent behavior to software objects. After all, physical objects also exist in a concurrent environment.[1] Indeed, it is often useful to equip objects with patterns of concurrent behavior. One such a pattern is the *active object*, which is an object with associated computational resources that are used for operations on the object. Another pattern is the *concurrent passive object*, which is an object without computational resources that guarantees that concurrently issued operations on it respect synchronization requirements such as mutual exclusion.

What is questionable however is to *enforce* a certain concurrency pattern on an object-oriented language. It is argued by Stroustrup [WL96] that an integration of concurrency in an object model necessarily favors one particular model for concurrent objects at the expense of other reasonable models. Therefore concurrency and objects should be kept separate in language design. In the course of the development of Objects in Oz, this argument became painfully clear. A concurrency model for objects was fixed at an early stage and carried along through various changes of the underlying language. Relatively late, this model was found inappropriate and the validity of Stroustrup's argument became apparent.

Instead of fixing one particular concurrent object model, the goal should be to provide easy-to-use abstractions for a wide variety of concurrent behavior of objects, ranging from purely sequential objects to active objects. Indeed, we shall

[1] As Kahn [Kah96] points out "programs typically model the world and the world is concurrent. Sequential programming languages provide a world in which only one thing can happen at a time. This is a very strange world."

argue that the programming language Oz provides a unique set of techniques to define such abstractions.

1.6 Concurrent Logic Programming

Logic programming developed as a procedural interpretation of Horn clause logic [Kow74] and as a specialized computation framework for natural language processing [CKPR73] around 1972. The first and most widely used logic programming language is Prolog, which uses SLD-resolution over constructor terms as its main computational principle. From the programmer's point of view, the logic variable plays a central role in Prolog. Logic variables can refer to values of which only partial or no information is known. Constraint logic programming [JM94, BC93] is a generalization of logic programming in which resolution is used as operational core for reasoning not only over constructor terms but over domains as diverse as infinite trees and intervals of real numbers. Constraint logic programming over finite domains developed efficient techniques for solving a variety of combinatorial problems.

Concurrent logic programming [Sha89] originated with the Relational Language [CG81] and is based on the observation that the logic variable can be used to synchronize concurrent computation. For this purpose, SLD-resolution was replaced by a new computation model based on the notion of committed choice.

Soon it was realized by Shapiro and Takeuchi [ST83] that concurrent logic languages can express active objects using message streams. Shapiro and Takeuchi also point out the power and elegance of the logic variable for synchronization. Syntactic considerations led to the development of a number of object-oriented extensions to concurrent logic languages, such as Vulcan [KTMB87], A'UM [YC88] and Polka [Dav89]. In these languages, many-to-one communication is realized with stream merging, which causes inefficiency and is conceptually problematic. Janson, Montelius and Haridi [JMH93] introduced a dedicated language primitive called *port* for efficient many-to-one communication in concurrent logic languages.

1.7 The Language Oz

Concurrent constraint programming [Mah87, Sar93] (ccp) generalized concurrent logic programming and constraint logic programming in a uniform computational framework. ccp introduces elegant notions of communication and synchronization based on constraints. A computational agent may add information to the store (tell) and wait for the arrival of a specified information in the store (ask). Janson presents the ccp language AKL [Jan94], which demonstrates that a practical programming system was possible in which concurrency and problem solving can fruitfully coexist.

Like AKL, the development of Oz [Smo95, ST97] was driven by the goal to provide a multi-paradigm programming framework. Already in Smolka's initial vision [Smo91] that led to the development of Oz, object-oriented programming was considered a "major objective". Rather than viewing object-oriented programming as a convenient extension of the language, it was treated as a central design issue. A rigorous pursuit of this goal led to crucial design decisions. The first step beyond AKL was to marry concurrent constraint programming with lexically scoped first class procedures, notions that are inherent in lambda calculus and were introduced to programming languages through Algol [N^+63] and Lisp.[2] This step opened the stage for new research directions as diverse as innovative problem solving engines and novel approaches to transparently distributed computation. For object-oriented programming, this had the consequence that a plethora of techniques from functional programming became available that rely on lexically scoped higher-order programming. Higher-order programming cut a restraining rope attaching Oz to its heritage from concurrent logic programming.

Being initially conceived without search, a novel programming abstraction [SS94] has been integrated to allow for a wide variety of search engines. As constraint domains, record constraints [ST94, RMS96] generalizing constructor trees, and finite domain constraints [SSW94] are provided.

Motivated by object-oriented programming, *cells* were integrated. From the perspective of functional and constraint programming, cells introduced the basic ingredient of imperative programming to the language. In summary, Oz can be seen as a unified programming framework subsuming imperative programming, lexically scoped higher-order functional programming and central aspects of concurrent constraint programming. Based on an abstract machine for Oz [MSS95], DFKI Oz [ST97] provides for a robust and efficient implementation.

1.8 The Development of Objects in Oz

Initial experiments with active objects led to the conclusion that while being a useful programming idiom they cannot serve as the base for a practical object system. A communication primitive called *constraint communication*—inspired by Milner's π calculus [Mil93]—was introduced to implement passive objects [HSW93].

[2]While Lisp's first-class procedures predate Algol, lexical scoping found its way into (then Common) Lisp from Algol via Scheme [SS75] as late as 1980. Steele and Gabriel [SG93] give an excellent account of the evolution of the Lisp family of programming languages.

Later, it was realized that while equally expressive, the notion of a cell provides a much simpler foundation for passive objects in Oz [SHW95].

Complementing the exploration of object-oriented concepts in Oz, a class-based object-oriented syntax extension was developed and implemented. A fortunate early development in Oz was the integration of records in the computation model [ST94], since records are the congenial data structure for various elements of an object model such as state, classes, and messages.

Initially, Oz was conceived as a language with fine-grained concurrency much in the tradition of concurrent logic programming. Most language constructs had the ability of spawning concurrent computation. Experience with larger programs such as the Oz Browser [Pop95] and the Oz Explorer [Sch97] suggested that this was not desirable as it made concurrency difficult to control. Thus, the more conventional *explicit concurrency* was adopted. The programmer can spawn new threads of computation, and unless he does so, a program runs sequentially. However, in contrast to other concurrent thread-based languages, synchronization is governed by data flow through logic variables.

The object system followed this development. For implicit concurrency, it was regarded as necessary to provide a standard synchronization mechanism for passive objects at the expense of fixing a particular concurrency model for objects. As central synchronization mechanism, a monitor semantics was chosen. Concurrent applications of objects was synchronized such that mutual exclusion of code that operated on the object's state was guaranteed [SHW95]. Negative consequences became apparent. General enforcement of mutual exclusion was perceived as overly restrictive, imposing the understanding of a complex semantics on the programmer and even leading to subtle programming errors in "sequential" programs.

The shift to explicit concurrency allowed to drop the monitor semantics for objects, because the programmer is in control of the created concurrency. In our experience, a coarse-grained concurrent structure is appropriate for many applications; large parts of the application can run entirely sequentially and communicate and synchronize each other using small concurrent interfaces. Given a clear design of these interfaces, the synchronization issues can be identified and solved. This situation allows to decouple Objects in Oz from concurrency issues, such that they are optimally suited for sequential programming. For programming of concurrent interfaces, we provide suitable object-oriented abstractions for communication and synchronization.

Without cells, the synchronization mechanisms provided by Oz are not sufficiently expressive to deal with more than the simplest synchronization tasks in concurrent programming. However, the cell complements the logic variable ideally in this respect, leading to elegant solutions to a wide variety of synchronization abstractions. This observation is used to provide high-level synchronization idioms with and for Objects in Oz.

Oz and Small Oz

The aim of this thesis is to reveal the concepts underlying Objects in Oz. To focus on this task, we simplify Oz in aspects that are not relevant to object-oriented programming, resulting in a language called *Small Oz*. Oz and Small Oz are so close that every Small Oz program that occurs in this thesis is a working Oz program (tested on DFKI Oz 2.0 and available at [Hen97a]). Whenever the difference between Oz and Small Oz is irrelevant, we simply talk about Oz instead of Small Oz. However, we shall explain where and why Small Oz differs from Oz. By Oz, we refer to the language Oz 2 as defined in [ST97]. The precursor of Oz 2 is called Oz 1 and is briefly discussed in Chapter 11.

1.9 Contributions

This thesis represents the first in-depth treatment of object-oriented programming in thread-based concurrent constraint programming with state. Although there has been considerable work in object-oriented concurrent logic programming, the programming language Oz opens up concurrent constraint programming to a wide range of techniques well known in imperative and functional programming. Consequently, several central contributions of this work consist of extending and adapting these techniques to thread-based concurrent constraint programming. Contributions have been made in the areas of programming language design, concurrent programming and programming language implementation.

Language Design

Conventional Object-Oriented Programming for Concurrent Constraint Programming. The central contribution of this thesis lies in the development of a simple yet expressive model for object-oriented programming in the thread-based higher-order concurrent constraint language Oz. For the first time, conventional object-oriented programming becomes available in the framework of concurrent constraint programming. To this aim, object-oriented programming techniques in particular from stateful functional programming are adapted to the available data and control structure and syntax of Oz.

Names for Privacy. We contribute the discovery that names together with lexical scoping can express important object-oriented concepts such as private attributes and methods and protected methods. These techniques have previously been realized in object-oriented languages by ad-hoc constructions.

Concurrent Programming

Synchronization with Logic Variables and State. We contribute the realization that logic variables together with an appropriate notion of mutable state provide a wide variety of synchronization techniques. We use the resulting constructs to define high-level concurrent object-oriented programming abstractions such as thread-reentrant locking.

Active vs Passive Objects for Concurrent Constraint Programming. Active objects have provided the underlying model for object-oriented programming in all previous concurrent (constraint) logic languages. While being a useful programming concept, we provide strong evidence that active objects are inferior to passive objects as the underlying object-oriented concept.

First-class Messages for Active Objects. Virtually no conventional object-oriented language provides first-class messages. We show that first-class messages allow a simple integration of active objects in conventional object-oriented programming with thread-level concurrency. Comparing with efforts of other object-oriented languages for supporting active objects, we conclude that first-class messages are the congenial programming concept for active objects on the base of passive objects.

A Concurrent Meta-Object Protocol. We describe a meta-object protocol for Oz that can serve as a flexible platform for experimentation with alternative and additional synchronization mechanisms for objects in Oz.

Implementation Technology

Integrating Objects in an Abstract Machine. We give the first detailed account of how object-oriented programming concepts can be efficiently supported by an abstract machine implementing a non-object-oriented language. We show that the performance of state-of-the-art object-oriented programming systems can be attained with few surgical modifications of such an abstract machine.

Implementing First-Class Messages. A novel technique is presented that allows to implement an object system based on first-class messages without runtime or memory overhead if first-class messages are not used. This technique is crucial for the practicality of realizing active objects on the base of passive objects.

1.10 Outline

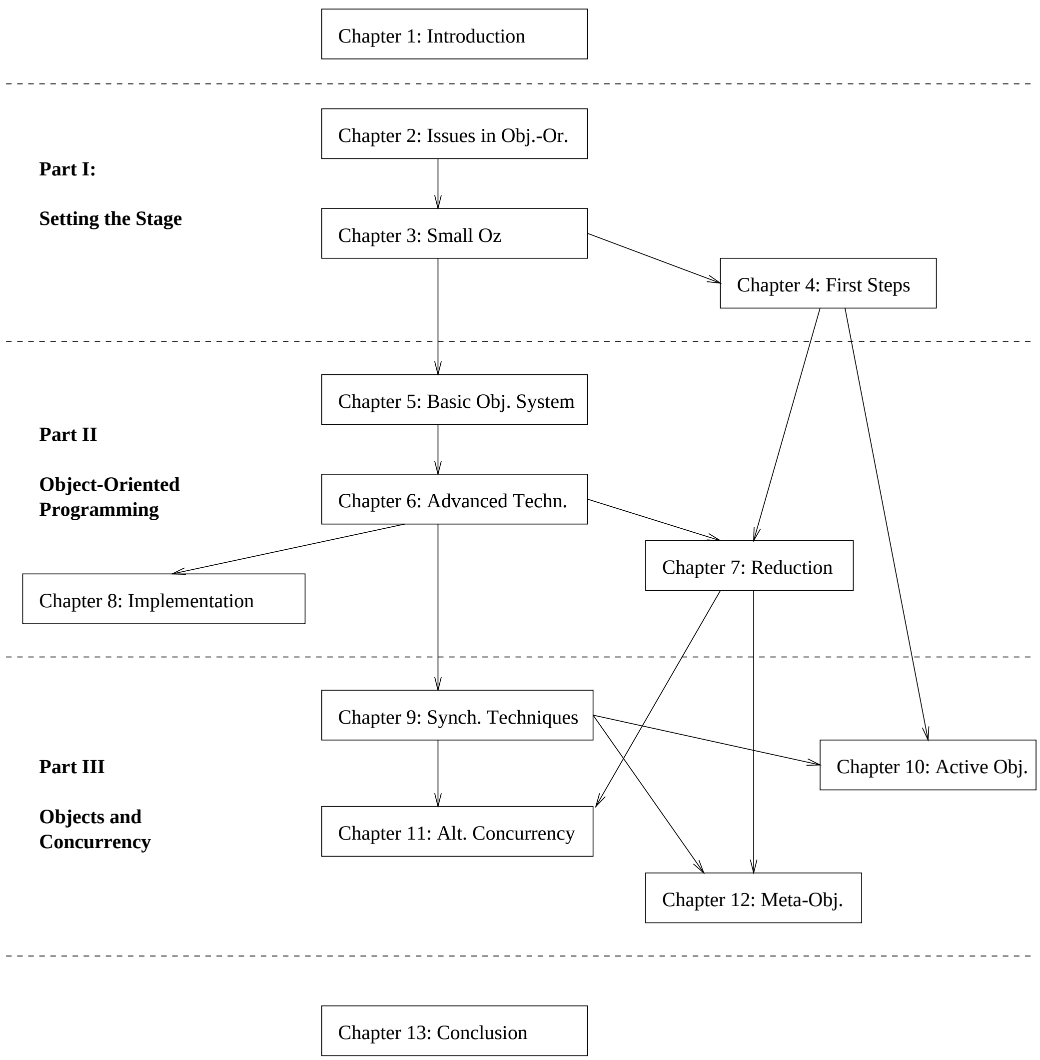
This thesis is organized in three parts, each consisting of three or four chapters. Figure 1.3 depicts the dependencies of the chapters and may be helpful for the reader to navigate through the thesis.

Part I sets the stage by introducing object-oriented programming and the language Small Oz. Chapter 2 provides an overview of issues in object-oriented and concurrent programming language design. We put object-oriented programming in a broader context of knowledge representation and software development and explain basic object-oriented abstractions such as like late binding and inheritance. We emphasize the role of logic variables in concurrent programming. Chapter 3 presents a duly simplified version of Oz that serves as the computational framework for this thesis. Chapter 4 takes first steps towards objects in Oz by casting two classical approaches to object-oriented programming in the framework of Oz.

Part II describes sequential object-oriented programming in Oz. In Chapter 5 we introduce the basic features of the Oz Object System and discuss the design decision taken. In Chapter 6 we present advanced features of the Oz Object System such as multiple inheritance, private identifiers, and first-classing. We present a semantic foundation for this object system in the form of a reduction to Small Oz in Chapter 7. In Chapter 8, we outline and evaluate a realistic implementation of the object system.

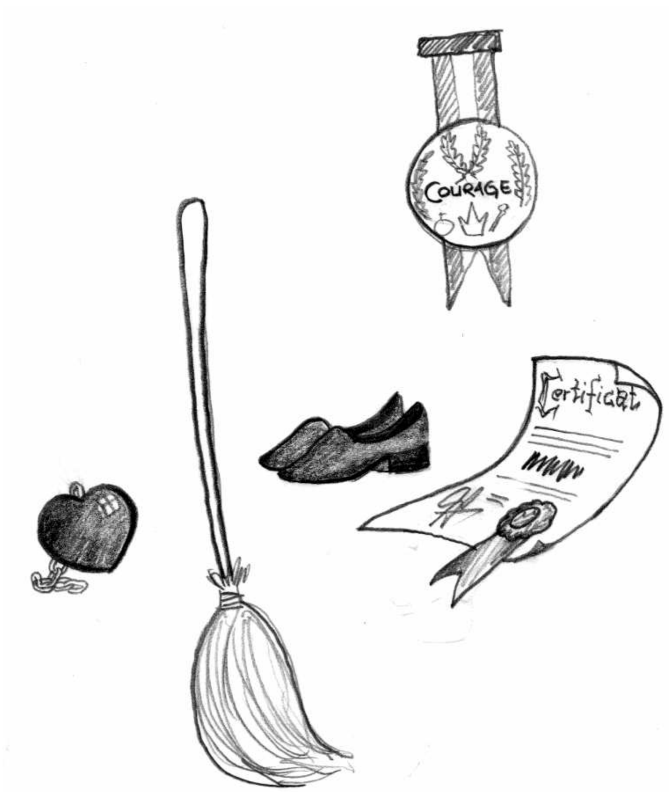
Part III treats concurrency issues in the framework of objects in Oz. In Chapter 9, we employ logic variables and cells to solve a variety of synchronization problems in concurrent programming. We show how reentrant locks, a common synchronization construct for concurrent objects, are integrated in the object system for Small Oz. In Chapter 10, we define programming abstractions for active objects. Chapter 11 discusses the design issues for objects in alternative concurrency models. In Chapter 12, we give a meta-object protocol that allows to use object-oriented programming to experiment with the concurrent aspects of objects.

Figure 1.3 Chapter Dependencies



Part I

Setting the Stage



This part sets the stage for Objects in Oz. Chapter 2 introduces the main concepts of object-oriented programming. Issues in the design of object-oriented languages are addressed and major object-oriented languages are compared. Chapter 3 introduces a sub-language of Oz called Small Oz that will be used throughout the thesis. We show in Chapter 4 that Small Oz can readily express established models of objects in functional and concurrent logic programming.

"Who are you?" asked the Scarecrow when he had stretched himself and yawned. "And where are you going?"

"My name is Dorothy," said the girl, "and I am going to the Emerald City, to ask the Great Oz to send me back to Kansas."

"Where is the Emerald City?" he inquired. "And who is Oz?"

"Why, don't you know?" she returned, in surprise.

"No, indeed. I don't know anything. You see, I am stuffed, so I have no brains at all," he answered sadly.

"Oh," said Dorothy, "I'm awfully sorry for you."

"Do you think," he asked, "if I go to the Emerald City with you, that Oz would give me some brains?"

Chapter: How Dorothy Saved the Scarecrow

Chapter 2

Issues in Object-Oriented Language Design

> My guess is that object-oriented programming will be in the 1980s what structured programming was in the 1970s. Everyone will be in favor of it. Every manufacturer will promote his products as supporting it. Every manager will pay lip service to it. Every programmer will practice it (differently). And no one will know just what it is.
>
> *Tim Rentsch in [Ren82]*

In this chapter we explore essential aspects of object-oriented programming languages in a top-down approach. Section 2.1 gives a view of object-oriented programming from the perspective of knowledge representation and argues that object-oriented programming supports a number of important abstraction principles. Section 2.2 covers aspects of software development. Here the notions of late binding, inheritance and encapsulation play a central role. Section 2.3 introduces central issues of concurrent programming as relevant to object-oriented programming.

2.1 Knowledge Representation View

Software is used to solve problems in given domains. To this aim, software expresses properties of entities, their relationship and interaction in a language accessible to automatic treatment such as compilation to processor instructions. The complexity of a given domain of application must be matched by the expressivity of the language in use. To understand a complex problem it is necessary to view it from different angles provided by *abstraction principles*. An often quoted hallmark of object-oriented programming is its support for the knowledge representation principles of *classification*, *aggregation* and *specialization* [Tai96].

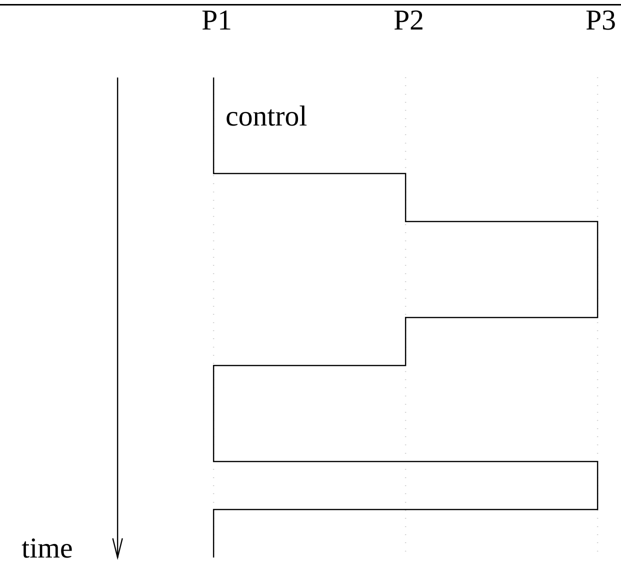
Classification This principle aims at grouping things together into classes such that common properties of the members can be identified. For example, it is useful to classify all individual participants of a road traffic scenario as *vehicles* that have properties like size, speed and direction of movement. Collectively, the *instances* of a *class* form the *extension* of that class. Object-oriented languages provide support for classification by allowing to define classes that describe the properties of their instances.

Aggregation This principle allows to form new concepts as collections of other concepts. For example, vehicles such as semitrucks are entities composed of parts such as cabin and trailer, each of these again being composed of wheels, axles, etc. In programming, aggregation is achieved by compound data structures that are called *objects* in the framework of object-oriented programming. The components are called *attributes* and can be referred to by *attribute identifiers*. During the lifetime of an object, attributes may change but usually the object structure, i.e. the names through which attributes are accessible, is fixed.

Specialization This principle allows to describe a concept as a more specific version of another concept. A concept C_s can be regarded as a specialization of another concept C if the extension of C_s is a subset of the extension of C. This relationship is often called *is-a* in the context of object-oriented programming [Ped89]. For example, concepts such as “car” and “truck” can be seen as specializations of the concept “vehicle”. In object-oriented programming, specialization can be achieved by defining classes as specialized versions of other classes using *inheritance*. A class C_s that inherits from another class C is called its subclass and C is called superclass of C_s. However, inheritance as provided by most object-oriented programming languages is more general than specialization. In particular, properties of a superclass can be overridden by a subclass. We shall see in Section 2.2.5 that the identification of inheritance with specialization is the source of much confusion in object-oriented programming.

When it comes to the design of a particular formalism such as an object-oriented programming language, interactions between these abstractions emerge. For example, the principle of aggregation suggests that classes describe the attributes of their instances and that these attributes are inherited.

Figure 2.1 Control Flow in Procedural Languages (time centered)



2.2 Software Development View

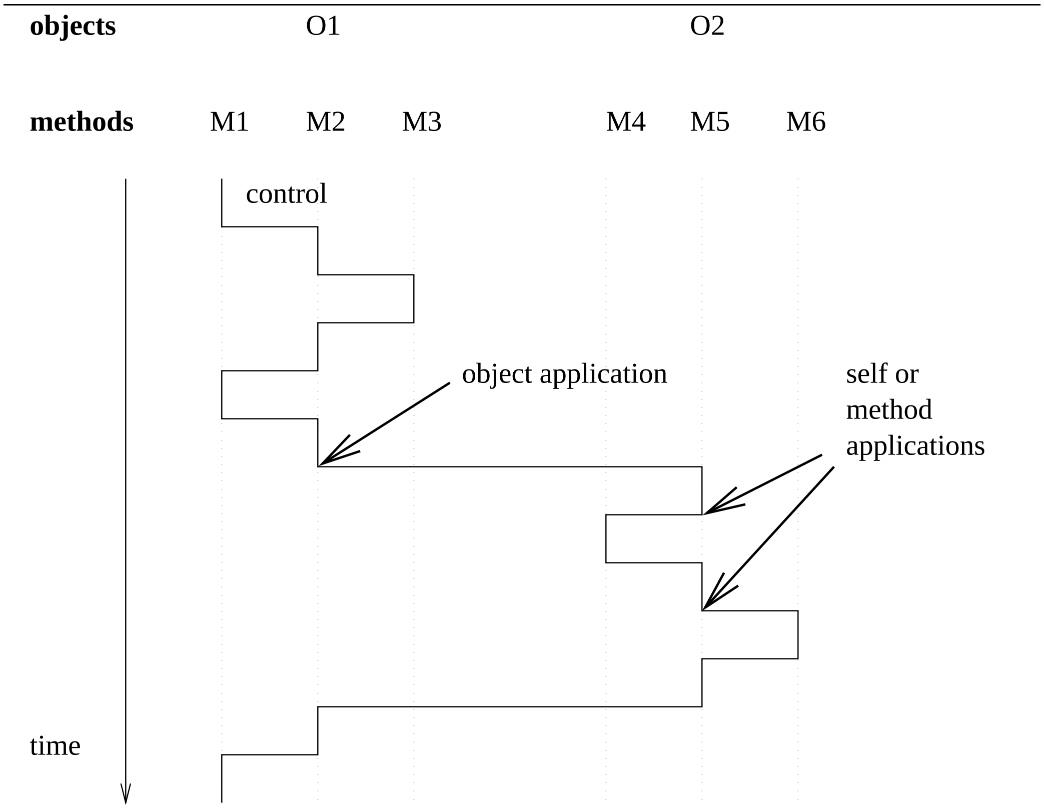
Over the years, a host of object-oriented analysis and design methods have been proposed (for overviews of this field see [Boo94, C^{+}93]). Usually issues like classification, aggregation and specialization play a central role in object-oriented analysis. The objects involved in a computation are identified and their properties are described. These issues can be characterized as static. Dynamic aspects come into play when the functionality of these objects is designed.

2.2.1 Focusing on Objects

In conventional programming languages, procedures are a central means to structure functionality. At runtime, control can be passed from one procedure to another by procedure application. The resulting control flow in conventional languages is depicted in Figure 2.1. Many languages allow to structure procedures according to their functionality, leading to the concept of modules.

A central idea behind object-oriented programming is that such structuring can be effectively guided by the *data* involved in the computation. At any point in time there is one dedicated data object *on* which the respective procedure is carried out. This *current* object is referred to as *self*. In object-oriented programming, procedures are called *methods*. The invocation of a method can either change self to another object or leave it the same. In the former case, we say that the object is *applied* to a message (object application); often this is called *message send-*

Figure 2.2 Control Flow in Object-Oriented Languages (time centered)



ing.[1] The latter case can be achieved either by a special case of object application, called *self application*, or by *method application*; we shall discuss these two possibilities later in this section. In all cases, the operation leads to execution of a corresponding method. The resulting refinement of the procedural control flow is depicted in Figure 2.2. It is convenient to group the methods that operate on a certain kind of object into *classes*. Classes are modules containing methods that all operate on the same kind of objects, which are called its *instances*.

Each instance of a class carries its own identity distinguishing it from all other objects. We call this approach to equality *token equality* as opposed to structural equality which defines two objects to be equal if they have the same structure and all their components are equal.

[1] The term "object application" used throughout the thesis is adopted from functional programming where objects are often represented as procedures [AS96] (procedural data structures). We refrain from using the term "message sending" since message sending has a conflicting connotation in concurrent programming (asynchronous communication).

2.2.2 Polymorphism and Object Application

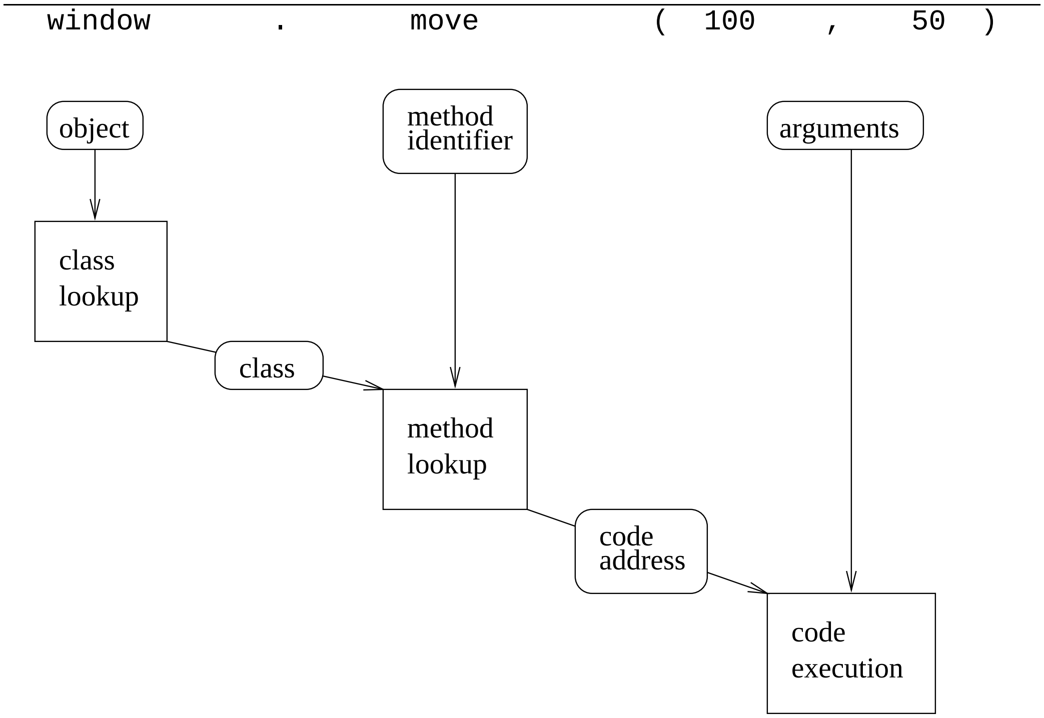
It is useful to group the values that can be referred to by identifiers into *types*. With respect to a given type structure, an identifier occurrence is *polymorphic*, if it can refer to values of different type at different points in time. Cardelli and Wegner [CW85] distinguish between two ways an operation can be applied to a polymorphic identifier occurrence. Either the operation can be performed uniformly on all values of the different types (e.g. you can compute the length of a list regardless of the type of its elements), or different operations will be performed for different types (an addition $x+y$ works differently if x and y are integers or floats). Operations of the former kind are called *universally polymorphic* and the latter *ad-hoc polymorphic*. Both kinds of polymorphism play a central role in object-oriented programming.

Object-oriented languages handle both kinds of polymorphism by using *late binding* for object application. Late binding introduces an indirection between the object application and execution of the corresponding method. An object is applied to a message which consists of a *method identifier* and further arguments. The method identifier and the class of the object being applied determine the method to be executed. Thus the class provides a mapping from method identifiers to methods. The other arguments are simply passed as arguments to the method. Figure 2.3 depicts the execution of an object application (using Java notation).

The object-oriented extension of Lisp, CLOS [Ste90], generalizes this scheme and allows all arguments to be considered for determining the method, which is therefore called *multimethod*. Late binding supports universal and ad-hoc polymorphism, since application of objects of different classes can lead to execution of the same or different methods as we shall see.

Polymorphism can be the source of programming errors, because the programmer may not be fully aware of the argument types of identifiers. Statically typed programming languages limit the polymorphism before the program gets executed by refusing programs that violate certain typing rules at compile time. Statically typed object-oriented languages usually introduce a type for every class. Polymorphism is restricted along the inheritance relation. Often the programmer can rely on the following invariant. If an identifier may hold instances of a class C, it may also hold instances of a class that inherits from C. This invariant is important in practice and is a central issue in defining type systems for object-oriented languages [PS94].

Figure 2.3 Execution of Object Application



2.2.3 Code Reuse

Often during development and maintenance of software, new functionality needs to be provided in addition to supporting old functionality. For example, a window system may provide for simple windows. In the next version of the software, labeled windows need to be supported in addition to simple windows. Without inheritance, the programmer can either "copy-and-modify" the code, or introduce case statements where the two kinds of windows must be distinguished. Both schemes lead to a proliferation of code, but not of programming productivity. Inheritance allows to reuse code more elegantly (but at the expense of certain intricacies as we shall see).

Conservative Extension

Let us first consider the possibility of code reuse by conservatively adding functionality. In order to provide for labeled windows, we define the class `LabeledWindow` by *inheriting* from `Window` and adding methods such as `setLabel` and `drawLabel`. Instances of `LabeledWindow` provide all functionality that instances of `Window` provide and in addition more specialized behavior related to their label. Thus, the class `LabeledWindow` is a specialization of

`Window`. Late binding provides the mechanism with which instances of the subclass can access the functionality of the superclass.

Non-conservative Extension

Most object-oriented languages provide for overriding inherited methods in subclasses by declaring that a method identifier refers to a new method in the subclass instead of an inherited method with the same identifier. An invocation of an instance of the subclass using this identifier will lead to execution of the new method.

As an example, let us assume that the class `Window` supports a method `redraw` that displays the content that the window currently holds. In our class `LabeledWindow`, we would like to redefine the method `redraw` such that it also displays the current label of the window.

Overriding is a powerful and potentially dangerous tool in the hands of the programmer. It is powerful since it allows to reuse code and radically change it along the way. It is dangerous, because the code being reused may not be prepared for the change. Overriding is the issue where an object-oriented language departs from the idea that inheritance models specialization, because in the presence of overriding, a superclass does not necessarily characterize properties of instances of subclasses. Overriding is a heatedly debated feature of object-oriented programming. Taivalsaari [Tai96] gives an excellent introduction to the literature in this field.

2.2.4 Late and Early Binding

> Broadly speaking, the history of software development is the history of ever-later binding time. *Encyclopedia of Computer Science [RR93]*

Late Binding

We saw that object-oriented programming allows a subclass to override a method inherited from a superclass. An application of an instance of the overriding class will result in a call to the new method. We saw in Section 2.2.1 that—apart from defining interfaces to objects—methods are used for structuring functionality similar to procedures in procedural languages. A method may pass control to another method without changing self. One problem emerges here. What happens to the calls to the overridden method issued by methods in the superclass? For example, a method `deiconify` defined in `Window` may call the method `redraw` that we override in `LabeledWindow`. In a framework in which methods call each other directly, the old method will remain in use by methods of the superclass, thus contradicting the user's intention to completely replace it by the new method.

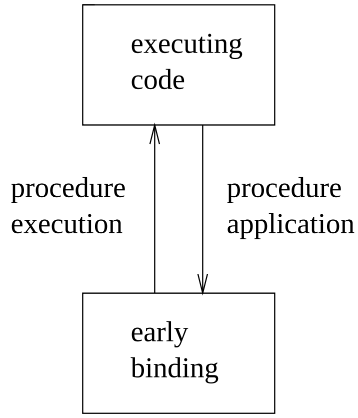
A particular usage of late binding can solve this problem. If we arrange that self is applied to the message `redraw` in the `deiconify` method of `Window` instead of calling the method directly, late binding will lead to execution of the new method `redraw`. If we insist on using late binding for every call of the method in the superclass, we can completely override it in a subclass. With self application, a programmer can open his code for change. On the other hand, self application in combination with overriding can make programs considerably more complex because it is not fixed by the programmer which code is being executed as result of the application. As Coleman and others [C^+93] note "the increased re-usability of class hierarchies must be balanced against the higher complexity of such hierarchies" and later "on the one hand, [inheritance] enables developers to make extensive use of existing components when coping with new requirements; conversely, clients can be exposed to a source of instability that discourages them from depending on a hierarchy of classes".

Early Binding

Late binding enforces the use of the method given by the class of the object being applied. Often, this is too restrictive. Consider the frequent case that an overriding method needs to call the overridden method. The use of late binding here would instead call the overriding method! Instead a mechanism is needed to call the overridden method directly. The classical idiom for this situation is the "super" call, which calls the method of the direct predecessor of the class that defines the method in which the call appears. A super call in a given method always calls the same method and thus implements *early binding*. The term "early binding" refers to the fact that the class, whose method matches the method identifier is known at the time of definition of the method in which the call occurs. A generalization of the super call is a construct that directly calls the method of a given class; we call this *method application*.

Early binding can be used to ensure the execution of a particular method. The programmer can limit the flexibility of designers of derived classes, and thus rely on stronger invariants. In practice, early binding is often used for efficiency reasons. Some object-oriented languages such as SIMULA [DN66] and C++ [Str87] treat early binding as the default and require special user annotations such as "virtual" for methods that may be overridden by descendant classes.

Figure 2.4 Control Flow in Procedural Languages (state centered)



Control Flow

We saw that object application changes the current self, self application does not change self and both use late binding, whereas method application uses early binding. We contrast this somewhat sophisticated control flow to the control flow in other procedural languages, which is depicted in Figure 2.4. In these languages, control flows from one procedure to the next through procedure call, with the possibility to pass parameters along.

The control flow in object-oriented languages is depicted in Figure 2.5. Method application corresponds to procedure application. General object application sets self and uses late binding whereas self application does not change self, but also uses late binding.

2.2.5 Encapsulation

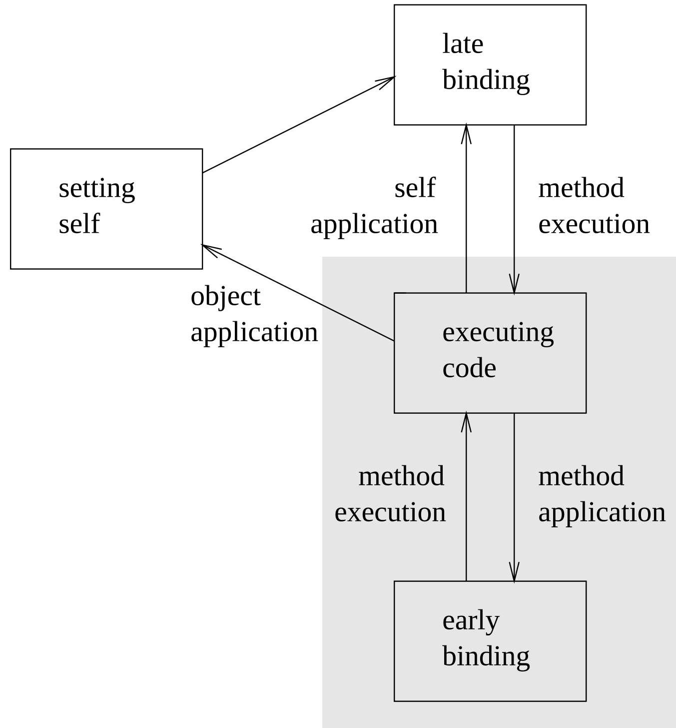
Software consists of different parts that interact with each other. Encapsulation allows to confine this interaction to a specified interface. No interaction between the parts is possible unless this interface is used. Encapsulation is crucial to software development for several reasons:

Independent development. After the interfaces have been defined, different programmers can design and implement the individual parts.

Structure. Encapsulation forces a structure on the software that is often beneficial for implementation and maintenance.

Change. The definition of interfaces can be done according to the expected rate of change. If the interfaces are stable over time, but the individual parts change frequently, then the encapsulation supports maintainability.

Figure 2.5 Control Flow in Object-Oriented Languages (state centered)



Encapsulation allows to view aspects of one part of a software as *internal* and of no significance to other parts. As Ingalls remarks [Ing78] "No part of a complex system should depend on the internal details of any other part."

Encapsulation for Attributes and Methods

Methods often enjoy privileged access to the current object. We say the method is *inside* the current object and *outside* all other objects. For example, Smalltalk [GR83] generally allows access to attributes of only the current object. Other languages allow to restrict the visibility of attributes statically. C++ [Str87] and Java [AG96] allow to declare attributes as *private* in which case they can only be accessed within the class in which the attribute was declared, or *protected* in which case they can be accessed additionally in all classes that inherit from this class (in Java additionally within the package of this class).

Object-oriented languages provide access to the current object such that it can be passed around in messages and stored in the state. For the current object the keyword `this` was introduced by SIMULA and adopted by Beta [KMMPN83], C++ [Str87], and Java [AG96]. Smalltalk uses `self`. Other languages force (CLOS [Ste90]) or allow (Objective Caml [RV97]) user variables to play the role of self.

The languages C++ and Java allow to statically restrict the visibility of methods similar to attributes. Object and method application can be limited by declaring methods private or protected. Private methods can be accessed only within the defining class and protected methods additionally in all classes that inherit from it.

2.3 Objects and Concurrency

In practice, many applications naturally exhibit a concurrent structure. The question arises how concurrency and object-oriented concepts interact. The primary abstractions underlying object-oriented programming have been designed in a sequential context. Unfortunately, a number of problems arise when these abstractions are carried over to a concurrent setting. This situation gave rise to the research area of concurrent object-oriented programming. From the perspective of programming languages, the main issues in concurrency are

- how the programmer can create concurrent computation and
- how different concurrent activities can communicate and synchronize with each other.

From the perspective of object-oriented programming, it is tempting to integrate support for these two issues in the object model.

Integrating the first issue into the object model leads to the concept of an active object that embodies a concurrent thread of control. This approach is taken to the extreme by actors languages [Hew77, HB77] where every data item is represented by an active object. While being conceptually clean, actors make it difficult to write sequential programs, which limits their applicability in practice. Less radical is the actors-like language ABCL [YBS86] in which the behavior of active objects is defined with sequential LISP-like routines—appropriately extended by concurrent constructs. With the exception of ABCL [TMY93], no practical concurrent language uses active objects as basic notion. On the other hand, we shall see that the notion of an active object is often useful and can often be expressed as higher abstractions.

Regarding the second issue, a common task is to achieve mutual exclusion, i.e. to prevent two concurrent threads from operating on the same object, possibly corrupting its state. It is claimed that general mutual exclusion (sometimes misleadingly called *atomicity* of objects [Löh93, LL94]) is a *natural* integration of objects and concurrency [Ame87, Car93]. On the other hand, mutual exclusion is perceived to be too restrictive in practice [Löh93, WL96]. Our own experience showed that while general atomicity can be helpful in a programming context with massive concurrency, enforced mutual exclusion causes more harm than good in a coarse-grained concurrent environment. We shall discuss this issue in more detail in Chapters 9 through 11.

To illustrate that a variety of approaches are conceivable regarding these basic design decisions we give the following table, in which we group a number of existing concurrent object-oriented languages according to whether they enforce active objects and mutual exclusion.

<table>
<tr><td colspan="2"></td><td colspan="2">mutual exclusion enforced</td></tr>
<tr><td colspan="2"></td><td>no</td><td>yes</td></tr>
<tr><td rowspan="2">active objects enforced</td><td>no</td><td>Smalltalk
Emerald [BHJL86]
CEiffel [Löh93]
Java</td><td>Eiffel [Mey93]
Maude [Mes93]</td></tr>
<tr><td>yes</td><td>ABCL
Polka [Dav89]</td><td>Eiffel|| [Car93]
POOL-T [Ame87]</td></tr>
</table>

For an overview of concurrent object-oriented languages consider [AWY93, Con93].

As varied as the approaches of concurrent object-oriented languages towards mutual exclusion and activeness are the synchronization mechanisms. The main means of synchronization of most languages is to equip methods with *synchronization code* that determines when an invocation can proceed. Arguably the most elegant synchronization mechanism in concurrent programming is provided by the logic variable. Synchronization is simply achieved through availability of information. Object-oriented extensions of concurrent logic languages like Vulcan [KTMB87], A'UM [YC88] and Polka [Dav89] use the logic variable as central synchronization concept.

2.4 Further Issues

Multiple Inheritance

So far, we assumed that a class can inherit only from at most one other class. We call this *single inheritance*. Many object-oriented programming languages allow to inherit from and thus merge the functionality of several classes, which is called *multiple inheritance*. Instead of spanning a tree of classes related by the inheritance as in single inheritance, multiple inheritance spans a directed graph.

Obviously, a super call is ambiguous if used in a class derived from several superclasses. Thus languages with multiple inheritance like C++ and Eiffel have to provide the more general method application.

In the context of multiple inheritance, a different possibility of overriding emerges. Methods with the same identifier may be inherited from classes of which neither one is ancestor of the other. Some languages resolve such conflicts by imposing a total ordering on the inheritance graph while others treat them as programming errors.

An important programming technique that is provided by multiple inheritance is to factor out useful functionality into classes that do not inherit from any other class, so-called *mixin classes*. If they are well designed they can greatly contribute to code sharing and structuring. Particularly popular is the technique to group interface and implementation aspects into separate classes and combine them with multiple inheritance (Marriage of convenience). A comprehensive study of different notions of multiple inheritance is presented in [Sin94].

Structure and State of Objects

Classes define the identifiers through which their attributes can be accessed. Thus the instances of a class share a common structure. The attributes themselves, however, are mutable and thus not shared among the instances. Some object-oriented languages such as CLOS, Objective Caml and SICStus Objects [SIC96] allow to define initial values of attributes such that instances share attributes at creation time, while other languages such as Smalltalk, C++ and Java leave initialization up to initialization methods which are called constructors in C++ and Java.

Attributes can change over time. Sometimes an attribute may not need to be changed. This knowledge provides strong invariants to the programmer and compiler. The language Objective Caml allows to declare attributes *immutable* and thus to exploit these invariants.

Several objects can be made to temporarily share attributes by assigning the same value to them. Smalltalk provides support for permanent sharing within the extension of a class through class attributes. The same effect can be achieved in C++ and Java by declaring components as `static`.

Object-based Programming

The term “object-oriented programming” is commonly used for languages that describe objects by classes which are related via inheritance. A related strand of research originated from the observation that classes are not needed if objects are allowed to define methods. Corresponding languages are commonly referred to as *object-based*. Examples for object-based languages are Self [US87] and

Obliq [Car95]. A consequence of avoiding classes is that the distinction between methods and attributes is not needed. Instead, we shall just talk about *properties*. Common to all object-based languages is the ability to *clone* an object, i.e. make a copy of all (or some designated) properties of an object. The clone is distinct from the original by its new identity. Object-based languages provide means to extend the functionality of an existing object. Together with cloning this provides for code reuse without inheritance. Abadi and Cardelli [AC96] give an overview of object-based programming concepts.

The most striking feature of object-based systems is their simplicity. We shall take advantage of this simplicity when we describe first steps towards objects in Chapter 4 in an object-based setting.

Meta-classes

In order to minimize the number of concepts, it is tempting to view classes as objects. Operations on classes such as instantiation then can be represented as object application. The question arises of which class these classes are an instance. Different languages provide different answers to this question, leading to systems of varying complexity and expressivity. Smalltalk takes the stand that with each class creation, a new meta-class is implicitly created whose sole instance is the explicitly created class. Apart from a singularity at the root, the meta-class hierarchy parallels the ordinary class hierarchy. The main purpose for this setup is to allow for class methods such as specialized object creation methods and class variables that are shared among the instances. More flexible is the meta-object protocol of CLOS in which meta-classes can be explicitly defined, giving rise to an experimentation platform for object-oriented language design of unprecedented expressivity [KdRB91]. Much simpler is the concept of Java, in which every class is instance of the fixed class `Class` which provides some debugging and self-documentation functionality. In Chapter 12, we present a novel flexible integration of concurrent objects in a meta-object protocol.

Implementation

The most distinguishing feature of object-oriented languages, namely late binding with inheritance, is also the most critical implementation issue. Implementation techniques have been developed that allow the programmer to assume fast constant time attribute access and late binding for all practical purposes. We shall discuss the implementation issues in object-oriented programming in more detail in Chapter 8.

2.5 Historical Notes

The first language that was designed with specific focus on object-oriented programming was SIMULA, which was conceived by Dahl and Nygaard to solving simulation problems [DN66]. With respect to the aggregation facility of SIMULA, they point out “the similarity between processes [objects] whose activity body is a dummy statement, and the record concept” which was introduced as a programming language construct by Hoare and Wirth [HW66] three months before in the same journal. The object-oriented abstraction was adopted by Goldberg and Kay as the central programming metaphor for Smalltalk-72 [GK76].

In the functional programming community, it was known early on that lexically scoped higher-order programming with state can model essential aspects of object-oriented programming. Steele [Ste76] shows that Scheme can implement “procedural data structures” that are accessed by late binding. Object-oriented extensions to Lisp such as Flavors [Moo86] and CLOS [Ste90] show that practical and extremely powerful object systems can be constructed as syntactic extensions of Lisp.

The developers of C++ [Str87] had to face the problem of integrating object-oriented concepts in the low-level programming language C. Thus, a syntax-oriented approach as in object systems for Lisp was not possible. Instead, the language semantics needed substantial extensions.

Java is an object-oriented language that provides a leaner and more elegant object model than C++, automatic memory management, a simpler and safer type system, and integrates a simple and powerful model of concurrency.

"Why do you wish to see Oz?" [the Tin Woodman] asked.
"I want him to send me back to Kansas, and the Scarecrow wants him to put a few brains into his head," she replied.
The Tin Woodman appeared to think deeply for a moment. Then he said:
"Do you suppose Oz could give me a heart?"
"Why, I guess so," Dorothy answered. "It would be as easy as to give the Scarecrow brains."
"True," the Tin Woodman returned. "So, if you will allow me to join your party, I will also go to the Emerald City and ask Oz to help me."
"Come along," said the Scarecrow heartily, and Dorothy added that she would be pleased to have his company. So the Tin Woodman shouldered his axe and they all passed through the forest until they came to the road that was paved with yellow brick.

Chapter: The Rescue of the Tin Woodman

Chapter 3

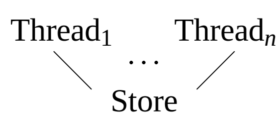
Small Oz

In this chapter, we describe the language Small Oz, a simplified version of Oz. The description follows the Oz Programming Model (OPM) [Smo95], a programming model underlying Oz.[1] OPM adds higher-order programming and explicit concurrency to the framework of concurrent constraint programming and extends functional programming by introducing data-driven synchronization of concurrent threads through logic variables.

Section 3.1 describes Basic Oz, a simple thread-based concurrent constraint language with first-class procedures. Section 3.2 extends Basic Oz by records, a data structure that will be used heavily throughout the presentation. Section 3.3 introduces imperative programming to Basic Oz in the form of cells. Section 3.4 provides convenient syntactic extensions, resulting in Small Oz.

3.1 Basic Oz

Concurrent computation in Small Oz is organized in threads and based on a shared memory model. All threads of control have access to a shared *store* through which they communicate.



Synchronization among threads is provided by a segment of the store called *constraint store* which is described in Section 3.1.1. In Section 3.1.2, we describe computation in Basic Oz, and in Section 3.1.3 we illustrate Basic Oz with an example.

[1]This presentation differs from OPM in [Smo95] in that aspects that are irrelevant to the topic of this thesis or that would unnecessarily complicate the presentation are left out.

3.1.1 Constraint Store

The information in the constraint store can be entered and accessed through *variables*. A variable is a placeholder for values. In Basic Oz, the only possible values are integers and names. Names are values without any structure. There are infinitely many names. We denote names by Greek letters such as ξ. The names **`true`** and **`false`** represent the boolean values. The name **`unit`** is used as synchronization token.

At any point in time the constraint store consists of a constraint which is a finite conjunction of *basic constraints*. Basic constraints have the form:

- $x = y$, where x and y are variables.
- $x = c$, where x is a variable and c is a value.

We use lower-case italic font for meta-variables, i.e. variables ranging over other syntactic constructs. If the constraint in the constraint store entails $x = c$ for some value c, we say that x is bound to c. For example, in a constraint store with the constraint $\mathsf{X} = \mathsf{Y} \wedge \mathsf{Z} = 1$ the variable Z is bound to the integer 1, whereas the variables X and Y are not bound. We will use the constraint store to synchronize computation by waiting for variables to become bound in the constraint store. With this setup, synchronization has the property that once a synchronization condition becomes met, it remains so forever. This property greatly simplifies programming concurrent applications and reasoning over concurrent programs.

Basic Oz is designed such that the constraint in the constraint store is always consistent. This is achieved by fixing the initial constraint store to the trivially satisfiable empty conjunction of basic constraints and limiting the way it can be altered to *telling* a basic constraint. Telling a basic constraint ψ to a constraint store containing a constraint ϕ results in $\phi \wedge \psi$ if $\phi \wedge \psi$ is satisfiable, and raises an exception otherwise.[2] If $\phi \wedge \psi$ is satisfiable, we say that ψ is consistent with the constraint store.

For example, to the store in the above example, we may tell the constraint $\mathsf{X} = 2$, resulting in the constraint store

$$\mathsf{X} = \mathsf{Y} \wedge \mathsf{Z} = 1 \wedge \mathsf{X} = 2$$

Telling the constraint $\mathsf{Y} = \mathsf{3}$ to this constraint store raises an exception, since it is not consistent with the constraint store.

[2] The exception handling mechanism of Oz is largely independent of an object system and thus not described in this thesis. Thus “raising an exception” means for our purpose aborting the program and issuing an error message.

Figure 3.1 Syntax of Basic Oz Statements

S	::=	x = e	*tell statement*
	|	$S_1\ S_2$	*composition*
	|	**skip**	*empty statement*
	|	**local** x **in** S **end**	*declaration*
	|	**case** x **of** p **then** S_1 **else** S_2 **end**	*conditional*
	|	**proc** $\{x\ \overline{y}\}$ S **end**	*procedure definition*
	|	$\{x\ \overline{y}\}$	*procedure application*
	|	**thread** S **end**	*thread creation*
	|	x = ~y | x = y (+ | - | > | >=) z	*arithmetic statement*
e	::=	c	*simple value*
	|	x	*variable*
p	::=	c	*simple pattern*
c	::=	*integer* | **true** | **false** | **unit**	*constant*
x, y	::=	*variable*	*variable*
$\overline{x}$	::=	ε | $x\ \overline{x}$	*list of variables*

3.1.2 Computation

Figure 3.1 describes the syntax of Basic Oz statements. For integers we use the usual notation. We use the prefix ~ for negative integers. For variables, we allow sequences of alphanumeric characters starting with an upper case letter. Examples for variables are `X` and `Apply`.

Computation in Basic Oz takes place in *threads* that have access to a shared *store* through which they synchronize and communicate with each other. The store has two distinct compartments: the *constraint store* and the *procedure store*. The procedure store contains a mapping from names to procedures. Each element of the mapping has the form $\xi : \overline{x}/S$, where ξ is a name by which the constraint store can refer to the procedure, $\overline{x}$ are the formal arguments of the procedure and S is its body.

Computation proceeds when threads are *reduced*. Each thread maintains a stack of statements. Reduction is only possible on the topmost statement of the stack. Reduction pops this statement from the stack and can have the following additional effects:

- New information is told to the store.

- One or more new statements are pushed on the stack of the reducing thread.
- A new thread is created.

Only one thread can carry out a reduction at a time; this policy is called *interleaving semantics*. Interleaving restricts but does not preclude parallel implementation.

A statement can be either unsynchronized or synchronized. Reduction of an unsynchronized statement does not depend on the constraint store. Reduction of a synchronized statement can only proceed, if the constraint store contains sufficient information. Reduction of threads is fair in a sense that if reduction of a thread can proceed, it will eventually do so.

Tell statement. A tell statement of the form $x = c$ or $x = y$ is unsynchronized. Its reduction results in telling the corresponding basic constraint to the constraint store.

Composition. A composition of the form $S_1\ S_2$ is unsynchronized. Its reduction pushes S_1 and S_2 on the stack of the reducing thread such that S_1 is on top of S_2. From now on, when we talk about "pushing statements" we mean "on the stack of the reducing thread".

Empty Statement. An empty statement **`skip`** reduces without effect.

Declaration. A declaration statement of the form

$$\texttt{local}\ x\ \texttt{in}\ S\ \texttt{end}$$

is unsynchronized. Reduction chooses a fresh variable u (i.e. a variable that is not used so far) and pushes the statement $S[u/x]$. The statement $S[u/x]$ is obtained from the statement S by replacing every free occurrence of the variable x with u. Declaration introduces a new variable x whose scope is restricted to this statement.

Conditional. A conditional statement of the form

$$\texttt{case}\ x\ \texttt{of}\ c\ \texttt{then}\ S_1\ \texttt{else}\ S_2\ \texttt{end}$$

is synchronized. It can only reduce if x gets bound. We say that the statement is synchronized *on* x. If x gets bound to the value c, reduction pushes S_1, and otherwise, reduction pushes S_2. Using the metaphor of concurrent constraint programming, we call this operation *ask*.

Procedure Definition. A procedure definition of the form

$$\texttt{proc}\ \{x\ \bar{y}\}\ S\ \texttt{end}$$

is unsynchronized. Reduction chooses a fresh name ξ, adds the pair $\xi : \bar{y}/S$ to the procedure store, and pushes the statement $x = \xi$. Note that the property of the procedure store to contain a mapping remains unchanged since ξ is a fresh name. The variables $\bar{y}$ are the formal arguments of the newly defined procedure x.

Procedure Application. An application of the form

$$\{x\ \bar{y}\}$$

is synchronized on x. The variable x must be bound to a name ξ for which there is an entry $\xi : \bar{z}/S$ in the procedure store such that the length of $\bar{z}$ is equal to the length of $\bar{y}$. Reduction pushes $S[\bar{y}/\bar{z}]$, thus replacing formal by actual parameters.

Thread Creation. A thread creation

$$\texttt{thread}\ S\ \texttt{end}$$

is unsynchronized. Reduction creates a new thread with an empty stack and pushes S on the stack of this thread.

Arithmetic Statement. An arithmetic statement of the form

$$x \texttt{=} y\ (\texttt{+} \mid \texttt{-} \mid \texttt{>} \mid \texttt{>=})\ z$$

is synchronized on y and z. Both must be bound to an integer value, otherwise an exception is raised. We say that the arithmetic statement is synchronized on y and z *to be* integers. Reduction pushes $x = c$ with the correct value c according to integer arithmetics. The comparisons > and >= return the boolean values **`true`** and **`false`**. Integer negation x `=` `~`y is similar and synchronizes on y to be an integer.

3.1.3 Example

A program, represented by a statement with no free variables, is executed by creating a thread whose stack is empty, connected to a store of which all compartments are empty. The program is pushed on the stack and reduction can start.

For example, consider Program 3.1 in which we allow ourselves to simultaneously declare several variables as an obvious extension to declaration. Execution

Program 3.1 An Example for Higher-Order Programming

```
local MakeAdder AdderOne One Two Result in
   proc {MakeAdder X Adder}
      proc {Adder Y Z}
         Z = X + Y
      end
   end
   One = 1
   {MakeAdder One AdderOne}
   Two = 2
   {AdderOne Two Result}
end
```

introduces five fresh variables and replaces the free occurrences of `MakeAdder`, `AdderOne`, `One`, `Two` and `Result` by them. The reason for introducing these fresh variables is to prevent capturing. Where there is no danger for capturing as in this example, we will keep talking about the original variables. Thus, `MakeAdder` will refer to the variable that replaced `MakeAdder`.

At that point, we have a composition of five statements on the stack. Since composition is unsynchronized and results in pushing of both expressions on the stack in the obvious order, the associativity of composition does not matter in that $E_1\ (E_2\ E_3)$ and $(E_1\ E_2)\ E_3$ behave equally if we consider the cost of composition irrelevant.[3] Thus we can assume that we have five statements on the stack. Execution of the first statement

```
proc {MakeAdder ···} ··· end
```

creates a new name ξ, enters the pair ξ : `X Adder`$/\cdots$ to the procedure store and binds the variable `MakeAdder` to ξ. The second statement

```
One = 1
```

will bind `One` to 1. The third statement

```
{MakeAdder One AdderOne}
```

will replace itself on the stack by the body of the procedure `MakeAdder` where we replace formal by actual parameters. Thus the top of the stack is now

```
proc {AdderOne Y Z} Z = One + Y end
```

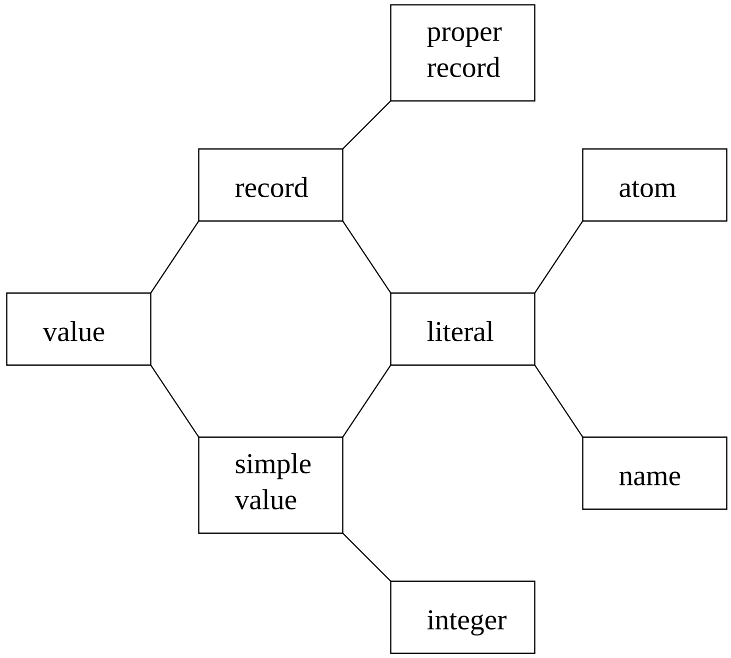
Execution binds `AdderOne` to the corresponding procedure. The forth statement

```
Two = 2
```

binds `Two` to 2. The fifth statement

[3]In a realistic implementation, composition does not incur reduction steps but is realized by sequential execution of code.

Figure 3.2 Value Types in OPM




```
{AdderOne Two Result}
```

applies the procedure `AdderOne` and after execution of the addition the variable `Result` becomes bound to 3.

Note that the variable `X` is statically bound by the formal argument of the procedure `MakeAdder` and thus exemplifies lexically scoped higher-order programming.

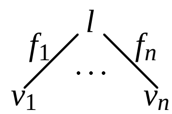
3.2 Atoms and Records

We extend Basic Oz in order to provide a richer set of data structures by adding atoms and records and define convenient relationships of these types with names and integers.

3.2.1 Atoms and Records in the Constraint Store

Figure 3.2 displays the final hierarchy of values types in Small Oz.

A literal is either an atom or a name. Atoms are symbolic values that have an identity made up of a sequence of characters. To distinguish atoms from other syntactic entities, we require that they are enclosed in quotes `´`, which can be omitted if the sequence consists only of alphanumeric characters starting with a lower case letter. Examples for atoms are `paul` and `´|´`. A *simple value* is either a literal or an integer. Given a literal l, n pairwise distinct simple values $f_1, \cdots, f_n$, and n values $v_1, \cdots, v_n$, $n \geq 0$, a record is an unordered tree of the form



We call l the *label*, $f_1, \cdots, f_n$ the *features*, and $v_1, \cdots, v_n$ the *fields* of the record. The features of a record are required to be pairwise distinct so that they identify the fields of the record. We say that v_i is *the field at feature* f_i. Records with no features are identified with their label and thus are literals, whereas other records are called *proper records*. The set of all features of a record is called its *arity*.

We extend the notion of a basic constraint to allow for records.

- $x = l(c_1 : x_1, \cdots, c_n : x_n)$, where x and x_i are variables, l is a literal and $c_1, \cdots, c_n$ are pairwise distinct simple values.

The constraint in the constraint store represents a first-order formula of a structure with equality that can express records. Such a structure is given in [ST94] along with efficient algorithms for entailment, disentailment and satisfiability.

If the constraint in the store entails $\exists x_1 \cdots x_n : x = l(c_1 : x_1, \cdots, c_n : x_n)$ for some n, some literal l and some simple values $c_1, \cdots, c_n$, we say that x is bound to a record with label l and features $c_1, \cdots, c_n$. Note that the variables x_i are not necessarily bound. Being a bit sloppy in terminology, we call the variable x_i to *be* the field of x at feature c_i, even if x_i is not bound yet.

Thus, variables may refer to partial information on records. For example, in a constraint store with the constraint

$$\mathtt{V} = \mathtt{f}(\mathtt{a} : \mathtt{X}, \mathtt{b} : \mathtt{Y}) \wedge \mathtt{X} = 1$$

the variable `V` is bound to a record, whose field at feature `a` is the integer `1`, and whose field at feature `b` is not known yet. Nonetheless, we can call `Y` the field of `V` at feature `b` since any value that may become the field at feature `b` can be referred to by `Y`.

3.2.2 Operations on Records

Figure 3.3 describes the extension of the syntax of Basic Oz for records and atoms. For several operations on records we overload the syntax of procedure application instead of inventing a new syntactic construct for each of them. Making them indistinguishable from procedure application is justifiable since the programmer does not care if an operation is defined by the language semantics or by a standard library.

Figure 3.3 Syntax Extension for Records

S	::=	$\cdots$	
	|	$x = y{=}{=}z$	*equality test*
	|	$x = y$`.`z	*field selection*
	|	`{Label` x y`}`	*label access*
	|	`{HasFeature` x y z`}`	*feature test*
	|	`{AdjoinAt` x y z v`}`	*record adjunction*
	|	`{NewName` x`}`	*name creation*
e	::=	$\cdots$	
	|	$x(x_1 : y_1 \cdots x_n : y_n)$	*record*
p	::=	$\cdots$	
	|	$x(x_1 : y_1 \cdots x_n : y_n)$	*record pattern*
c	::=	$\cdots$	
	|	*atom*	*atom*

Tell statement. A tell statement of the form $y{=}x(x_1 : y_1 \cdots x_n : y_n)$ is synchronized on x to be a literal and on $x_1, \cdots, x_n$ to be pairwise distinct simple values. If the variable x is bound to the literal l and $x_1, \cdots, x_n$ to the pairwise distinct simple values $c_1, \cdots, c_n$, respectively, the tell statement results in telling the basic constraint $y{=}l(c_1 : y_1, \cdots, c_n : y_n)$.

Conditional. A conditional statement of the form

```
case x of y(x1:y1···xn:yn)
then S1 else S2 end
```

is synchronized on x, and on y to be a literal that we call l and on $x_1, \cdots, x_n$ to be simple values that we call $c_1, \cdots, c_n$, respectively. If x is bound to a record with label l and features $c_1, \cdots, c_n$, then reduction pushes

$$y_1{=}x.x_1 \ \cdots \ y_n = x.x_n \ S_1$$

Otherwise, reduction pushes S_2.

Equality Test. An equality test of the form $x = y{=}{=}z$ is synchronized. If the equality of y and z is entailed by the constraint store, reduction pushes x = **`true`**, and if the equality of y and z is disentailed, reduction pushes x = **`false`**. Reduction suspends until equality of y and z is either entailed or disentailed. Note that for records, the equality test implements structural equality. Two records are equal if and only if they have the same label, the same features and their fields at corresponding features are all equal.

Field Selection. A field selection of the form x = y`.`z is synchronized on z to be a simple value c and y to be a record with a variable v at feature c. Reduction pushes $x = v$.

Label Access. A label access of the form `{Label` x y`}` is synchronized on x to be a record. Reduction pushes $y = l$, where l is the label of x.

Feature Test. A feature test of the form `{HasFeature` x y z`}` is synchronized on x to be a record and y to be a simple value that we call c. Reduction pushes z = **`true`** if x has the feature c, and reduction pushes z = **`false`** if it does not.

Record Adjunction. Record adjunction of the form `{AdjoinAt` x y z v`}` is synchronized on x to be a record and y to be a simple value that we call c. If x gets bound to a record $l(c_1 : x_1, \cdots, c_n : x_n)$ such that $c_i = c$ for some i, then reduction pushes $v = l(c{:}x_1 \cdots c_i{:}z \cdots c_n{:}x_n)$. Otherwise (if x gets bound to a record $l(c_1 : x_1, \cdots, c_n : x_n)$ such that $c \neq c_i$ for all i), reduction pushes $v = l(c{:}x_1 \cdots c_n{:}x_n\ c{:}z)$.

Name Creation. A name creation of the form `{NewName` x`}` is unsynchronized. Reduction chooses a fresh name ξ and pushes $x = \xi$.

3.2.3 Example

Consider Program 3.2. After declaring all variables in line *%1*, most of them are bound in line *%2* with a tell statement. The variable `F` gets bound to a new name via name creation. The tell statement in line *%3* is synchronized on `F`, `A` and `B`. Since they are all bound to literals, computation can proceed, binding `X` to a record. Similarly, `AdjoinAt` binds `Z` to a record. Finally, the conditional in line

Program 3.2 Example for Record Construction

```
local A B C F One Two Three X Y Z B1 B2 in           %1
   A=a B=b C=c {NewName F} One=1 Two=2 Three=3      %2
   X = F(A:One B:Two)                               %3
   {AdjoinAt X C Y Z}                               %4
   case Z of F(A:U B:V C:W)                         %5
   then Three=Z.C else skip end                     %6
end
```

Figure 3.4 Syntax Extension for Cells

S	::=	`{NewCell` $x\ y$`}`	*cell creation*
	|	`{Exchange` $x\ y\ z$`}`	*cell exchange*

%5 can reduce since `Z`, `F`, `A`, `B` and `C` are bound. Since `Z` has label `F` and features `A`, `B` and `C`, field selection in line *%6* will bind `Y` to `3`.

3.3 Cells

Obviously a notion of state is needed in order to express sequential object-oriented programming. In Small Oz as we described it so far, only a very weak notion of state is supported. For instance, there seems to be no way to write procedures `Access` and `Assign` such that the program fragment

```
{Access C X}
{Assign C Y}
{Access C Z}
```

results in binding different values to `X` and `Z`. It is argued in the functional and logic programming communities that stateless computation facilitates programming and reasoning about programming.

However, even in these communities it is recognized that stateful programming is sometimes necessary and often practical. The most primitive form of state is a read/write memory *cell*, that can hold a variable and be updated to hold a different one. We provide for cells in Small Oz by introducing a new compartment in the store similar to the procedure store that contains a mapping from names to variables. Each element of the mapping has the form $\xi : x$, where ξ is a name by which the constraint store can refer to the cell, and x is the *current content* of the cell. Figure 3.4 shows the new statements providing for cell creation and update in the form of an exchange operation.

Cell Creation. A cell creation of the form `{NewCell` $x\ y$`}` is unsynchronized. Reduction chooses a fresh name ξ, adds the pair $\xi : x$ to the cell store, and pushes the statement $y = \xi$. Note that the property of the cell store to contain a mapping remains unchanged since ξ is a fresh name.

Cell Exchange. A cell exchange of the form

$$\texttt{\{Exchange}\ x\ y\ z\texttt{\}}$$

is synchronized on x to be a name ξ for which there is an entry $\xi : v$ in the cell store. Reduction changes the entry for ξ in the cell store to $\xi : z$ and pushes $y = v$.

Note that interleaving semantics is of particular importance for `Exchange`. It guarantees that after two concurrent exchange operations `{Exchange C X1 Y1}` and `{Exchange C X2 Y2}`, either `X1 = Y2` or `X2 = Y1` holds.

Using `Exchange` we can define the more usual operations to access the current value of a cell and assign the cell to a new value in the form of the procedures `Access` and `Assign`.

```
proc {Access C X}
   {Exchange C X X}
end
proc {Assign C X}
   {Exchange C _ X}
end
```

Thus, the program

```
C = {NewCell One}
{Access C X}
{Assign C Two}
{Access C Y}
```

will bind `X` to the value of `One` and `Y` to the value of `Two`.

3.4 Syntactic Extensions

Small Oz provides a rich computational framework, but is syntactically rather poor. We add some syntactic extensions that are usually called *syntactic sugar* because they make programs more palatable without adding anything substantial.

3.4.1 Declaration

In the following, we will employ an interactive style of programming. We want to execute programs in which we refer to variables of previously entered programs. This is not possible so far, since the scope of variables is always statically restricted either by the body **`local`** or the body of a procedure for formal arguments. For interactive programming, we allow for declaration with open ended scope as in

```
declare X in
X = 1
```

In subsequent programs, we can now refer to X as in

```
declare Y in
Y = X + 1
```

Here we used the integer 1 in an arithmetic statement. Generally, we allow the use of expressions *e* within statements by requiring that a corresponding tell statement precedes it. Thus the above program is an abbreviation for

```
declare Y in
local One in
   One = 1
   Y = X + One
end
```

Every program that is entered in an interactive Oz session runs in its own thread. Thus new information and computation tasks can be entered and run concurrently to ongoing computation.

3.4.2 Lists and Tuples

Lists are data structures that we will heavily use in the following. A list is either empty or consists of a head and a tail, where the tail is also a list. As a convention, we use the atom `nil` for the empty list. Records label `´|´` and the features `1` and `2` represent non-empty lists, where the field at feature 1 is the head and the field at feature 2 the tail of the list. For example, in

```
declare X in
local Y Z V W
in
   X = ´|´(1:Y 2:Z)
   Y = a
   Z = ´|´(1:V 2:W)
   V = b
   W = nil
end
```

the variable `X` is bound to a list with head `a` and tail `Z`, which in turn is bound to a list with head `b` and tail `nil`. Using nesting of expressions, we may also write

```
declare X in
X = ´|´(1:a 2:´|´(1:b 2:nil))
```

Such records with integers from 1 through *n* as features can be abbreviated by omitting the features as in

```
declare X in
X = ´|´(a ´|´(b nil))
```

Lists enjoy particular syntactic support through infix notation as in

```
declare X in
X = a | b | nil
```

We will often employ an alternative syntax for lists with known length as in

```
declare X in
X = [a b]
```

3.4.3 Functional Syntax

As an example, consider the simple task of generating a new list `Ys` from a given list *l* by applying a given procedure `F` to every element of *l*. Program 3.3 implements this task in the form of a procedure `Map`. Using a case statement, `Map` dispatches over the form of `Xs`, and in case it is a nonempty list, it introduces new variables `Y` and `Yr`, binds `Map`'s last argument `Ys` to `Y|Yr`, calls `F` and itself recursively. We can apply this mapping procedure to compute the list of squares

Program 3.3 A Mapping Procedure in Oz

```
proc {Map Xs F Ys}
   case Xs of X|Xr
   then
      local Y Yr in
         Ys = Y|Yr
         {F X Y}
         {Map Xr F Yr}
      end
   else
      Ys = Xs
   end
end
```

of the elements of a list as in

```
declare Squares Square in
proc {Square X Y} Y = X * X end
{Map [1 2 3 4] Square Squares}
```

In this usage, the last argument `Ys` of `Map` is an output argument in a sense that the application binds it to a value. It contributes to the readability of such procedures when we suppress this argument and write it in functional notation (syntactically realizing the correspondence to functional programming mentioned before) as in

```
fun {Map Xs F}
   case Xs of X|Xr
```

```
   then {F X}|{Map Xr F}
   else Xs
   end
end
```

The syntax **fun** ··· **end** is a purely syntactic abbreviation for procedure definition and is not to be confused with mathematical functions.

We can carry functional nesting further by allowing to nest procedure definitions similar to lambda abstractions in functional programming as in

```
declare
Squares={Map [1 2 3 4]
         fun {$ X} X * X end}
```

Note that the $ symbol indicates where the omitted auxiliary variable is inserted in the nested statement. In the course of this presentation, we will introduce more syntactic variations when convenient. An introduction to the syntax of Oz is given in [Smo97], and a formal description in [Hen97b].

3.5 Small Oz in Context

Functional vs Relational Setup. A functional program is an expression that is evaluated as computation proceeds. The syntax of Oz on the other hand is statement-oriented. The execution of statements performs operations on the store such as binding a variable. Functional syntax is introduced by simple syntactic transformation to statements using auxiliary variables. In practice, there is often a strong correspondence between "functional" Oz programs and their counterpart in eager functional languages. Niehren [Nie94] explores the formal relationship between a sub-language of Basic Oz with corresponding programming models of functional programming.

Conditionals. Compared to other ccp languages, the conditional presented here provides a rather weak control construct. Control primitives in other ccp languages like committed-choice or atomic test-and-set are interesting by themselves, however they do not seem to have much to contribute to object-oriented programming. Their ability to implement many-to-one communication of active objects is problematic as we will see in Chapter 10. Note that the language Oz provides a much richer set of control constructs than Small Oz, including committed-choice with deep guards and guarded disjunction.

Records. Every high-level programming language supports compound values of one sort or the other. Examples range from pairing (Scheme), over un-labeled tuples (Erlang) to labeled tuples (Prolog). The records supported by Oz generalize over tuples by allowing any set of simple values as arity whereas tuples are confined to continuous integer domains starting at one.

Usually records are not labeled (Pascal, SML). The reason for labeled records of Oz is largely a historic one.[4] For us labeled records come in extremely handy, since they are the ideal first-class message. The label literal serves as method name and the fields of the record represent the arguments neatly identified by features.

The down side of records of course is their complexity. The simplest way to express a pair is by a record with two fields. The label and arity are redundant, and the removal of such overhead complicates the implementation.

Records in Oz provide an operation that allows to obtain a list of all features of a given record. Small Oz does not provide such an operation since it would force us to introduce *chunks* for object encapsulation as explained in Section 2.2.5 which would make this presentation considerably more complex.

Procedures as First-class Values. One of the key innovations of Oz is to introduce lexically scoped higher-order functional programming into a concurrent constraint language which allows to present it as a generalization of (dynamically typed) functional programming. This enables us to use object-oriented programming techniques that rely on higher-order programming.

Cells. Small Oz follows the tradition of some functional and logic languages in that there is a clear distinction between stateless and stateful computation. Stateless programs provide strong invariants to the programmer especially in concurrent programming. Cells serve as the entry point to stateful computation similar to references in SML and mutable terms in SICStus Prolog.

Cells do not belong to the standard language constructs of ccp. On the contrary, cells are antagonistic to the spirit in which concurrent logic programming was conceived in that they destroy the declarative nature of computation by introducing state. In our view, this criticism is not justified in the context of object-oriented programming, since state lies at the heart of objects. All approaches to objects in concurrent logic programming therefore eventually introduced state in one form or the other. We argue that in this situation, state should be introduced as simply and as orthogonally as possible, and exactly this is done by the cell. Cells will allow us in a straightforward way to express sequential object-oriented programming in

[4] The initial idea of records in Oz was to extend Prolog's labeled tuples to provide for richer structure [ST94].

Chapter 7. Incidentally, cells in combination with the logic variable provide elegant formulations of a wide variety of synchronization mechanisms that we will explore in Chapter 9. In Chapter 10 we will discuss alternative synchronization constructs from concurrent logic programming.

Atomic exchange is a popular idiom in concurrent programming. Usually operating systems libraries for multi-threaded programming provide atomic exchange in the form of a swap operation. In Multilisp [Hal85] atomic exchange plays a central role.

Typing. Oz is a dynamically typed language. Wrong argument types lead to runtime errors. An argument for dynamic typing as opposed to static typing as in SML or Haskell is simplicity of language design. In fact, the definition of suitable type systems for concurrent higher-order languages with state and logic variables is a challenge of its own. For us, a practical advantage of dynamic typing is the ability to adopt known techniques for object-oriented programming in dynamically typed languages such as Scheme, and the relief not having to integrate the resulting object system into a static type system, which would incur another set of interesting research questions [PS94].

Threads. In contrast to languages with fine-grained concurrency such as concurrent logic languages [Sha89] or the actors model of computation [Hew77, HB77], the concurrency model of Oz is more in line with conventional programming languages that extend sequential computation with coarse grained concurrency. The first language that provides language-level access to concurrency is SIMULA [DN66]. Thread level concurrency similar to Oz can be found in other object-oriented languages like Smalltalk [GR83] and Java [AG96].

Synchronization. Logic variables serve as the main synchronization concept in all concurrent logic languages. Logic variables are acknowledged for providing a simple and effective mechanism for data-driven synchronization [Bal91]. Chapter 9 examines the expressivity of logic variables together with cells. As opposed to thread-level concurrency, most concurrent logic languages adhere to a model of fine-grained concurrency that we shall discuss in Chapter 11. Besides Oz, the only language that combines thread-level concurrency with logic variables is PCN [FOT92].

The concept of a *future* used in the functional programming community comes close to synchronization of threads with logic variables. Baker and Hewitt [BH77] give the first clear account (and earlier sources) of this concept. In their computational setup an expression e can immediately evaluate to a future. A new process is devoted to evaluate *e* and thus “make the future’s value available”. Processes that need the future’s value suspend on its availability. Futures have been adopted as the main synchronization mechanism in Multilisp [Hal85] in the form of the construct `(future` *e*`)` which returns a future and evaluates the expression *e* in a new thread. Logic variables generalize over futures in the following way. A logic variable can be created independently of the computation of its value. Thus the variable can be passed around and it is decided at runtime which thread binds it, whereas a future is statically tied to an expression that computes its value.

Logic variables allow synchronization through the availability of information similar to data flow languages [Den74]. In Oz, data-driven synchronization is embedded in a more traditional computation model with explicit concurrency. As with futures, logic variables generalize data flow variables in that their direction of data flow is not statically determined.

"I am going to the Great Oz to ask him to give me some [brains]," remarked the Scarecrow, "for my head is stuffed with straw."
"And I am going to ask him to give me a heart," said the Woodman.
"And I am going to ask him to send Toto and me back to Kansas," added Dorothy.
"Do you think Oz could give me courage?" asked the Cowardly Lion.
"Just as easily as he could give me brains," said the Scarecrow.
"Or give me a heart," said the Tin Woodman.
"Or send me back to Kansas," said Dorothy.
"Then, if you don't mind, I'll go with you," said the Lion, "for my life is simply unbearable without a bit of courage."

Chapter: The Cowardly Lion

Chapter 4

First Steps towards Objects

In this chapter, we present two established models for object-oriented programming. The first model represents sequential objects as procedural data structures in stateful programming with lexical scoping. The second model shows how active objects are modeled in concurrent logic programming languages. These models give first insights to the range of possible concepts available for object-oriented programming in Oz. They also serve as programming examples to deepen the comprehension of Oz and as a base for discussions in following chapters.

4.1 Objects in Functional Programming with State

4.1.1 Procedural Data Structures

It was observed by Steele [Ste76] that a higher-order functional language with state can model objects in the form of "procedural data structures". Abelson and Sussman [AS96] exemplify this approach, emphasizing the capability of lexical scoping to realize encapsulation. In this section, we follow the presentation by Abelson and Sussman and transpose the ideas to Oz.

As an example, consider Program 4.1 which defines a procedure `Transaction` that has access to a cell initialized with `100` which is bound to the local variable `BC`. The procedure `Transaction` can be applied to an `Amount` which leads to adding it to the current content of `BC`, resulting in `NewBalance`. If `NewBalance` is greater than or equal to `0`, the cell `BC` is updated to `NewBalance` and otherwise an error message is issued. The procedure `Transaction` represents a stateful data structure in a sense that holds an integer data item and its behavior changes over time. For example, after a first application `{Transaction ~75}`, the cell `BC` is updated to `25`. After another identical application, an error message is issued. The state of `Transaction` is encapsulated in that it is accessible by calling

Program 4.1 A Stateful Procedure

```
local
   BC={NewCell 100}
in
   proc {Transaction Amount}
      NewBalance={Access BC}+Amount
   in
      case NewBalance>=0
      then {Assign BC NewBalance}
      else {Show "insufficient funds"}
      end
   end
end
```

`Transaction`. This encapsulation is achieved by lexical scoping; the scope of the variable `BC` is limited to the procedure `Transaction`. Furthermore, we can say that `Transaction` holds the cell `BC` as a component and thus provides for aggregation. We can easily see that further cells can be added to implement different components of the data structure.

4.1.2 Classification

We can provide for a rudimentary form of classification by abstraction of Program 4.1 in a procedure as in shown Program 4.2.

Program 4.2 Generating Stateful Procedures

```
fun {MakeTransaction InitialBalance}
   BC={NewCell InitialBalance}
in
   proc {$ Amount}
      NewBalance={Access BC}+Amount
   in
      case NewBalance>=0
      then {Assign BC NewBalance}
      else {Show "insufficient funds"}
      end
   end
end
```

Every call to `MakeTransaction` now creates a new cell initialized with the argument `InitialBalance` along with a procedure that has exclusive access to the cell. Now we can create instances of our transaction scheme as in

Program 4.3 Generating Stateful Procedures with Late Binding

```
fun {MakeAccount Balance}
   BC={NewCell Balance}
   proc {Transaction Amount}
      NewBalance={Access BC}+Amount
   in
      case NewBalance>=0
      then {Assign BC NewBalance}
      else {Show "insufficient funds"}
      end
   end
   proc {GetBalance ?B}
      B={Access BC}
   end
in
   proc {$ M}
      case M
      of transaction then Transaction
      elseof getBalance then GetBalance
      else {Show "message not understood"}
      end
   end
end
```

```
T1={MakeTransaction 100}
T2={MakeTransaction 200}
```

that manage their state independently from each other. We can say that the procedure `MakeTransaction` provides a classification of data structures and that `T1` and `T2` result from instantiating this classification.

4.1.3 Late Binding

Extending the idea of representing data as stateful procedures, Program 4.3 demonstrates how late binding can be incorporated. Like the procedure `MakeTransaction` in Program 4.1, the procedure `MakeAccount` creates a new cell bound to the local variable `BC`. This cell is accessible by the procedures `Transaction` and `GetBalance`. The unary procedure returned by `MakeAccount` accepts atoms `M` of the form `transaction` and `getBalance`. The question mark in front of the formal argument `B` in procedure `GetBalance` is a comment indicating an output argument, i.e. an argument that is going to be bound in the body of the procedure.

Consider the following application of MakeAccount.

```
Account={MakeAccount 100}
```

Applying Account to the atom transaction returns a procedure that can be applied to an amount, resulting in updating the cell encapsulated by Account.

```
T={Account transaction}
{T ~75}
```

Similarly, we can access the current balance of Account by

```
B={{Account getBalance}}
```

Note the use of functional nesting in procedure position. We say that the procedure Account represents an object which accepts messages of the form transaction and getBalance. Each Account object has associated with it two procedures that operate on its state. Such procedures, we call methods. Due to lexical scoping, these methods are not accessible outside of MakeAccount. Thus the procedure MakeAccount defines the interface of its instances in the form of a mapping from atoms to procedures. Every operation on Account objects has to go through this indirection, which is called late binding.

With respect to inheritance, Abelson and Sussman ([AS96], page 200) opine that "a variety of inadequate ontological theories have been embodied in a plethora of correspondingly inadequate programming languages" by which they presumably mean the intricacies of overriding and late binding in object-oriented programming languages. On these grounds they refuse further investigation of inheritance.

4.1.4 Delegation-Based Code Reuse

Friedman, Wand and Haynes [FWH92] pick up the thread of Abelson and Sussman and note that objects as procedures lend themselves naturally to object-based programming with code reuse by delegation. As an example, Program 4.4 where the bank account is extended by a fee that is subtracted for each transaction ("transposed" from a bounded stack example in Scheme in [FWH92], page 225).

The procedure MakeAccountWithFee takes an argument Fee in addition to Balance. It creates a local Account object using MakeAccount from Program 4.3 and defines a new procedure Transaction that calls Account's method transaction with Fee subtracted from the given Amount. The object returned by MakeAccountWithFee returns this Transaction procedure upon receipt of the message transaction. Other messages are directly delegated to its regular Account object. Thus, MakeAccountWithFee reuses the code of MakeAccount by creating the object Account to which all messages except transaction are delegated. The new transaction overrides and reuses the

Program 4.4 Delegation-based Code Reuse with Passive Objects

```
fun {MakeAccountWithFee Balance Fee}
   Account={MakeAccount Balance}
   proc {Transaction Amount}
      {{Account transaction} Amount-Fee}
   end
in
   proc {$ M}
      case M
      of transaction then Transaction
      else {Account M}
      end
   end
end
```

old `transaction` method. The call `{Account transaction}` corresponds to a super call in object-oriented languages.

4.1.5 Discussion

We have seen that objects with encapsulated state and late binding can be expressed in Small Oz in a simple way. Delegation-based code reuse can be supported. It is remarkable that statically scoped higher-order programming with state suffices for these programming abstractions. Apart from atoms that are needed for late binding, no other data structures such as records were employed. However, observe the following deficiencies of the approach presented so far.

- For every object, a new procedure (closure) is created for every method identifier that this object can receive, which incurs a considerable memory overhead.
- Attributes of inherited "classes" are not accessible in "subclasses". Therefore, public attributes need to be modeled by methods.
- Each object needs as many auxiliar objects as there are "classes" from which its "class" inherits directly or indirectly.

Friedman, Wand and Haynes realize that a different representation lends itself more naturally to conventional class-based object-oriented programming, and thus abandon objects as procedures for their further treatment of object-oriented programming. Instead of using procedures, they represent objects and classes as records. We will see in Chapter 7 that our object system is based on the same idea.

In particular, we shall see a similar implementation of encapsulation through lexical scoping and late binding through a mapping from method identifiers to methods represented by procedures.

4.2 Objects in Concurrent Logic Programming

In this section we take a radically different approach towards objects. We follow the lines of Shapiro and Takeuchi [ST83] by representing objects as active entities that read messages from a stream.

4.2.1 Streams

A stream is a data structure to which new information can be added incrementally. Programming languages that provide logic variables lend themselves naturally to a simple implementation of streams. Pairing is usually used for building streams, leading to a representation of streams as incomplete lists. For example, a stream that holds no information yet can be represented by a variable.

```
declare Ms
```

We can add a data item to the stream by binding `Ms` to a pair.

```
declare Mr in
Ms = transaction(~75) | Mr
```

As pairing constructor, we use records with label `´|´` and the features `1` and `2` for which Oz provides particularly convenient syntax (see Section 3.4.2). The first entry of the stream is `transaction(~75)` and the rest of the stream is accessible via the new variable `Mr`.

4.2.2 Stream-based Objects

Stream-based active objects process such a stream by continuously reading its entries.

```
proc {Account Ms Balance}
   case Ms
   of M|Mr then {Account Mr {ProcessMessage M Balance}}
   else skip
   end
end
thread {Account Ms 100} end
```

Here an active object is implemented by a thread that applies the procedure `Account` to the stream `Ms` and an initial balance of `100`. The procedure

`Account` waits until `Ms` becomes instantiated to a pair, computes a new balance from `Balance` and the first entry of the stream `M` using the auxiliar procedure `ProcessMessage`, and calls itself recursively on the rest of the stream and the new balance.

The procedure `ProcessMessage` given in Program 4.5 dispatches on the form of the message and is straightforward.

Program 4.5 Processing Messages of Active Objects

```
fun {ProcessMessage M Balance}
   case M
   of transaction(Amount)
   then
      TmpBalance=Balance+Amount
   in
      case TmpBalance>=0
      then TmpBalance
      else {Show "insufficient funds"}
         Balance
      end
   elseof getBalance(B)
   then
      B=Balance
   end
end
```

Our active object will first process the first entry `transaction(~75)` in the stream `Ms`, leading to a recursive call on the rest of the stream `Mr` and the new balance `25`. Observe that the state of the object is represented by the argument of the recursive call. At this point the thread suspends since `Mr` is not bound yet.

We can put a new message `getBalance(B)` on the stream by instantiating `Mr`.

```
declare B Mrr in
Mr = getBalance(B) | Mrr
```

This will wake up the object's thread and result in binding the variable `B` to `25`. The next recursive call leads to another suspension. The technique of passing a logic variable (here `B`) along in a message to a stream-based object hoping that the object will instantiate it is called *incomplete messages* in concurrent logic programming.

Note that the records on the stream represent messages that are sent asynchronously. From the perspective of the sender, the message sending only consists of adding a constraint to the stream. The computation resulting from it is carried out in the thread dedicated to serve the active object. However, the sending thread may decide to suspend on information on variables that were passed in messages such as `B`. This way, synchronization between sender and receiver can be enforced. The logic variable together with synchronized reduction allows to express data-driven synchronization.

4.2.3 Delegation-Based Code Reuse

As in the previous section, code reuse can be implemented as delegation. We create an account with fee by

```
proc {MakeAccountWithFee Ms Balance Fee}
    AccountMs
    thread {Account AccountMs Balance} end
in
    {AccountWithFee Ms Fee AccountMs}
end
thread {MakeAccountWithFee Ms 100 10} end
```

The procedure `MakeAccountWithFee` declares a new stream `AccountMs` and creates a regular `Account` object using the given `Balance`. Then the procedure `AccountWithFee` is applied to the original stream `Ms`, `Fee` and the new stream `AccountMs`. The procedure `AccountWithFee` shown in Program 4.6 delegates appropriate messages to `Account` by putting them on the stream `AccountMs`. All messages are passed as they are to `AccountMs` except `transaction` messages, which are manipulated to account for the fee.

Program 4.6 Delegation-base Code Reuse with Active Objects

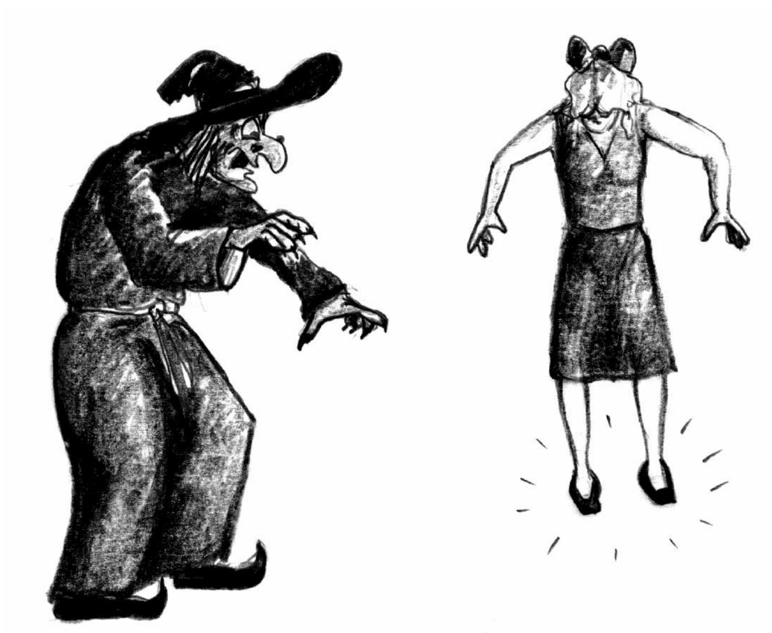
```
proc {AccountWithFee M|Mr Fee AccountMs}
    AccountM|AccountMr=AccountMs
in
    case M
    of transaction(Amount)
    then AccountM=transaction(Amount-Fee)
    else AccountM=M
    end
    {AccountWithFee Mr Fee AccountMr}
end
```

4.2.4 Discussion

We argued in Section 2.3 that—while providing a useful programming idiom—active objects cannot serve as the basic notion of objects. We shall come back to active stream-based objects in Chapter 10 where we will encapsulate them in a more expressive abstraction called *server*. An issue that we need to bear in mind is how messages are represented. In a concurrent setting, it becomes important that messages are first class values that can be manipulated and—as we saw—stored in data structures like streams. To be prepared for this possibility, we shall insist that messages are values in Oz.

Part II

Object-Oriented Programming



This part concentrates on sequential object-oriented programming in Small Oz. In Chapter 5 we describe a simple object system for Small Oz. Chapter 6 enriches this object system with a number of advanced programming techniques. Both chapters introduce the programming concepts of the object system in their own right, whereas Chapter 7 sketches a semantic foundation for it by syntactic reduction of the object-oriented constructs to Small Oz. In Chapter 8, we describe an implementation of Oz based on an abstract machine, discuss critical issues in the integration of objects in such an implementation, present a realistic implementation, and evaluate its performance.

"Well," said the [Wizard of Oz], "I will give you my answer. You have no right to expect me to send you back to Kansas unless you do something for me in return. In this country everyone must pay for everything he gets. If you wish me to use my magic power to send you home again you must do something for me first. Help me and I will help you."
"What must I do?" asked the girl.
"Kill the Wicked Witch of the West," answered Oz.

Chapter: The Emerald City of Oz

Chapter 5

Basic Object System

In this chapter we introduce the elementary features of an object system for Small Oz. We explain how the concepts discussed in Chapter 2 are cast in Small Oz's computational framework. We compare individual design decisions with solutions found in other object-oriented programming languages.

An object-oriented program consists of definitions of classes that describe the structure and behavior of their instances. When we accept that we conservatively extend Small Oz by class definition, we face a set of questions that any object-oriented extension of a programming language has to face. What are the syntactic and semantic means by which classes are defined? How does the construct mix with the rest of the language? More specifically, how does the added construct fit in the scoping rules of the language? These questions will be addressed in this chapter.

5.1 Classes

As usual in object-oriented programming, we support the definition of a class by a specialized syntactic construct that describes the properties of its instances. Consider the example in Program 5.1. This class definition defines the class `Account` whose instances have the attribute `balance` and can be applied to messages with identifier `transaction` and `getBalance`.

For classes, we add a new type of value to Small Oz. The class definition binds the variable `Account` to a class value. Thus classes underlie the visibility scheme of lexical scoping.

The class `Account` describes objects with an attribute that can be referred to by the atom `balance`. We call such atoms *attribute identifiers*. Note that here we use the flat name space spanned by Oz atoms. In the next chapter, we shall see how lexical scoping can be used for attribute identifiers as well.

Program 5.1 A Simple Account Class

```
class Account
   attr balance:0
   meth transaction(Amount)
      balance <- @balance + Amount
   end
   meth getBalance(B)
      B=@balance
   end
end
```

Class definition is integrated in the compositional syntax of Oz in that it can appear inside of any other syntactic construct, including procedures and threads and—conversely—any other syntactic construct can appear within its methods. This design decision contributes greatly to the expressivity of the object system as we shall also see in the next chapter.

Methods are defined with a syntax similar to procedure application. Method heads have the form of records; formal arguments appear as record fields. Thus the transaction method in Program 5.1 has the formal argument `Amount`. The label of the record is the method identifier. For every class, there is a mapping from method identifiers to methods. Instances of the class `Account` can handle messages with label `transaction` and `getBalance`. Similar to attributes, we use atoms for method identifiers and shall extend this convention later. The bodies of the methods contain the code to be executed as operations on instances of `Account`.

5.2 Objects

The variable `A` can be bound to an `Account` object by applying the object creation procedure `New`.

```
A = {New Account transaction(100)}
```

Like classes, objects are values and can be referred to by variables. Objects are the second (and last) value type that we add to the type system of Small Oz. Object creation with the procedure `New` takes as argument an initial message to which the new object is applied. The attribute `balance` has `0` as initial value as indicated in the class definition by **`attr`** `balance:0`. Application of the initial message executes the body of `Account`'s transaction method which consists of the assignment `balance <- @balance + Amount`.

As usual in imperative languages, the right hand side of such an assignment is evaluated before the assignment is carried out; an alternative formulation for the method `transaction` would have been

```
meth transaction(Amount)
   NewBalance=@balance + Amount
in
   balance <- NewBalance
end
```

Assignment is a statement with the "side effect" of assigning an attribute to a value whereas attribute access is an expression which evaluates to the current value of the attribute.

Operations on objects are performed in the form of object application for which we use the syntax of procedure application.[1] Syntactically, messages have the form of records matching the head of a method defined by the object's class. Class and method identifier determine the method to be executed. Object application results in applying this method, replacing actual for formal parameters. As an example, consider the object application

```
{A getBalance(B)}
```

The execution of the corresponding method results in binding `B` to 100. The method `getBalance` is the only way to access the current balance of `A`. Attributes are encapsulated.

Objects in Oz are encapsulated data structures. As in any conventional object-oriented language, object application reduces by pushing the body of the corresponding method on the stack of the current thread. There is no implicit concurrency built in the object system. This stands in sharp constrast ot other object models in concurrent languages. This issue will be further discussed in Part III.

Each new object has its own identity. An equality test with an object that stems from a different object creation results in **`false`**.

Object creation with `New` enforces an initial message. In the case where no initialization is needed, we must use a dummy method. Such a method is provided by the class `BaseObject` which is predefined in the following way.

```
class BaseObject meth noop skip end end
```

5.3 Inheritance for Conservative Extension

Program 5.2 shows how the class `Account` can be conservatively extended with a new method `verboseTransaction` using inheritance. Instances of

[1] This syntactic convention stems from the view of objects as procedural data structures as described in Section 4.1.

Program 5.2 Conservative Extension through Inheritance

```
class VerboseAccount
   from Account
   meth verboseTransaction(Amount)
      {self transaction(Amount)}
      {Show @balance}
   end
end
```

`VerboseAccount` inherit from `Account` the attribute `balance` including its initial value `0` that is referred to in the method `verboseTransaction`. In this sense, the scope of attribute identifiers extends down the inheritance tree. The method `verboseTransaction` refers to the inherited method `transaction` through self application. In methods, the current object can be referred to by the keyword **`self`**. Late binding is used for accessing the method without changing self. An object of class `VerboseAccount` can be considered being of class `Account` since it behaves identically to instances of `Account` with respect to the operations defined on `Account` objects. Thus, we can argue that `VerboseAccount` is a specialization of `Account`.

5.4 Inheritance for Non-Conservative Extension

Conversely, Program 5.3 shows a class `AccountWithFee` which inherits from `VerboseAccount`, but overrides its method `transaction` and thus does not represent a specialization of `VerboseAccount`.

Program 5.3 An Account with Fee

```
class AccountWithFee
   attr fee:5
   from VerboseAccount
   meth transaction(Amount)
      VerboseAccount , transaction(Amount-@fee)
   end
end
```

The new method `transaction` refers to the old method via method application using the syntax *x*`,` *e*, which calls the method of class *x* with method identifier given by *e* without changing self. Note that the class *x* does not have to directly define the method, but can inherit it as is the case in our example.

In the light of this redefinition, a decision that we took in the definition of the method `verboseTransaction` in Program 5.2 becomes important. We used late binding for calling the method `transaction`. The effect is that when an instance of `AccountWithFee` is applied to a `verboseTransaction` message, the corresponding method of `VerboseAccount` will call the new `transaction` method, properly subtracting the fee. Self application in `verboseTransaction` means "call whatever the current `transaction` method is". Had we used method application in `verboseTransaction` such as

```
Account , transaction(Amount)
```

then applying an `AccountWithFee` object to a `verboseTransaction` message would not charge the fee. Note that the choice between late and early binding is not always this obvious. It takes careful design to provide for safely reusable classes and often, reusability and efficiency are in conflict with each other.

5.5 Case Study: Sieve of Eratosthenes

In this section, we consider a slightly less trivial example to show how late binding realizes polymorphism. Furthermore, we shall use this case study in the performance evaluations in Sections 8.6 and 10.4.

We compute prime numbers among the first n natural numbers with the sieve of Eratosthenes. To this aim, we send natural numbers starting with 2 in sequence to a filter object, representing the prime number 2. This filter is the first one of a growing chain of prime filters, terminated by a special object called *last*. Numbers i from 2 to n are passed successively through this chain in the form of messages $f(i)$. This process stops for a given number i at a given filter representing the prime number p, if p is a multiple of i. When a number p made it to *last* it is prime, and a new filter for p is appended at the end of the chain right before *last*.

Program 5.4 shows the corresponding Oz program. We define the classes `Filter`, whose instances represent the prime filters, and `Last`, whose instance is used to terminate the chain. The initial configuration of filters generated by lines *`%2`* and *`%3`* is depicted in Figure 5.1 (a).

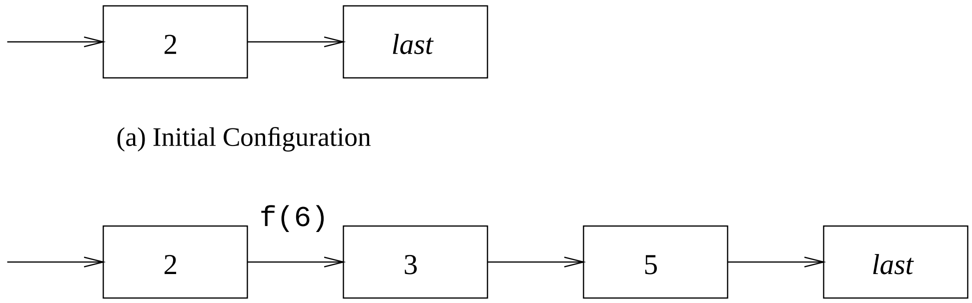
If a filter for prime p is the last one in the chain and receives a number q that is not divisible by p, a new prime filter must be created for q. Instead of testing whether this is the case, we use late binding. The message `f(N)` is simply passed to its neighbor in line *`%1`*. If the neighbor happens to be an instance of `Last` it creates a filter for `N` and inserts it right before `Last`. Figure 5.1 (b) depicts the configuration before the message `f(6)` arrives at filter `3`.

In this program, we use late binding to realize polymorphism. The expression `@next` in line *`%1`* evaluates to an instance of either `Filter` or `Last`. This is ad-hoc polymorphism since the operation exhibits different behavior depending on

Program 5.4 Sieve of Eratosthenes

```
class Filter
   attr next:unit prime:unit
   meth init(Prime Last)
      {Show Prime}
      prime <- Prime
      next <- Last
   end
   meth f(N)
      case N mod @prime \= 0
      then {@next f(N)}                                   %1
      else skip end
   end
   meth setNext(Next)
      next <- Next
   end
end

class Last
   attr previous
   meth init(Previous)
      previous <- Previous
   end
   meth f(P)
      NewFilter={New Filter init(P self)}
   in
      {@previous setNext(NewFilter)}
      previous <- NewFilter
   end
end

TwoFilter={New Filter                                     %2
           init(2 {New Last init(TwoFilter)})}            %3
%% send requests f(I), i=2..100, to TwoFilter
{For 2 100 1 proc {$ I} {TwoFilter f(I)} end}
```

Figure 5.1 Configurations of Sieve of Eratosthenes



(a) Initial Configuration

(b) Configuration Before Number 6 Enters Filter 3

the class of the object. In statically typed languages, this polymorphism enforces an inheritance relationship between `Filter` and `Last`. For comparison, corresponding implementations of this program in C++, CLOS, Java, Objective Caml, SICStus Prolog Objects and Smalltalk are provided [Hen97a].

5.6 Discussion

We extended the compositional syntax of Oz by a similarly compositional class definition construct. A consequence is that classes are referred to by variables and underlie lexical scoping. Object-oriented extensions to languages with lexical scoping such as CLOS [Ste90] or Objective Caml [RV97] generally make use of this possibility. Other languages such as Smalltalk [GR83] or SICStus Objects [SIC96] use a flat name space for classes with the potential danger of "name space pollution" for classes.

For object creation, our system enforces initial messages by the fact that `New` takes a message as argument. This may seem overly restrictive, but in practice most classes rely on initialization methods anyway. We shall give a compelling argument for initial messages in Chapter 9 in the context of concurrency. For simplicity, we do not distinguish constructor methods semantically as is done in C++ and Java.

For simplicity, classes are not objects for which we would need to introduce another class. Smalltalk uses class objects primarily to provide specialized object creation methods, and class attributes, i.e. assignable attributes that are shared by all instances. Creation methods are made obsolete by initialization methods. In the next chapter we shall see how all instances of a class can share common information.

Syntactically, messages have the form of records. We shall see in the next chapter that they actually *are* records and how we can exploit this fact.

This made Dorothy so very angry that she picked up the bucket of water that stood near and dashed it over the Witch, wetting her from head to foot.

Instantly the wicked woman gave a loud cry of fear, and then, as Dorothy looked at her in wonder, the Witch began to shrink and fall away.

“See what you have done!” she screamed. “In a minute I shall melt away.”

“I’m very sorry, indeed,” said Dorothy, who was truly frightened to see the Witch actually melting away like brown sugar before her very eyes.

Chapter: The Search for the Wicked Witch

Chapter 6

Advanced Techniques

This chapter builds upon the simple object system of the previous chapter. We extend this system by a number of new and useful concepts and explore the expressivity of the resulting language.

6.1 Features

The attributes of an object can change over its lifetime and thus provide adequate support for stateful data. However, not all aspects of an object are intended to be changed. It is often argued that stateless computation eases program design and analysis (for a forceful line of argument see [AS96]). The property of an object to be stateless can be useful in the design of concurrent applications [Lea97]. In our experience, it can even be helpful to declare which components of an object can change and which cannot. For this purpose, we introduce—in addition to stateful attributes—stateless object components called *features*. Features are declared similarly to attributes, but with the keyword **`feat`** instead of **`attr`**. Inheritance of features and inheritance of attributes are uniform. The strong correspondence between object features and record features led to adopting the syntax of record access for objects. Thus the feature of an object `O` at `f` is accessed by `O.f`.

Object features are immutable by design and thus enforce a security from change of object components that are not meant for change. By adding features we trade security for simplicity of the language. For example, the attribute `fee` of class `AccountWithFee` in Program 5.3 is initialized with `5` and is not changed by any method. In order to enforce immutability of the fee component for subclasses of `AccountWithFee`, we can declare it as a feature, resulting in Program 6.1. Within the method `transaction`, the feature `fee` is accessed by **`self`**`.fee`. The feature `fee` can also be accessed from outside of the methods of `AccountWithFee` as in

Program 6.1 Use of Object Features in an Account with Fee

```
class AccountWithFee
   feat fee:5
   from VerboseAccount
   meth transaction(Amount)
      VerboseAccount , transaction( Amount-self.fee )
   end
end
```

```
A={New AccountWithFee transaction(50)}
{Show A.fee}
```

Due to outside access, object features provide a particularly convenient way to create record-like structures. One could argue that outside access violates the principle of encapsulation. However, external access to stateless components seems to be less critical from a software development view. In Section 6.5, we shall give a general technique to protect attributes, methods and features from access outside of a class or class hierarchy.

Note that object features introduce a second form of aggregation in our object model. Attributes provide for *stateful* aggregation, features for *stateless* aggregation.

6.2 Free Attributes and Features

As described so far the values of features and the initial values of attributes are defined by the corresponding entry in the class. This means that all objects of a class share the same values for features and initial values for attributes. Sometimes it is desired instead that a feature or initial attribute value of each instance is independent from the other instances. Consider the class `Filter` in Program 5.4. The attribute `prime` is not changed by any method; however, its value is different for every instance. For such situations, we introduce a mechanism that enforces the creation of a fresh variable in every new instance for a given attribute or feature. This variable can then be bound during initialization. Syntactically, this is done by simply leaving out the value in attribute and feature declaration. Program 6.2 shows an alternative formulation of the class `Filter`.

Note that there is a difference between leaving a feature unbound in class definition and declaring it free.

```
class C feat f1:_ f2 ⋮ end
```

Whereas the variable at feature `f1` is shared among all instances of `C`, a fresh variable for `f2` is introduced for every instance. Note that the symbol _ stands for

Program 6.2 Using Free Feature and Attribute in Sieve of Eratosthenes

```
class Filter
   attr next
   feat prime
   meth init(Prime Last)
      {Show Prime}
      self.prime = Prime
      @next = Last
   end
   meth f(N)
      case N mod self.prime \= 0
      then {@next f(N)}
      else skip end
   end
   ⋮
end
```

an anonymous variable; the above class definition is equivalent to

```
local V in class C feat f1:V f2 ⋮ end end
```

if `V` does not occur free in the class definition.

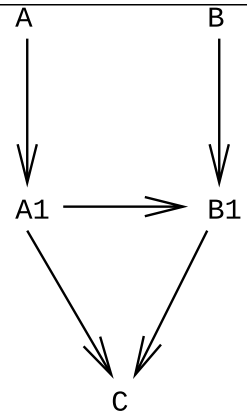
6.3 Attribute Exchange

Often access and assignment of an attribute are carried out in sequence. In concurrent programming, it is not guaranteed that no other thread manipulates this attribute simultaneously and thus corrupts the intended operation on the attribute. Often more complex synchronization can be avoided if atomicity of access and assignment is guaranteed. Thus, we introduce an operation for attribute exchange similar to cell exchange. Syntactically, we achieve attribute exchange by allowing attribute assignment at expression position in which case it returns the old value of the attribute. For example, a method that returns and withdraws the current balance can be written as

```
meth withdrawAll(?B)
   B = balance <- 0
end
```

We shall see in Chapter 9 that attribute exchange plays a central role in the encoding of synchronization techniques. The attribute-related operations of access, assignment and exchange together are referred to as *attribute manipulation*.

Figure 6.1 Example of Inheritance Graph




```
class A ··· end
class B ··· end
class A1 from A ··· end
class B1 from B ··· end
class C from A1 B1 ··· end
```

6.4 Multiple Inheritance

We argued in Section 2.4 that multiple inheritance is a powerful tool to combine the functionality of classes. We are going to support multiple inheritance in a permissive manner in the style of the object-oriented Lisp extensions Flavors [Moo86] and CLOS [Ste90] but with a technical improvement.

Classes may inherit from one or several classes appearing after the keyword **`from`** in the class definition. The classes from which a class inherits directly are called its *parents*. If a class `D` inherits directly or indirectly from another class `A`, then we call `D` *descendent* of `A` and we call `A` *ancestor* of `D`. The parent relation spans a graph of classes. If we add to this graph the linear order between parents as given in the class definition, we get a new graph that we call the *inheritance graph*. For example, the inheritance graph spanned by five classes is depicted in Figure 6.1.

Class definition suspends until all parents are determined. We require that the inheritance graph be acyclic; if it contains a cycle, class definition raises an exception. So far we follow the inheritance scheme of CLOS [Ste90]. The definition of CLOS now extends this graph to a linear order using topological sorting. Inheritance proceeds as if there was single inheritance with a class hierarchy represented by the linearization. Naturally, the linearization is not unique. Thus the semantics of inheritance in CLOS is essentially defined by the algorithm that implements it. In particular, methods[1] that are defined in classes which are unrelated in the inheritance graph can override each other, resulting in programming errors that are subtle and hard to detect. In our example, if the classes `A1` and `B` both define a method `m` and neither `B1` nor `C` override `m`, which method `m` is inherited by `C`? (The

[1]For simplicity, we only talk about methods in this section. The same of course holds for attributes and Oz's object features.

CLOS algorithm will make `C` inherit `B`'s method `m`.) We forbid such situations to avoid programming errors and give a simple declarative description of inheritance instead of defining its semantics by giving a particular algorithm for topological sorting.

Our definition relies on the notion of *closeness* in the inheritance graph. A class *a* is closer to a class *c* than a class *b* if *a* lies on a path from *b* to *c*. In our example, the class `B1` is closer to `C` than `A1`, but `A1` is not closer to `C` than `B`. To avoid ambiguities, we require that for every inherited method *m* of a class *c* there be a class *a* that defines the method which is closest to *c*. The class *c* inherits *a*'s method *m*. If there is no such a closest class, the class definition raises an exception. For example, if both `A1` and `B` define the method `m` and neither `B1` nor `C` override it, the class definition is illegal. If on the other hand all of `A`, `B`, `A1` and `B1` define the method `m`, then `B1` is the closest and thus `C` inherits `B1`'s method `m`.

Of course since classes in Oz are runtime entities, inheritance is also performed at runtime. The programmer needs to keep in mind that illegal use of inheritance is a source of runtime errors.

Drawbacks of this notion of multiple inheritance are an implementation effort in system programming and a runtime penalty for class definitions that use multiple inheritance. In our experience both drawbacks are outweighed by the gain in security.

Other languages are much more restrictive than that. For example, Eiffel [Mey92] does not allow any horizontal overriding, but provides the possibility of renaming identifiers to prepare classes for multiple inheritance. This precludes certain uses of mixin classes as argued by Schmidt and Omohundro [SO93]. Eiffel opts here for even more security and against expressivity.

6.5 Privacy

The only kind of encapsulation offered by the object system so far is the encapsulation of attributes. In this section we use names in combination with lexical scoping to implement a variety of encapsulation techniques such as private and protected identifiers. All other languages provide these encapsulation techniques by specialized compile time notions.

In Oz, literals can be used as features of records. A literal is either an atom or a name. Similarly, we shall allow names as attribute identifiers, feature identifiers and method identifiers. The possibility to create unique names together with lexical scoping gives the programmer full control over the use of these identifiers. For example, in the following code fragment the user restricts the use of the identifiers `PrivateAttr`, `PrivateFeat` and `PrivateMeth` to the class `Example`.

```
local
   PrivateAttr={NewName}
   PrivateFeat={NewName}
   PrivateMeth={NewName}
in
   class Example
      attr !PrivateAttr:0
      feat !PrivateFeat:100
      meth !PrivateMeth(f:X g:Y)
         ⋮
      end
      ⋮
   end
end
```

Ignore the exclamation marks `!` for now; their usage becomes clear soon. The names to which the identifiers are bound are by definition unique and thus cannot be forged. Thus the feature `PrivateFeat` of instances of `Example` cannot be accessed outside the scope of `PrivateFeat` unless the programmer passes `PrivateFeat` to the outside. The method `PrivateMeth` is private in the same sense. The methods of classes that inherit from `Example` cannot access the attribute `PrivateAttr`. However, note that the private identifiers can be passed around like any other Small Oz value, and can therefore overcome protection if the programmer wishes so. For example, if our class `Example` contains the following method

```
   meth getPrivateFeat($)
      PrivateFeat
   end
end
```

the feature `PrivateFeat` can be accessed from outside the class definition as in

```
ExampleInstance={New Example noop}
ThePrivateFeat={ExampleInstance getPrivateFeat($)}
ThePrivateValue=ExampleInstance.ThePrivateFeat
```

Privacy of attributes, features and methods is only guaranteed if the class definition makes disciplined use of the corresponding identifiers. In particular, this is the case, if the private features are only used as usual in field selection, the private attributes only in attribute manipulation and the private methods only in object and method application. Fortunately, this is usually the case and a sufficient condition could be statically checked by a compiler.

The use of private identifiers increases programming security and should be made as convenient as possible. Thus we introduce the following convention.

A variable in the declaring position of an identifier, i.e. after the keywords **feat**, **attr**, and **meth** represents a private identifier. The variable is implicitly bound to a name and its scope limited to the enclosing class definition. Thus the class `Example` above can simply be written as

```
class Example
   attr PrivateAttr:0
   feat PrivateFeat:100
   meth PrivateMeth(f:X g:Y)
      ⋮
   end
   ⋮
end
```

The exclamation mark can be used to avoid the implicit declaration and binding of the variable so that the programmer can decide himself how big the scope of the variable ought to be. This technique is employed by the first of the following examples that demonstrate the flexibility provided by this notion of privacy.

Friends

The programmer can freely specify the scope of variables that are used as attribute, feature and method identifiers. This allows to statically group several class definitions together and let them share identifiers that are not visible from the outside.

For example the following two classes share the identifiers `SharedFeat` and `SharedMeth`.

```
local
   SharedFeat={NewName}
   SharedMeth={NewName}
in
   class C1 from BaseObject
      feat !SharedFeat
      meth !SharedMeth(f:X g:Y)
         ⋮
      end
   end
   class C2
      attr a
      meth useSharedIdentifiers
         I1={New C1 noop}
      in
         a <- I1.SharedFeat
```

```
         {I1 SharedMeth(f:1 g:100)}
      end
   end
end
```

Note that this protection technique follows directly from the use of names bound to statically scoped variables and avoids special purpose compile-time concepts such as `friend` in C++. Like in C++, the friend relation is neither inherited nor transitive.

Protected Identifiers

Another use of private identifiers is to restrict the visibility of an identifier to descendant classes. This concept is known as `protected` in C++. Note that all attributes are protected in Small Oz since they are only accessible within methods. We can extend this protection mechanism from attributes to features and methods by binding their identifier to an attribute. Then, only classes that inherit this attribute have access to the protected identifier. For example consider the following class definitions.

```
class C1
   attr protectedMethod:ProtectedMethod
   meth ProtectedMethod(f:X g:Y)
      ⋮
   end
end
class C2 from C1
   attr a
   meth m
      PM=@protectedMethod
   in
      {self PM(f:1 g:100)}
   end
end
```

Note that a syntactic limitation of Oz prevents a more convenient nesting and forces the introduction of an auxiliary variable `PM` instead of simply `{self @protectedMethod(f:1 g:100)}` which is refused by the compiler.

The cost of this implementation of protected identifiers is one attribute per protected identifier and one indirection through the state for each use of the protected identifier. Protected identifiers rely on the discipline that the corresponding attributes are not changed via assignment or attribute exchange.

6.6 First-Class Messages and Message Patterns

In most object-oriented languages, messages are not first-class citizens in a sense that they cannot be bound to variables or passed as arguments. In Oz, messages are represented by records, which are first-class citizens. In this section, we will exploit this fact.

Sometimes, it depends on a complex computation to which message an object needs to be applied. Instead of applying the object at several places to a different message, we can separate the computation of the message from the object application as in

```
{O {ComputeTheMessage}}
```

Messages can be referred to by variables in object application and similarly in method application as in `C,M`.

Often, it is convenient to keep the arity of acceptable messages flexible and introduce default values for non-specified fields. Most prominent examples are interfaces to complex services such as window systems, which provide a large set of different options of which typically only a few are actually used often. Similar to Lisp, we provide for optional and open arguments for methods.

Optional method arguments are indicated by default values as in

```
meth announce(source:S destination:D
              weight:W<=StandardWeight)
   ⋮
end
```

The message feature `weight` is optional. If it is omitted, `W` is bound to `StandardWeight`. As in Lisp, any expression can occur after <= which is evaluated only if the corresponding feature is omitted.

We also provide for open methods corresponding to the Lisp declaration `&rest`. Consider

```
meth announce(source:S destination:D ...)
   ⋮
end
```

The ellipses indicate that any `announce` message with features `source` and `destination` is accepted. Other features are simply ignored. When ellipses are used we frequently want to be able to refer to the whole message instead of just ignoring other features. This can be done as in

```
meth announce(source:S destination:D...)=Announcement
   case {HasFeature Announcement weight}
   then {CheckWeight Announcement.weight}
   else skip end
```

```
   ⋮
   end
end
```

Message patterns avoid a proliferation of methods as in Smalltalk, C++ and Java where for every possible combination of arguments a new method must be defined. With respect to message patterns we obviously deem the design issue of expressivity more important than simplicity, since in our experience the gain in expressivity clearly outweighs the expense of a relatively straightforward and local extension. We shall discuss implementation aspects of message patterns in Section 8.5.2.

6.7 Higher-Order Programming with State

Recall that procedures in Oz are first class values, and class definition is fully compositional. A consequence is that procedures can be created within methods. Consider the following use of the procedure `ForAll` that provides iteration over a list.

```
class C from BaseObject
   attr a
   meth addAll(Xs)
      {ForAll Xs
       proc {$ X} a <- @a+X end}
   end
   ⋮
end
```

This method adds all elements of a given list `Xs` to the attribute `a`. Note that the state can be referred to within the embedded procedure. In this example there is no doubt to which object the attribute `a` in the procedure refers to since the procedure is being executed in the same environment in which it is defined.

This changes if dynamically created procedures are exported outside of its defining method as in

```
meth getSetPrivate($)
   proc {$ X} Private <- X end
end
```

As enforced by lexical scoping, the attribute `Private` refers to the current object at the time of procedure definition. Thus the receiver of the procedure can freely manipulate the value of the attribute `Private`, which severely breaks encapsulation. This rather pathological possibility that Oz shares with CLOS is a consequence of compositionality and lexical scoping.

6.8 Final Classes

The concept of a public method has two aspects. Firstly, the method can be accessed by any object of a corresponding class, and secondly, the method can be overridden by inheritance. Sometimes, we want the first without the second. Consider for example a class that provides a method `validatePassword`. This method must be accessible from outside, but we certainly do not want it to be overridden to

```
meth validatePassword(P) skip end
```

For such situations we introduce the concept of *final* classes similar to Java. A final class cannot be inherited and thus its methods cannot be overridden. A class can be declared to have the final property by writing **prop** `final` in its declaration. Java also allows to declare single methods as final; for simplicity we do without this feature.

6.9 Classes as First-Class Values

Full compositionality of class definition implies that class definition can appear within procedures, allowing for parameterization of classes. For example, instead of using an attribute or feature for the fee of our class `AccountWithFee` (Programs 5.3 and 6.1), we can parameterize over the class and create a class with a given fee. Consider Program 6.3. We can create a new instance of this parameterized class as follows.

```
A={New {MakeClassAccountWithFee 5} transaction(100)}
```

The application of procedure `MakeClassAccountWithFee` results in a class whose method `transaction` has access to the given `Fee` of `5`. From this class, we create an instance using `New` as usual.

Classes can be parameterized over any of their components such as inherited classes, attribute or method identifiers, initial values (as in this example) etc.

Program 6.3 A Parameterized Class for Account with Fee

```
fun {MakeClassAccountWithFee Fee}
   class $
      from VerboseAccount
      meth transaction(Amount)
         VerboseAccount , transaction(Amount-Fee)
      end
   end
end
```

Program 6.4 Calculator

```
class Calc
   attr arg acc equals
   meth reset
      arg <- 0.0
      acc <- 0.0
      equals <- class $ meth m($) @arg end end
   end
   meth equals($)
      @equals , m($)
   end
   meth enter(N)
      arg <- N
   end
   meth add
      acc <- @equals,m($)
      equals <- class $ meth m($) @acc + @arg end end
   end
end
```

A more elaborate use of first-class classes is adapted from Cardelli [Car94]. Consider Program 6.4. The class `Calc` defines the behavior of a pocket calculator with accumulator. The attribute `arg` hold the last number entered by the user (see method `enter`). The attribute `acc` holds the left-hand side of any operation to be applied, and the attribute `equals` holds the operation to be executed when the user asks for the current value in the form of a class that defines a corresponding method `m`. For instance the method `add` assigns to `equals` a method that adds `arg` to `acc`. The following session shows how to use `Calc`.

```
C={New Calc reset}
{C enter(3.5)} {C add} {C enter(2.0)}
{Show {C equals($)}}
% ==> 5.5
{C reset} {C enter(3.5)} {C add} {C add}
{Show {C equals($)}}
% ==> 10.5
```

This example was used by Cardelli to demonstrate the flexibility of the object-based programming language Obliq. We showed that first-class classes have similar expressivity albeit with slightly more syntactic effort, since methods need to be wrapped in classes and stored in the attribute `equals`. In Obliq, the methods of an object can be directly manipulated. First class methods together with the possibility to directly apply a given method would alleviate this syntactic shortcoming.

6.10 First Class Attribute Identifiers

A byproduct of first-classing identifiers for privacy is that we can pass attribute identifiers to methods. In the following example, we combine this feature with first-class messages. The aim is to provide *dynamic assignment* as a generic mechanism. Dynamic assignment [FWH92] allows to execute a procedure under a temporarily changed environment. As such an environment, we consider the current object's state and allow to perform self application under a temporarily changed attribute.

```
meth dynAssign(Attr NewVal Msg)
   OldVal = Attr <- NewVal
in
   {self Msg}
   Attr <- OldVal
end
```

A call of method `dynAssign` corresponds to the form `dynassign (var exp body)` in [FWH92]. As an example, consider the task to typeset a paragraph `P` with a text width that is temporarily set to 110. This can be done within a class that defines the method `typeset` and the attribute `textwidth` as follows.

```
{self dynAssign(textwidth 110 typeset(P))}
```

Such situations occur frequently in text processing and thus text processing languages such as TEX make heavy use of dynamic scoping and dynamic assignment.

6.11 Case Study: N-Queens

To further deepen the understanding of the object system, we undertake a second case study which allows us to discuss features, local classes, private methods, free features and attributes, defaults in method heads and the final property. Like the previous case study, we shall use this one as well for performance evaluation in Sections 8.6 and 10.4.

Consider the task of placing n queens on an $n \times n$ chess board such that no queen can attack any other according to the rules of chess. Figure 6.2(b) depicts a solution of the 4-queens problem. The idea for the program is due to Chris Moss [Mos94] and implements backtracking with forward checking in an object-oriented setting.

A trivial property of every solution is that each column contains exactly one queen. Thus, we represent each queen by an object with a fixed attribute `column`

Program 6.5 N-Queens in Small Oz

```
local
   class NullQueen from BaseObject
      meth first             skip            end
      meth next              {Show ´no sol´} end
      meth canAttack(R C $) false            end
      meth print             skip            end
   end
in
   class Queen from BaseObject
      prop final
      attr row column
      feat n neighbor

      %% init creates new queen and its
      %% neighbor to the left.

      meth init(column:C<=N n:N)
         self.n        = N
         self.neighbor = case C>1
                         then {New Queen
                                init(column:C-1 n:N)}
                         else {New NullQueen noop} end
         @column       = C
      end

      %% first asks the neighbor for a solution,
      %% guesses 1 and goes to testOrAdvance

      meth first
         {self.neighbor first}
         row <- 1
         Queen , testOrAdvance
      end

      %% if a queen further left can attack, try next

      meth TestOrAdvance
         case {self.neighbor canAttack(@row @column $)}
         then Queen , next
         else skip end
      end
```

continued on next page

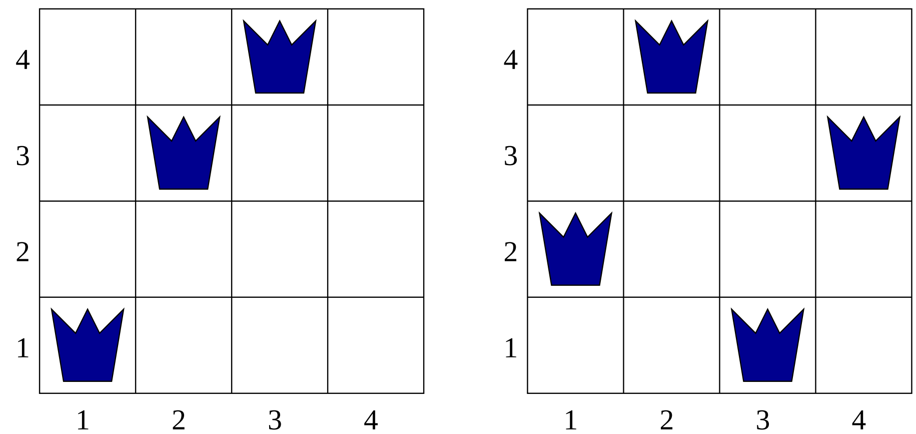
continued from previous page

```
      %% canAttack checks if self can attack.
      %% If not, it asks if neighbor can attack.

      meth canAttack(R C $)
         @row==R
         orelse R==@row+@column-C
         orelse R==@row-@column+C
         orelse {self.neighbor canAttack(R C $)}
      end

      %% if we reached the limit, the neighbor has
      %% to change, otherwise try next row

      meth next
         case @row==self.n then
            {self.neighbor next}
            row <- 1
         else row <- @row+1 end
         Queen , TestOrAdvance
      end

      meth print
         {self.neighbor print}
         {Show @row#@column}
      end
   end
end

X={New Queen init(n:8)} {X first} {X print}
```

and a mutable attribute `row`. Thus the queens can only move vertically. To solve the problem, we ask the rightmost queen for its first solution. If a queen *q* receives such a request, it asks its left neighbor *l* for the first partial solution among the queens left of *l* including *l*. Then *q* places itself in the first row and checks if this is consistent with the partial solution. If it is, *q* is done. If not, it moves its queen one step ahead and checks again. If it reaches the last column, it asks its left neighbor for the next partial solution and starts all over from the first row. Program 6.5 shows the Oz program that implements this solution.

Figure 6.2(a) depicts the situation where the third queen tries to place itself after having asked the second queen for the first partial solution. It reached the top without finding consistency with the partial solution to the left. Thus in the next

Figure 6.2 Configurations of the 4-Queens Board



(a) Intermediate Configuration (b) First Solution

step it will ask its left neighbor for the next solution and start on the bottom again. After one further unsuccessful attempt, the first solution depicted in Figure 6.2(b) is found.

Observe that in class `Queen` every method except `init` contains a polymorphic object application. The class `NullQueen` is defined local to the class `Queen` since it does not need to be accessible from outside. The method `TestOrAdvance` is declared private since it is not part of the interface to `Queens` objects. The method `init` has an optional argument `column`. If it is left out as in the last line of the program, the variable `C` is bound to the mandatory argument `n`. The number `n` and the queen's neighbor are kept in features as opposed to attributes since they do not change during the lifetime of the objects. We do not turn the immutable attribute `column` into a feature for syntactic uniformity in methods like `canAttack`. This decision arguably reveals a drawback of using different syntax for constructs that are semantically fairly closely related. All attributes and features of class `Queen` are declared free and initialized in the methods `init` and `first`. The class `Queen` is declared `final` to prevent inheritance. The only operation on the class is instance creation which we use in the last line to solve the 8-queens problem.

6.12 Discussion

In this chapter a number of advanced object-oriented programming techniques were presented on the base of the simple object system given in the previous chapter. Some of them, such as private identifiers, message patterns, and multiple inheritance, incurred an extension of the simple system. Others such as first-class classes, identifiers and messages fell out as byproducts of our overall approach. We showed that in a base language that combines lexically scoped higher-order programming and names, a powerful object system can be defined with little syntactic effort. Their straightforward semantics were presented informally. The next chapter will sketch a formal semantic foundation by reduction to the base language Small Oz.

"No, you are all wrong," said the little man meekly. "I have been making believe."
"Making believe!" cried Dorothy. "Are you not a Great Wizard?"
"Hush, my dear," he said. "Don't speak so loud, or you will be overheard—and I should be ruined. I'm supposed to be a Great Wizard."
"And aren't you?" she asked.
"Not a bit of it, my dear; I'm just a common man."
"You're more than that," said the Scarecrow, in a grieved tone; "you're a humbug."
"Exactly so!" declared the little man, rubbing his hands together as if it pleased him. "I am a humbug."

Chapter: The Discovery of Oz, the Terrible

Chapter 7

Reduction to Small Oz

Small Oz is expressive enough to support the object-oriented abstractions introduced in the previous two chapters. This comes of little surprise since Small Oz subsumes functional programming and the presented object system is sufficiently close to existing object systems for functional languages like Flavors [Moo86] and CLOS, which get by with little or no semantic extension of the base language Lisp. A secondary issue is then how the syntactic sugar for classes that we could not resist introducing can be translated to plain Small Oz.

The object-oriented abstractions are provided by a Small Oz program, called *object library*. In this chapter, we will sketch this library and the syntactic reduction of the class syntax to Small Oz. These two components together can be seen as a simple semantic foundation for Objects in Oz.

The library must be constructed in such a way that the safety conditions introduced in the previous two chapters are met. In particular, programs that use the library must be protected in the following way.

- Attributes must not be accessible from outside an object,
- private attributes, features, and methods must not be accessible outside their class definition, and
- insecure multiple inheritance must be prevented.

The user is free to define his own object-oriented abstractions, and—as we have seen in Chapter 4—Oz provides a wide variety of possibilities in this respect. The point is that these abstractions should not be allowed to mingle with code that uses the object system. For example, user programs should not allow to unsafely manipulate classes and objects provided by standard libraries such as the window system.

Program 7.1 Overall Structure of Object Library

```
declare MakeClass New ObjectApply MethodApply
   AttrAssign AttrAccess AttrExchange          in
local
   OODesc={NewName} OOAncestors={NewName} ···
in
   fun  {MakeClass    ···} ··· end
   fun  {New          ···} ··· end
   proc {ObjectApply  ···} ··· end
   proc {MethodApply  ···} ··· end
   fun  {AttrAssign   ···} ··· end
   proc {AttrAccess   ···} ··· end
   fun  {AttrExchange ···} ··· end
end
```

7.1 Class Definition

The obvious approach to reduce class definition to Small Oz is to translate it to the application of a fixed procedure. The arguments of this procedure can describe the parents, attributes, features, properties and methods of the class to be defined. For example, the class definition

```
class Account
   from BaseObject
   prop final
   attr balance:0
   feat fee:2
   meth transaction(Amount) ··· end
   meth getBalance(B) ··· end
end
```

is translated to the following application of the procedure `MakeClass`.[1]

```
Account={MakeClass
         desc(parents:    [BaseObject]
              properties: [final]
              attributes: [balance#0]
              features:   [fee#2]
              methods:    [transaction#proc {$ ···} ··· end
                            getBalance #proc {$ ···} ··· end])}
```

[1] In Oz, a syntactic convention makes sure that the user cannot accidentally redeclare such implicitly used variables and thus render parts of the object system unusable.

Program 7.2 Class Definition

```
fun {MakeClass Desc}
   case {CheckInheritance Desc}
   then aClass(OODesc:Desc
               OOAncestors:{TopologicalSort Desc.parents}
               OOId:{NewName})
   end
end
```

The procedure `MakeClass` defines how classes are represented in Small Oz. The definitions of the procedure `MakeClass` together with `New` for object creation form the core of the object library. We will introduce other procedures that define object and method application (`ObjectApply`, `MethodApply`) and attribute manipulation (`AttrAccess`, `AttrAssign`, `AttrExchange`).

The record `desc(`$\cdots$`)` defines all properties of the class, and thus we could use this record to represent the class. However, in order to meet the safety conditions, we wrap the class description in another record whose features are names. These names are bound to the local variables `OODesc` and `OOAncestors` whose scope is limited to the object library. Program 7.1 shows the overall structure of the object library.

Program 7.2 shows the definition of the procedure `MakeClass`. The only fact that is known to the user about a class `C` defined by `MakeClass` is that it is a record with the label `aClass`. Its features are not known and consequently its fields cannot be accessed; for every literal `F` that the user can get a hold of, the application `{HasFeature C F}` returns **`false`**. Furthermore, any attempt of the user to forge a class will fail, since all operations on classes results in accessing one of these fields.

The procedure `CheckInheritance` is defined locally in the object library and returns **`true`** if and only if the following conditions are met.

- None of the parents has the `final` property (Section 6.8), i.e. for no parent class *c* the list *c*`.OODesc.props` contains the atom `final`,
- the inheritance graph (Section 6.4) is not cyclic, and
- for any attribute, feature and method defined by any ancestor, there is only one class that defines it which is closest to the new class (see Section 6.4).

For convenience, we store the list of ancestors under feature `OOAncestors` in topological order with respect to the inheritance graph, beginning with the closest classes. This field will be used by object creation and object and method application.

The feature `OOId` carries the identity of the created class in the form of a new name. Thus classes resulting from different class definitions are different.

7.2 Object Creation

We are now able to define the procedure `New` that is used for object creation, shown in Program 7.3. Similar to classes, objects are represented by records.

Program 7.3 Library Procedure for Object Creation

```
fun {New Class Message}
   O={MakeInstance Class}
in
   {O Message}
   O
end
```

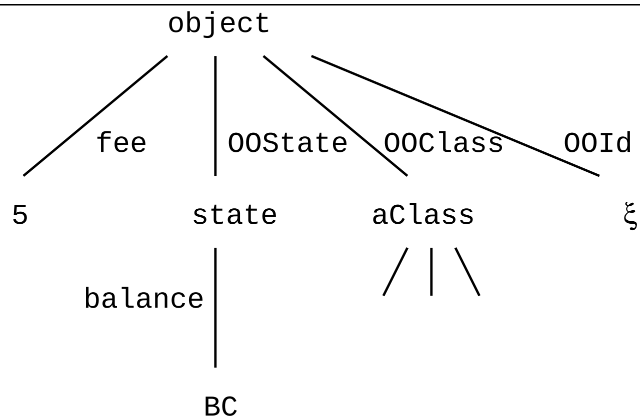
These records are constructed by the procedure `MakeInstance`.

```
fun {MakeInstance Class}
   {AdjoinList {MakeFeature Class}
               [OOState # {MakeState Class}
                OOClass # Class
                OOId    # {NewName}]}
end
```

The object features are represented by record features such that they can be accessed using field selection. An object of class *c* thus is constructed by the procedure `MakeFeatures` by adjoining all features defined by *c*'s ancestors to a record such that features of closer ancestors override features of less close ancestors. The state of the object is created by `MakeState` by adjoining all attributes using the attribute identifiers as features and cells as fields which are initialized by the proper value given in the class definition. This state record is adjoined by `MakeInstance` to the object at feature `OOState`. Furthermore, we adjoin to the object its own class at feature `OOClass`. Both `OOState` and `OOClass` are library names similar to `OODesc`. The value of a third feature `OOId` is a new name. For example, the record to which the variable `A` will be bound by `A={New Account noop}` is depicted in Figure 7.1, where the variable `BC` is bound to a cell with current value `0`.

The fact that every new instance is equipped with a new name at a feature `OOId` implements object identity in the form of token equality. Without this feature, objects without mutable state would enjoy structural equality, i.e. objects that have the same features and of which the values at corresponding features are

Figure 7.1 Structure of an Account Object



equal would be equal. The presence of attributes would even in this case enforce token equality due to the fact that every new cell comes with a new name. Structural equality of stateless objects is arguably an attractive alternative to general token equality. This is approximately the solution to object equality proposed by Baker [Bak93], who gives an excellent overview of the issue. However, we argue that encapsulation is improved with a uniform treatment of identity in mutable and immutable objects.

Free features and attributes (Section 6.2) need special treatment here. They are marked in their defining class using the name `OOFree`. The procedures `MakeFeatures` and `MakeState` create a fresh variable upon encountering `OOFree`.

After the object is created, the procedure `New` applies it to the initial message and returns it. In Section 9.9 we give an argument why we bind the output argument of `New` *after* applying the initial message.

7.3 Methods

The translation scheme for classes hinted at a translation of methods to procedures. Let us take a closer look. In our object model, the method body needs access to the current object and to the current message. Accordingly, we represent methods by procedures whose first argument represents the current object **`self`** and whose second argument the current message. Object and method application must pass the proper arguments. Object application changes **`self`** to the object being applied by passing that object as first argument to the message. Self application (as a special case of object application) and method application pass the current object as first argument to the method and thus leave self unchanged. The formal parameters of the method are bound to the fields of the message by pattern matching. Thus a method of the form

```
meth m(a:X b:Y) ··· end
```

is translated to the procedure

```
proc {$ Self Message}
   case Message
   of m(a:X b:Y)
   then ···
   end
end
```

Like for the variable `MakeClass`, we make sure that the user does not accidentally redefine the variables `Self` and `Message`. For the message patterns described in Section 6.6 this scheme is suitably modified such that a method of the form

```
meth announce(source:S
              destination:D
              weight:W<=StandardWeight ...)=Announce
   ⋮
end
```

is translated to a procedure of the form

```
proc {$ Self Announce}
   local S D W in
      S = Announce.source
      D = Announce.destination
      case {HasFeature Announce weight}
      then W = Announce.weight
      else W = StandardWeight
      end
      ⋮
   end
end
```

7.4 Attribute Manipulation

The syntax for accessing the state is translated to applications of the procedures `StateAccess`, `StateAssign` and `StateExchange` defined by the object library such that an expression @e becomes `{StateAccess Self` e`}`, a statement e_1 `<-` e_2 becomes `{StateAssign Self` e_1 e_2`}`, and an expression e_1 `<-` e_2 becomes `{StateExchange Self` e_1 e_2`}`. Note that the variable occurrences `Self` are captured by the formal argument of the closest enclosing method. The

Program 7.4 Library Procedures for State Use

```
fun {StateAccess Self Attr}
   {Access Self.OOState.Attr}
end
proc {StateAssign Self Attr NewVal}
   {Assign Self.OOState.Attr NewVal}
end
fun {StateExchange Self Attr NewVal}
   {Exchange Self.OOState.Attr $ NewVal}
end
```

keyword **`self`** is also translated to the variable `Self`. Attribute manipulation and the keyword **`self`** are not allowed outside of method bodies. Program 7.4 defines the procedures `StateAccess`, `StateAssign` and `StateExchange`.

As an example, consider the method `transaction` in Program 5.1 on page 64. According to our translation scheme, this method is translated to

```
proc {$ Self Message}
   case Message of transaction(Amount)
   then {StateAssign Self balance
         {StateAccess Self balance} + 1}
   end
end
```

The order of unnesting guarantees the usual semantics of evaluating the right hand side before carrying out the assignment.

7.5 Object and Method Application

Similar to state use, method application is implicitly applied to **`self`** and thus must occur within method bodies. A method application of the form e_1 `,` e_2 is translated to a procedure application `{MethodApply Self` e_1 e_2`}`. As for state use, the variable `Self` is captured by the formal argument of the closest surrounding method. The procedure `MethodApply` is given in Program 7.5. The auxiliary procedure `Lookup` first checks if there is a method with the identifier

Program 7.5 Library Procedure for Method Application

```
proc {MethodApply Self Class Message}
   { {Lookup Class {Label Message}}
     Self Message }
end
```

Program 7.6 Library Procedure for Object Application

```
proc {ObjectApply Object Message}
   { {Lookup Object.OOClass {Label Message}}
     Object Message }
end
```

`{Label Message}` in the method table `Class.OODesc.methods` and if local lookup fails, it searches sequentially in the method tables of the elements of `Class.OOAncestors`.

Object application is syntactically somewhat complicated due to the decision to make object application syntactically indistinguishable from procedure application. We prevent the user from directly using unary procedure application of Small Oz, but instead a new operation that we indicate by bold-face braces. Such an operation **{**`P X`**}** is translated to the application of the Small Oz application `{UnaryApply P X}` where the procedure `UnaryApply` is defined as follows.

```
proc {UnaryApply P X}
   case {HasFeature P OOClass}
   then {ObjectApply P X}
   else {P X}
   end
end
```

Note that this procedure definition must be part of the object library so that it has access to `OOClass` and that it uses Small Oz's unary procedure application. A value represents an object if and only if it has the feature `OOClass`.

The procedure `ObjectApply` is similar to `MethodApply` and given in Program 7.6. In the following, we shall use application syntax {*x* *y*} in the sense of **{***x* *y***}**.

"I have come for my brains," remarked the Scarecrow, a little uneasily.

"Oh, yes; sit down in that chair, please," replied Oz. "You must excuse me for taking your head off, but I shall have to do it in order to put your brains in their proper place."

"That's all right," said the Scarecrow. "You are quite welcome to take my head off, as long as it will be a better one when you put it on again."

So the Wizard unfastened his head and emptied out the straw. Then he entered the back room and took up a measure of bran, which he mixed with a great many pins and needles. Having shaken them together thoroughly, he filled the top of the Scarecrow's head with the mixture and stuffed the rest of the space with straw, to hold it in place.

Chapter: The Magic Art of the Great Humbug

Chapter 8

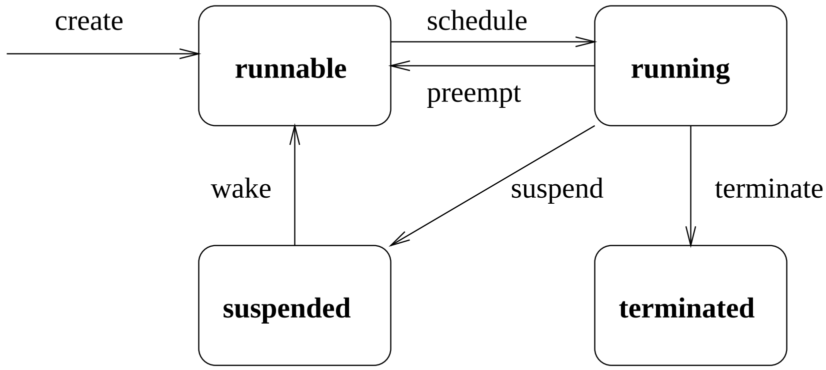
Implementation

In this chapter, we show how to integrate objects as described in the previous three chapters efficiently into an implementation of Small Oz. Any implementation of a high-level programming language has to bridge the gap between high expressivity of complex operations at the source level and low expressivity of simple operations of the target processor. Instead of compiling directly to instructions of the target processor, we use an *abstract machine*. The instructions of such an abstract machine express the basic operations of the source language, but are simple enough to allow for a straightforward and efficient interpretation by an implementation of the machine in software. Often, the even simpler approach of directly interpreting the high-level language is taken.

Compared to interpretation of high-level source code, abstract machines provide a clear efficiency advantage. Compared to compilation to native code, abstract machines simplify implementation and increase portability. Furthermore—as this chapter itself shows—abstract machines support extensibility towards language extensions and their concomitant optimizations.

Mehl, Scheidhauer and Schulte [MSS95] describe an abstract machine called AMOZ of a previous version of Oz. Most features of AMOZ carry over to the implementation of Small Oz. The first three sections introduce the aspects of AMOZ that are needed for the rest of the chapter. Section 8.1 shows how AMOZ handles threads, Section 8.2 shows how the data structures of Small Oz are represented, and Section 8.3 shows how the operational semantics of Oz is mapped to AMOZ. This presentation puts us in the position to explain in Section 8.4 the consequences of the design of our object system from the implementation perspective; we identify several critical issues. In Section 8.5, we address these issues and describe a realistic implementation of objects in Oz. We describe how a number of implementation techniques for object-oriented languages can be integrated in AMOZ and show that AMOZ can be adapted to efficiently support first-class messages. In Section 8.6, we evaluate the performance of the resulting implementation and

Figure 8.1 Life Cycle of Threads



compare it with object systems of other programming languages.

8.1 Threads

The active entities in AMOZ are called *workers*. At each point in time, each worker serves one single thread by reducing the statements on its stack. This thread we call *running*; it is the worker's *current thread*. The worker monitors its reduction and preempts the thread after spending a certain amount of runtime on it. Preemption makes running threads *runnable*. If a worker encounters a synchronized statement whose synchronization condition is not met yet, the thread becomes *suspended*. Upon preemption and suspension of a thread, the worker picks up another runnable thread and makes it running. When the synchronization condition of a suspended thread becomes met, the thread is *awoken* and becomes runnable. A thread whose stack is empty becomes *terminated* and subject to garbage collection. Figure 8.1 depicts the life cycle of a thread.

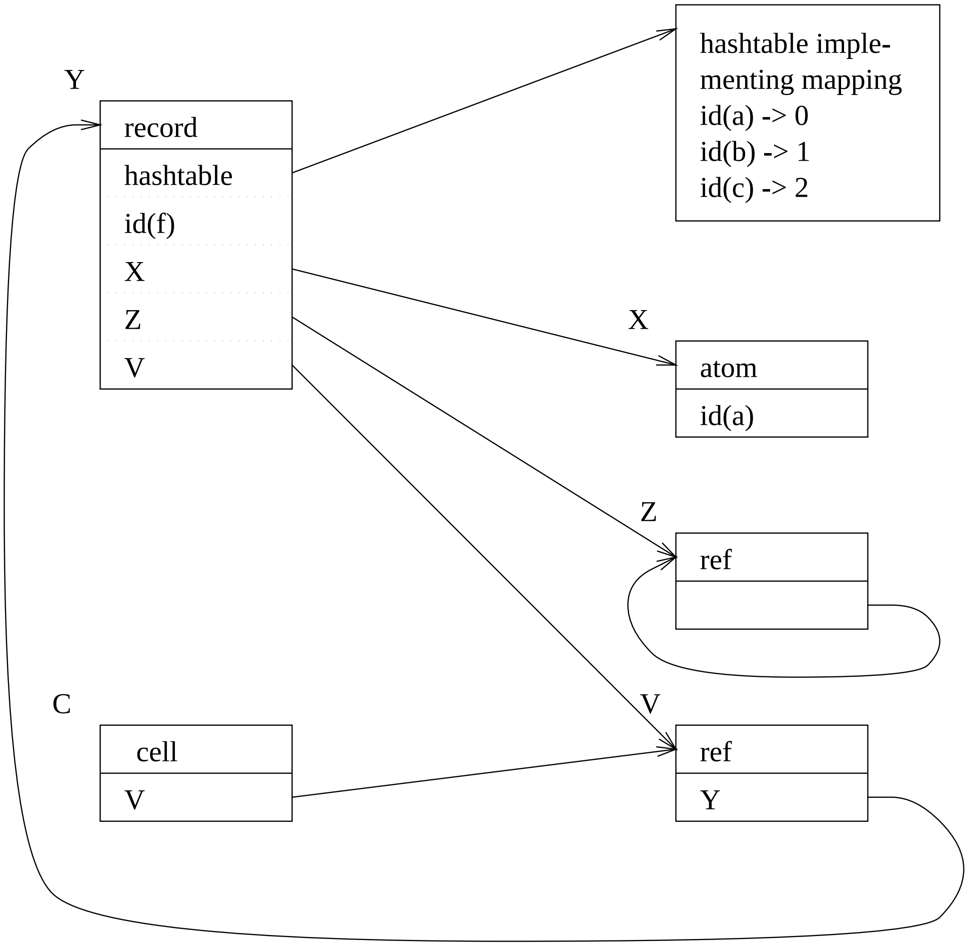
In an implementation with only one worker, interleaving semantics defined in Section 3.1.2 can be guaranteed by allowing preemption and suspension only *between* reductions of statements. In a parallel implementation, the workers must be carefully designed to enforce interleaving semantics; we are not going to address these issues here and instead silently assume interleaving semantics for reduction. The design of a parallel implementation is described by Popov [Pop97].

8.2 Representing the Constraint Store

Recall that basic constraints in Small Oz have one of the following forms:

$$x = y, \quad x = c, \quad x = l(c_1 : x_1 \cdots c_n : x_n)$$

Figure 8.2 Nodes on the Heap



The tell statement can add consistent basic constraints to the constraint store and synchronized statements must wait until enough information arrives in the store. The basic constraints and the operations on the store are designed such that a variable-centered representation is possible. In such a representation, the store does not represent basic constraints but rather nodes containing constants, records and references to other nodes. This representation of the store we call *heap*. Variables are represented by addresses of nodes on the heap. Figure 8.2 shows a heap segment after executing the program

```
X=a Y=f(a:X b:Z c:V) V=Y
{NewCell V C}
```

Each node contains a tag with its type as first entry.[1] The variable `X`, which

[1]In this presentation, we use tagged nodes. In practice, using a whole word for such a tag can

is bound to the atom `a`, is represented by the address of the corresponding atom node. We can assume that the compiler generates for each atom *at* a word $id(at)$ that uniquely identifies it. The atom node contains this identifier as its second entry. We can arrange for names to get unique identifiers different from atom identifiers such that we know whether a given identifier represents an atom or a name. Name nodes contain a name identifier as second entry and integer nodes their integer value.[2] Nodes corresponding to simple values we call *simple nodes*.

The variable `Y` is represented by the address of a record node. For fast field selection, record nodes contain as second entry the address of a hash table that maps feature identifiers to indices. Such a hashtable we call *index table*. We can arrange that all records of a given arity share the same index table by using a hashtable of index tables upon record creation [RMS96]. This optimization is crucial for our object system, since—as we will see—the attributes (and free features) of an object are kept in a record and all such records for instances of a given class have the same arity. The remaining three entries of the record represent its fields that can be accessed through the index resulting from hashing with an offset of three.

The variable `V` is bound to the variable `Y`. It is represented by the address of a reference node that contains the address of `Y` as second entry. The field of `Y` at feature `b` represents the free variable `V` in the form of a self-referring reference node. The process of following chains of reference nodes, until such a free variable or a node with a tag other than `ref` is encountered, is called *dereferencing* a variable.

Instead of having a separate cell store, we add a new kind of node for cells as shown in Figure 8.2. Their tag is `cell` and their only field contains the address of the node representing the current content. Exchange simply writes a new address in the field and returns the old content of the field.

8.3 The Abstract Machine

A worker reduces the statements on the stack of its current thread. A central role is played by the application statement. To implement procedure application, Small Oz prescribes to copy procedure bodies from the procedure store onto the heap, substituting actual for formal arguments. To avoid the copying of any code, we introduce the usual indirection in the relationship between variables in the code and the heap. Variables in the code do not directly refer to nodes on the heap, but are represented by indices into an *environment* that in turn holds references to

be avoided by adding the type information to references to nodes (tagged references), as is done in DFKI Oz 2.0.

[2]In this implementation, integers are limited to word size. DFKI Oz provides arbitrarily sized integers using an indirection.

the heap. Thus by using different environments, several invocations of the same procedure (in the same or in different threads) can share the same code. Each thread has access to its *current* environment E. We refer to such thread-specific information as *registers*; thus E represents the thread's *environment register*.[3] We refer to the slot in the current environment at index n by $E[n]$.

Oz code is compiled into a sequence of abstract machine instructions. The compiler writes *abstract machine code* in a memory segment called *code area*. Threads have another register, called *program counter* or shorter *pc*, that refers to the next instruction in the code area to be executed. Consider for example the synchronized Small Oz statement `X = R.A`. This instruction may be compiled to a machine instruction of the form

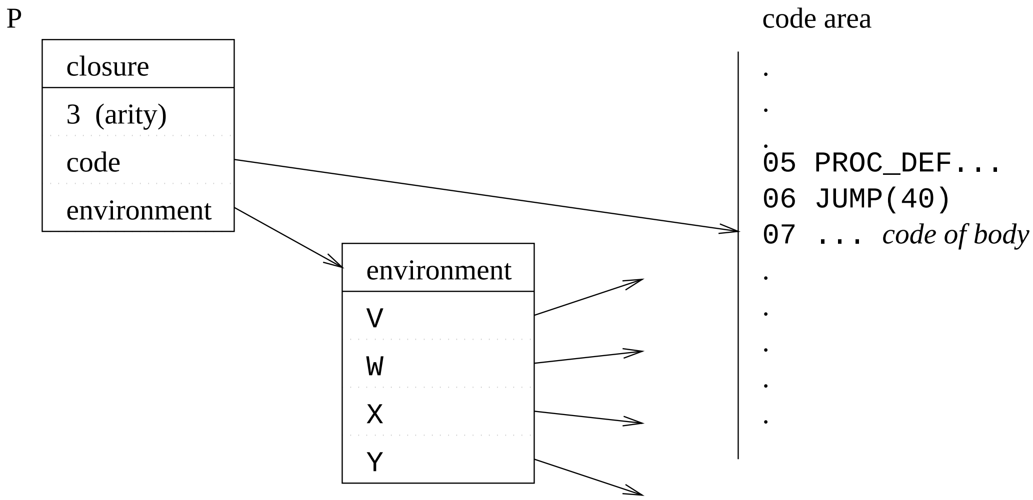
```
02 ...
03 SELECT(2,3,1)      % field selection using environment slots
04 ...
```

Here, the compiler decided to use the environment slots `2`, `3`, and `1` for the variables `R`, `A` and `X`, respectively. When the *pc* of the current thread of a worker is set to `03`, this instruction is executed. First, the worker dereferences $E[2]$ and $E[3]$. If $E[2]$ or $E[3]$ refer to an unbound variable, the current thread is suspended. To suspend the thread, the worker first makes sure that binding the variable to a value checks if the thread can be awoken (details are not important here), and then picks up another runnable thread. If dereferencing $E[2]$ results in a record node r and $E[3]$ in a simple node l, execution can continue; otherwise an exception is raised. We say that the worker performs *synchronized dereferencing* of $E[2]$ *for* a record node r and of $E[3]$ *for* a simple node l. The record node r is accessed by hashing for an index in its index table using $id(l)$ resulting in an address v. In order to tell the equality of the corresponding variable with `X`, the worker dereferences v and $E[1]$, and checks if their equality is consistent with the current store. If this is not the case, an exception is raised. If it is the case, the worker modifies them such that the heap represents the equality of both variables. This process is known as unification in logic programming. An implementation in the framework of Oz is given in [MSS95]. After that, *pc* is incremented by the size of the instruction, and the worker continues with the next instruction.

Since procedures are values, they must be represented on the heap. Instead of representing them by names as in Small Oz, we introduce a new type of node, called *closure*. Procedure definitions contain code to be executed upon application. Thus the closure must have a reference to the corresponding code in the code area. Variables occurring in the body of procedure definition are either bound in

[3] A sequential implementation can optimize registers such that they are kept in global variables of the abstract machine, which saves the indirection through the thread for accessing them.

Figure 8.3 A Closure in the Store



the body (local variables), or bound by a formal argument of the procedure (argument variables), or not bound in the procedure definition (free variables). The free variables are statically scoped and thus the closure must have a reference to a *closure environment* that maps indices as used in the body of the procedure to the addresses of nodes of the free variables on the heap. Threads have a register *F* that holds a reference to their current closure environment. This closure environment implements lexical scoping of non-local variables occurring in procedure definition.

The first instruction in the body of the procedure allocates a new environment for local variables. To check whether the arities of procedure application and definition match, the closure contains the arity of the definition. A closure for a procedure `P` defined by

```
proc {P X Y Z} ··· end
```

whose body refers to four non-local variables is depicted in Figure 8.3.

The above procedure definition is translated to the following machine code.

```
05 PROC_DEF(n,07,3,[v,w,x,y])      % create closure node
06 JUMP(40)
07 ···                             % code of body
⋮
40 ···                             % other code
⋮
```

Figure 8.4 Machine Code implementing Procedure Application

```
40 MOVE_TO_A(6,1)    % A[1] ← E[6]
41 MOVE_TO_A(7,2)    % A[2] ← E[7]
42 MOVE_TO_A(8,3)    % A[3] ← E[8]
43 APPLY(5,3)        % apply P to 3 arguments
44 ...
```

The instruction PROC_DEF creates a closure node on the heap using the code address 07, the arity 3, and tells the equality of this node with $E[n]$. The created closure has a reference to a closure environment containing the variables $E[v]$, $E[w]$, $E[x]$, $E[y]$.

In order to provide the procedure body with the actual arguments, the arguments of the application are written to a new set of registers called *argument registers*, and retrieved from there by the body of the procedure. We refer to the register for the *n*-th argument by $A[n]$. Consider a procedure application in Small Oz of the form

```
{P X1 X2 X3}
```

Let us assume the compiler decided to use slots 5, 6, 7, 8 in the environment for the variables P, X1, X2 and X3, respectively. Then the application is translated to the code segment given in Figure 8.4.

The instructions MOVE_TO_A move the arguments of the procedure from the environment to the argument registers. For executing the instruction APPLY, the worker performs four tasks. The first task consists of synchronized dereferencing of the first argument $E[5]$ of APPLY for a closure *p* with arity 3. Secondly, the address of the following instruction ($pc + 1 = 44$) is pushed on the stack of the current thread. Thus the stack of statements in Small Oz is implemented as a stack of code addresses. Every procedure body is terminated by a RETURN instruction that pops the return address from the stack, sets *pc* to this address and the worker continues with executing the procedure body.[4] Thirdly, the worker must set the *F* register to the closure environment of *p*. Finally, *pc* is set to the code address in *p* and the worker starts executing the procedure body. For pushing the return address and setting *pc* to the code address of a procedure *p*, we say the worker *jumps to p*.

Procedures in Oz are first-class values. Often, however, this expressivity is not used and the procedure to be called is known at compile time. In this case the worker can do without synchronized dereferencing of the procedure and jump directly to the closure for which the compiler allocated a heap address. Thus, if P is declared outside of a procedure, and if the compiler knows that P is a ternary

[4]This is also a convenient point to check if the thread must be preempted.

procedure, it can allocate a heap address p for it and compile the application to an instruction of the form

44 `APPLY_STATIC(`p`)` `%` *jump to p.*

The executing worker can jump directly to p without dereferencing through an environment register. Note that this instruction must be updated appropriately, when garbage collection changes the address of the closure.

8.4 Implementation Issues

8.4.1 Memory Consumption

A direct implementation according to Chapter 7 results in representing an object with f features and a attributes on the heap by a record node (one word each for tag, label and index table address) with the following further entries (one word each):

- one field entry for each of the f features,
- one field entry at feature `OOState` for the state. This field refers to a record with a fields ($3+a$ words), each referring to a cell ($2a$ words), and
- one field entry at feature `OOClass` referring to the class of the object.

This amounts to a memory consumption of

$$3+f+1+3+a+2a+1 = f+3a+8 \text{ words}$$

per object, assuming that the index tables of all instances are shared.[5] Observe in particular that three words are needed for each attribute which is clearly suboptimal.

8.4.2 Messages

Object and method application are performed on messages which are records whose label is used for method lookup and whose fields represent the method arguments. We already saw in Section 4.2 that first-class messages are convenient from the programmer's point of view. However, their naive implementation results in allocating a record node on the heap for every message with at least one

[5] The DFKI Oz implementation makes sure that all records of a given arity share a single index table.

field. The arguments are put in the fields of the new record node. The method body accesses the record node to extract the method arguments. Considering the fact that object and method application are the core operations in object-oriented programs, it is clearly not acceptable to allocate memory for each invocation and access the heap twice for each argument.

8.4.3 Self

Self is implemented as an additional argument to each method. Furthermore, attribute manipulation has self as argument. Thus for every object/method application and attribute manipulation, self must be loaded into and retrieved from an argument register. This overhead should be reduced.

8.4.4 Late and Early Binding

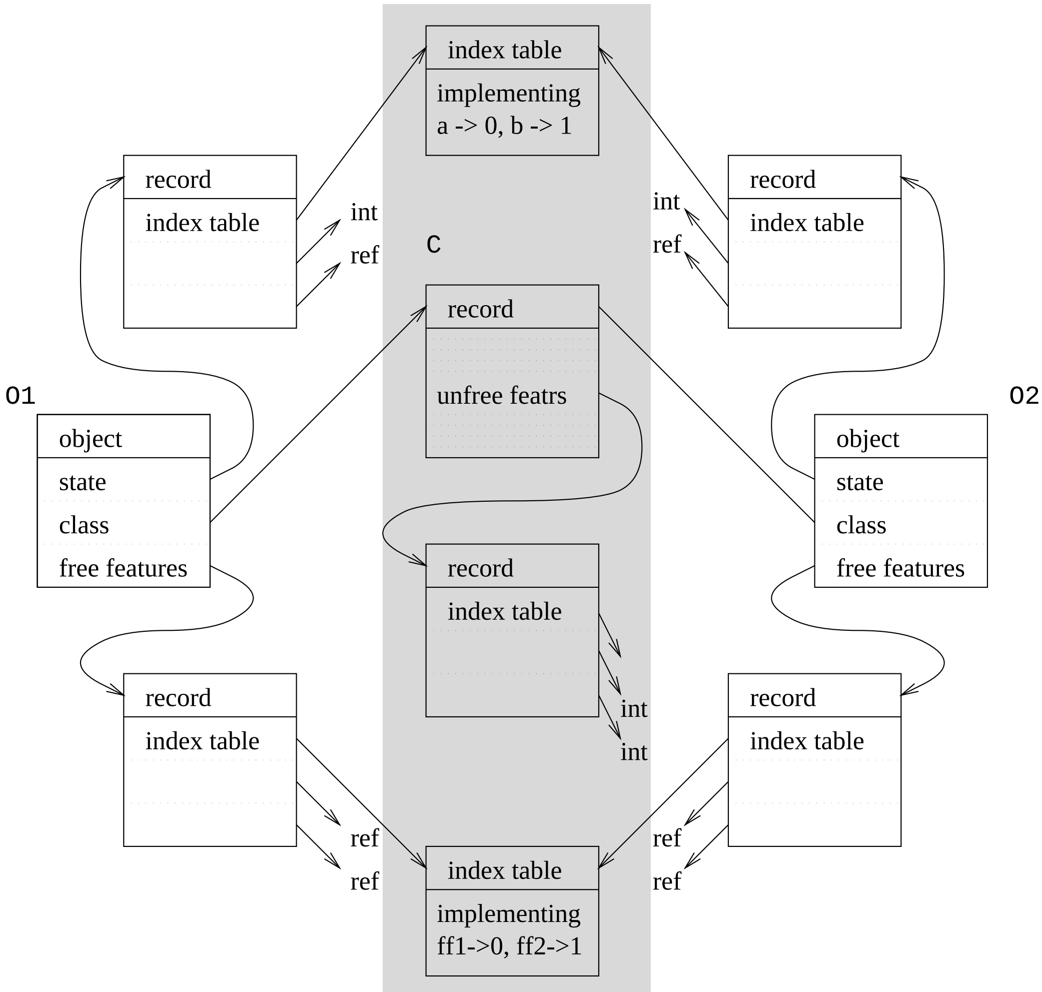
The mapping from method identifiers to methods in object and method application is done by method lookup in the appropriate class (procedure `Lookup` in Section 7.5). For simplicity, we encoded method tables as lists. A first improvement is to represent method tables by records. In that case method lookup creates runtime costs practically linear in the inheritance distance from the class where lookup starts to the class that holds the method. Even if the method is found in the first class, the cost of hashing in the method table is significant compared to ordinary procedure application.

For method application, the situation is particularly unsatisfactory. If the class `C` and label `L` of a method application `C,L(···)` are statically known, so is the method that will be called. We want to make use of this information and reduce the cost in this case to the cost of procedure application.

8.5 A Realistic Implementation

We address these issues by first describing a better memory layout for objects (Section 8.5.1) and the instructions that operate on this layout. These instructions form the base for subsequent optimizations such as a technique to avoid allocating records for messages in many cases (Section 8.5.2), optimized treatment of self (Section 8.5.3), and optimized late binding (Section 8.5.4). The latter three optimizations are orthogonal to each other; for simplicity of presentation we describe them independently, rather than combining them with each other as is done in a real implementation.

Figure 8.5 Memory Layout of Two Objects of the Same Class



8.5.1 Memory Layout

We introduce a new type of node for objects and modify the representation of classes. Object nodes contain the address of a record containing the object's free features, the address of a record containing its attributes, and the address of its class. Figure 8.5 depicts the memory layout of two objects, `O1` and `O2`, of the same class `C` whose definition includes **`attr`** `a:10 b` **`feat`** `ff1 ff2 uf1:50 uf2:100`.

The attribute record contains references to the values of the attributes rather than to cells holding the values. Assignment and exchange destructively modify this record. Note that this optimization is not observable since access, assignment and exchange are the only operations that have access to the fields of the state record. In the feature record, we only keep the free features. The unfree features are shared among all instances and can be kept in the class. For this purpose, each class provides a field at feature `OOUnfree` containing the record of unfree features.

Thus we end up with a space consumption of four words per object, three words each for free feature and attribute record, and one word for each of its *a* attributes and its *ff* free features, resulting in

$$4+3+3+a+ff = a+ff+10 \text{ words}$$

per object.[6]

Sharing of unfree features is crucial for some applications. In fact, it was adopted as a reaction to unacceptable memory consumption in the Oz Explorer [Sch97] which makes heavy use of unfree features.

Obviously, with this representation of objects, we cannot use the implementation of the object-related operations object creation, object/method application, attribute manipulation and field selection given in Chapter 7. Instead, we compile these operations to special purpose machine instructions:

Object Creation. An application of the object library procedure `MakeInstance` (see Program 7.2) of the form `{MakeInstance Class Object}` is translated to a machine instruction

MAKEINSTANCE(n,m)

where n and m are the environment slots allocated to `Class` and `Object`, respectively. This instruction creates a new object node with reference to properly initialized free feature and state records and to the class $E[n]$, and unifies the object node with $E[m]$.

Object Application. Due to the overloading of the syntax for application with one argument for both procedure and object application (see Section 7.5), we need to consider both procedures and objects as functor of a statement `{X Y}`. The corresponding machine instruction is

APPLY1(n)

The variable `X` resides in $E[n]$, and `Y` in $A[1]$. This instruction performs synchronized dereferencing of $E[n]$ for a node q and dispatches on the tag of q. If q is a closure, the instruction behaves like APPLY(n,1). If q is an

[6] We could further decrease the constant by integrating either the free feature record or the attribute record in the object node at the expense of implementing an adapted version of record-like lookup for the object node.

object node, synchronized dereferencing of $A[1]$ for a record or literal node is performed. Using the label of this node, method lookup is performed in the class of q, resulting in a closure p. Then $A[1]$ (the message) is moved to $A[2]$ and $E[n]$ (the new **`self`**) to $A[1]$, and the worker jumps to the method p. Recall from the previous chapter that methods are procedures that expect the new self as first and the message as second argument. Note that this instruction corresponds to the procedure `UnaryApply` in Section 7.5.

Method Application. Method application need not be changed. However, we introduce the following machine instruction for the Small Oz statement `Class,Message` so that we can modify it later.

APPLY_METHOD(n)

The variable `Class` resides in $E[n]$, `Self` in $A[1]$ and `Message` in $A[2]$. The worker performs synchronized dereferencing of $E[n]$ for a record node c and of $A[2]$ for a record or literal node m. Using the label of this node, method lookup is performed in c, resulting in a closure p, and the worker jumps to p.

Attribute Manipulation. Attribute access `X = @A`, assignment `A <- X` and exchange `X = A <- Y` is translated to the machine instructions

ACCESS(n)
ASSIGN(n)
EXCHANGE(n)

where `A` resides in $E[n]$, `Self` in $A[1]$, `X` in $A[2]$ and `Y` in $A[3]$ (in case of exchange). The executing worker performs synchronized dereferencing of $E[n]$ for a simple node l. The instruction `ACCESS` performs field selection at feature l on the attribute record of the object node referenced by $A[1]$, similar to `SELECT` on page 105. The instruction `ASSIGN` writes the address in $A[2]$ destructively into the field at feature l of the attribute record. This is safe since `ACCESS`, `ASSIGN` and `EXCHANGE` are the only operations that have access to this record. Destructive record update allows us to do without a cell for each attribute and save the indirection through the cell. The instruction `EXCHANGE` atomically performs field selection like `ACCESS` and destructive field update like `ASSIGN`.

Field Selection. To accommodate field selection of objects we need to modify the machine instructions for field selection introduced in Section 8.3. Consider an instruction

SELECT(n,m,l)

> After synchronized dereferencing of $E[n]$ for a node q, the worker dispatches on the tag of q. If q is a record node, the worker proceeds as in Section 8.3, and if q is an object node, it selects the free feature record of q at the literal referenced by $E[m]$. If selection is successful, the address of the result is entered in $E[l]$. If this selection is not successful, the worker instead performs selection on the field of the class of q at feature `OOUnfree`.

8.5.2 Messages

Messages are used as first-class values if a variable is used as message in object or method application or if the method body requires a reference to the message by using the method pattern *m*=*x* (see Section 7.3). All other cases of object and method application we would like to optimize such that no message record is created on the heap. For this purpose, object/method application as well as method definition are modified.

The idea of the optimization is to delay the creation of the message on the heap and pass the method arguments in argument registers, if the message is statically given in object/method application. Hopefully, the method does not refer to the message as such but only to its fields. If it does refer to the message, there is still time to create the message on the heap; if not, the method arguments are retrieved from the argument registers as in procedure application.

Method Definition

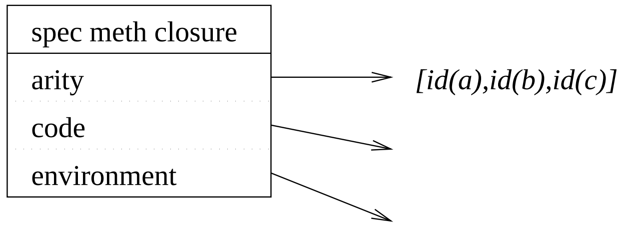
Recall from Chapter 7 that methods are represented as binary procedures in a method table of their defining class. We call the corresponding closures *standard method closures*. We introduce a second method table that contains for some methods a *special method closure*. Methods qualify for such a node if the features occurring in the method pattern are known to the compiler and if their default values fulfill certain conditions (see below). For methods that do not qualify, the special method table contains the name `OONoSpecialMethod` under the corresponding method label. Let us first consider methods with only atoms as features, no defaults or ellipses in their message pattern and which do not refer to the message as first-class value. Their special method closures are variants of (procedure) closures that hold as arity, instead of an integer, an ordered list of identifiers of their features. Thus, a method definition of the form

```
meth m(a:X b:Y c:Z) ··· end
```

results in a special method closure depicted in Figure 8.6.

The bodies of special methods expect the actual arguments corresponding to the fields of the message pattern in the same order as the arity in the argument

Figure 8.6 A Special Method Closure



registers. We avoid duplicating the body of methods in the code area by making the standard method call the special method.

Object/Method Application

For this optimization, an object/method application qualifies, if the label and arity of the message is known to the compiler. This is the case if the message is given in record syntax as argument to the object, and the features are given as atoms or integers (we discuss names as features below). Consider an object application of the form

```
{Object f(a:X1 b:X2 c:X3)}
...
```

Let us assume the compiler decided to use the environment slots 4, 5, 6 and 7 for the variables `Object`, `X1`, `X2` and `X3`, respectively, and $id(\texttt{a}) = 100$, $id(\texttt{b}) = 101$, $id(\texttt{c}) = 102$, and $id(\texttt{f}) = 105$. Then the object application is translated to

```
00 MOVE_TO_A(4,1)                    % A[1] ← E[4]
01 MOVE_TO_A(5,2)                    % A[2] ← E[5]
02 MOVE_TO_A(6,3)                    % A[3] ← E[6]
03 MOVE_TO_A(7,4)                    % A[4] ← E[7]
04 APPLY_OBJECT(4,105,[100,101,102]) % apply Object with
05 ...                               % label f and
                                     % arity [a b c]
```

Like `APPLY1`, execution of `APPLY_OBJECT(`n`,`m`,`a`)` dispatches on the tag of the node q referenced by $E[n]$. If q is a closure, the worker must construct a record node on the heap using label m, arity a and the first $length(l)$ argument registers as fields. It puts a reference to this node in $A[1]$ and continues like for `APPLY(`n`,1)`. If q is an object node, the worker looks for a method with label m in the class of q and its ancestors. If a special method sm is found in the ancestor class c, its arity is compared with a. If they are equal, the worker jumps to sm. If not, or if special method lookup encounters the name `OONoSpecialMeth`, it looks up the standard method *sm* in *c* with label *m*, creates a record node *r* on the heap as above, puts the address of *r* in $A[2]$, and jumps to the standard method of *c* with label *m*.

The instruction `APPLY_METHOD` is modified similarly.

Methods with Ellipses and Defaults

Recall that method patterns with ellipses allow the actual message to have more features than mentioned in the pattern (see Section 7.3). Special method closures for methods with ellipses contain a flag that indicates this. The instructions `APPLY_OBJECT` and `APPLY_METHOD` are modified such that if this flag is set, the comparison of arities is able to skip features. Along the way, the content of the argument registers is moved up to fill skipped fields so that they are where the body of the special methods expects them to be.

Recall that a feature with default in method patterns indicates that the actual message need not have this feature, and if it does not, the default expression is evaluated instead, and the result is bound to the formal argument. Correspondingly, during comparison of the arities the worker needs to put the default value in the corresponding argument register if it is left out in the actual message. This is the easiest to implement if the default expression does not incur any computation and only uses variables not captured within the method. In this case, we can pull the default expression out of the method pattern and construct a table of defaults corresponding to the table of special methods.[7]

Names as Features of Messages

Since the optimization relies on extracting both the arity of messages in object/method application and the arity of message patterns in method definition, it can only be applied when the arity is known at compile time. Thus variables as features are only allowed if the compiler knows their value. Fortunately, this is the case when names are used as features, if they are represented by variables that are declared outside of procedures and bound to names before they are used. Thus, we can use names to implement *private message features* similar to private messages, attributes and object features and still enjoy optimized compilation as described in this section.

[7]DFKI Oz optimizes methods with defaults if the default is either ground or consists of `_` in which case a special flag is entered in the default table. The default `<=_` seems to be a frequent idiom for optional output arguments.

Object Creation Revisited

Recall that the procedure `New` in Program 7.3 uses the initial message as first-class value. In order to use this optimization for this message, we *inline* any application of the procedure `New` (that the compiler knows of), i.e. we replace it statically by its body. Then the initial message becomes the argument of an object application and the optimization can work as usual.

Summary

We managed to avoid the creation of record nodes for messages in the object/method application, when the following conditions are met:

- The structure of the message record (label and arity) is statically known.
- The method definition does not use the message as first class value.
- The default expression does not incur any computation and only uses variables not captured within the method.

The cost of this optimization is one additional closure on the heap per method definition and one extra table for special methods per class. The bodies of standard and special methods in the code area are shared and thus consume little extra space.

8.5.3 Self

Since self is used implicitly in method application and attribute manipulation, passing it in an argument register to these operations turns out to have significant cost. In practice, we observe that self is accessed much more often than it is changed. Thus, we introduce a new register, called *self register*, into which self is put by object application and from which it is retrieved by the other operations. Before setting the self register, object application saves its current content to be reinstalled after returning from the body of the corresponding method. This is done most conveniently by pushing the current self register (properly marked as such)—in addition to $pc+1$—on the stack. The corresponding `RETURN` instruction will reinstall self.[8] The instructions `APPLY_METHOD`, `ACCESS` and

[8] Note that Oz's exception mechanism provides a way to leave an object application before the corresponding `RETURN` instruction is executed. An exception handler can be pushed on the stack. Raising an exception by executing a corresponding statement results in looking for an appropriate handle down the stack. This has the consequence that the worker must reinstall the values for self along the way whenever it encounters a saved self register. This way, the handler always uses self as defined by lexical scoping.

`ASSIGN` are modified such that they retrieve self not from $A[1]$ but from the self register.

A complication is posed by the possibility to use self (implicitly or explicitly) within procedure definitions in methods. The semantics of course prescribe static binding (as exploited in the programming technique described in Section 6.7). Without precaution, this optimization implements dynamic binding of self, since the self register of the caller would be used! Thus, the compiler must access the self register before the procedure definition with a special machine instruction `GET_SELF`, bind it to a local variable, and set the self register with another machine instruction `SET_SELF` in the body of the procedure. The instruction `SET_SELF` pushes the current self on the stack like object application. As an example, consider the following method.

```
meth m(P)
   ⋮
   proc {P X}
      a <- 1
   end
   ⋮
end
```

This method is translated to the following code.

```
43 ...
44 GET_SELF(4)
45 PROC_DEF(n,47,1,[4,···])
46 JUMP(60)
47 SET_SELF(1)      % set self to slot 1 of closure environment
48 ...              % code of P
49 ...
```

Note that with this technique we delegate the static binding of self to static binding of free variables of procedures.

8.5.4 Late and Early Binding

A naive implementation, in which object application searches the ancestor hierarchy of the receiver's class for a matching method, incurs a dramatic overhead. Driesen reports that modern implementations of object-oriented languages are able to reduce the time spent on handling messages to about 20% of the total runtime [Dri93a]. In this section, we show how standard implementation techniques for dynamic binding can be applied in our setting. The first technique employs the idea of memoization in that the results of previous method lookups are stored in descendant classes. The second technique reduces the cost of hashing by storing lookup results in the machine code and is also used for attribute manipulation and feature access.

Lookup Caches

If method lookup with a message label *m* in a class *c* fails, the result of the lookup in the ancestor classes is stored in the method table of *c* so that subsequent requests to look up a method for *m* in *c* can use the stored method. This technique is called *lookup cache* and is reported to improve the overall performance of an implementation of the pure object-oriented language Smalltalk by as much as 37% [UP83]. To implement the technique, we use dynamic hash tables for method tables as opposed to the static hashtables of records.[9]

In previous versions of Oz, complete method tables were used [SHW95], i.e. during class creation, a table of all methods was computed by adjunction of all method tables of inherited classes. With complete method tables, method lookup consists of a single hashing operation. We experienced a considerable memory consumption for complete method tables for medium-sized object-oriented programs. For example, the complete method tables of the standard library of DFKI Oz 2.0 consumed about 150 kBytes of live memory, with negative impact also on the runtime of garbage collection. As a reaction, we implemented lookup caching. Another possibility would have been to investigate memory and runtime efficient implementation techniques for complete method tables as presented by Driesen [Dri93b] and Vitek and Horspool [VH94].

Inline Caches

The key to this optimization is the observation that for a particular instruction for object/method application in the code the *object* to which the instruction is applied may change frequently, but the *class* of these objects changes much less frequently. It is this class that determines which method is applied. Thus the worker remembers for each call of a machine instruction the class *c* in which lookup is performed and the address *m* of the resulting method closure on the heap. If the next execution of the same instruction (by the same or another worker) uses the same class *c* for lookup as the previous one, the worker directly jumps to *m*. It needs to perform a new lookup only if the class is different. The most convenient

[9] In Oz, dynamic hashtables are provided by dictionaries. With dictionaries, an implementation of this lookup mechanism in Oz becomes practical. In fact, the current implementation accesses the local dictionary directly from the machine instructions `APPLY_METHOD` and `APPLY_OBJECT`, but upon failure falls back on an Oz procedure that implements lookup including filling the lookup caches.

place to store c and m is the instruction itself. Therefore, this technique is called *inline caching*. Inline caching has been reported to improve the performance of an implementation of Smalltalk-80 by 33% [Ung86].

We implement inline caching by adding two words c and s to the instruction for object application (method application similar), resulting in the format

`APPLY_OBJECT(`n`,`m`,`l`,`c`,`s`)`

The word c is called *class cache* and is initialized with a value different from any address of nodes on the heap. For execution, the worker retrieves the class c' of the object referred to by $E[n]$ and compares c' with c. If they are different, it overwrites c in the instruction by c', performs the usual method lookup resulting in a procedure node s', overwrites s by s' in the instruction and jumps to s'. If c' is equal to c, we can do without the lookup and directly jump to s. Note that garbage collection invalidates the content of inline caches and thus must reset the class cache to a value different from any heap address.

We can do even better for a special case of method application. Method application in which the variable referring to the class is declared outside of procedure definition implements static method binding. Such a method application will always call the same method. We translate such a method application to an instruction of the form

`APPLY_METHOD_STATIC(`n`)`

The first worker that executes this instruction looks up the appropriate method node m in the class $E[n]$, and can safely replace the instruction by

`APPLY_STATIC(`m`)`

Recall that we introduced `APPLY_STATIC` in Section 8.3 as the optimized version of procedure call with statically known closure.

The analogous situation for object application is that the variable referring to the object is declared outside of procedure definition. This case however occurs rarely and is not worth introducing a special purpose machine instruction.

Note that lookup caches complement inline caches in that they speed up the lookup time incurred by inline cache misses.

Inline caches avoid hashing in method tables. We use the same idea for attribute manipulation and object feature access. The corresponding instructions `ACCESS`, `ASSIGN` and `SELECT` get two more words that refer to the class and the offset by which the corresponding attribute (object feature) can be reached in the state record node (free feature record).

8.6 Performance Evaluation

In this section, we evaluate the performance of objects in DFKI Oz 2.0 [ST97] in comparison to a number of state-of-the-art object-oriented programming systems. DFKI Oz 2.0 is an implementation of Oz 2 of which Small Oz is—with slight variations—a sub-language. DFKI Oz 2.0 is based on an abstract machine along the lines of AMOZ, realized by a sequential (one worker) byte-code emulator. All optimization techniques presented in the previous section have been integrated in DFKI Oz 2.0.

Comparing the performance of implementations of different programming languages is a rather elusive topic, because different languages encourage the use of different programming idioms to implement the same algorithm. For example, pure object-oriented languages use object application for every operation even on primitive data structures like integers. Non-pure object-oriented languages such as C++ and Oz provide objects as one of many available data structures. A fair performance comparison will code the same algorithm in the easiest possible way in the respective language. For Oz this will result in practice in programs in which the non-object oriented features of Oz dominate the runtime opening discussions about the "purity" of the language in terms of object-oriented programming and the fairness of comparison to other languages. Our aim here is to concentrate on the central operations of object-oriented programming such as attribute manipulation and late binding.

However, the attempt to measure the performance of individual constructs is difficult since techniques like procedure inlining and caches lead to dramatically different behaviour. Instead, we use the algorithms "Sieve of Eratosthenes" and "N-Queens" described in Sections 5.5 and 6.11 for comparative performance case studies. In both studies, the object-oriented programming idioms make up a considerable part of computation. In particular, late binding and state use are dominant. Each study provides a different mix of these idioms. We do not claim that these programs are in any way representative for the use of object-oriented constructs in the respective languages, so the result can only give a rough idea on the performance, rather than exact results. We implement the algorithms in the object-oriented languages and systems given in Table 8.1. The selection was driven by the wish to cover a wide variety of languages (imperative, functional, logic) and state-of-the-art systems. We emphasize that we are measuring *systems* and not *languages* and that we only measured the systems on one single platform.

All measurements have been done on a Sun Sparc 20 (712/128MB) running Solaris 2.5.1 under low utilization. Occasionally operating system activity disturbes the runtime (memory cache misses etc), resulting in an obviously exceptionally large runtime. The reported measurements are the arithmetic mean of 5 "undisturbed" runs in seconds.

Table 8.1 Languages and Systems used for Performance Comparison

Language	System	Implementation Technique	Abbreviation
C++	GNU gcc 2.7	native code	C++N
CLOS	Allegro CL 4.3	native code	CLOSN
Java	JOLT Kaffe 0.82	JIT native code	JavaN
Java	SUN JDK 1.0	emulated byte code	JavaE
Objective Caml	Objective Caml 1.03	native code	OCamlN
Objective Caml	Objective Caml 1.03	emulated byte code	OCamlE
Oz	DFKI Oz 2.0	emulated byte code	OzE
Smalltalk	VisualWorks 2.0	JIT native code	SmalltalkN
SICStus Objects	SICStus 3.0	native code	PrologN
SICStus Objects	SICStus 3.0	emulated byte code	PrologE

When taking undisturbed runs, the Coefficient Of Deviation (COV = standard deviation /arithmetic mean) was always below 2%, which justifies the sample size of 5. All source programs and further information is available online [Hen97a], including comments on particular coding decisions in the respective languages.

8.6.1 Sieve of Eratosthenes

We use the algorithm given in Section 5.5. In this application, we have roughly twice as many attribute accesses as object application. In comparison, attribute access and object creation are negligible. All object applications use dynamic binding. If inline caching is used, very few cache misses occur. Compared to the object operations, little arithmetics is carried out. The performance of the systems being studied for computing the prime numbers among the first 20.000 natural numbers is summarized in Figure 8.2.

8.6.2 N-Queens

We use the algorithm given in Section 5.5. For benchmarking, we solve the 16-queens problem. We have about 5 times as many attribute accesses as message sendings, and these two constructs are dominant. Most message sendings use late binding. The performance of the systems is summarized in Figure 8.3.

An interesting aspect is how much time the programs spend on the arithmetic part of the problem. This number varies from 0.091 seconds in GNU gcc 2.7 to 3.34 seconds in Java SUN JDK. In Figure 8.4 we subtract a lower bound on the arithmetic computation time, resulting in a runtime closer to the time spent on

Table 8.2 Performance Figures for "Sieve of Eratosthenes"

System	runtime in seconds	runtime / runtime OzE
C++N	2.60	0.31
OCamlN	2.91	0.34
SmalltalkN	6.05	0.71
OCamlE	6.72	0.80
OzE	8.44	1.00
JavaN	9.43	1.12
JavaE	11.5	1.36
CLOSN	14.0	1.67
PrologN	15.7	1.86
PrologE	25.3	3.00

Table 8.3 Performance Figures for "16-Queens"

System	runtime in seconds	runtime / runtime OzE
C++N	0.355	0.059
OCamlN	0.546	0.091
SmalltalkN	1.36	0.23
JavaN	1.68	0.28
OCamlE	3.54	0.59
CLOSN	4.00	0.67
OzE	5.99	1.00
JavaE	6.88	1.15
PrologN	10.2	1.71
PrologE	14.4	2.41

Table 8.4 Performance Figures without Arithmetics for "16-Queens"

System	runtime arithmetics in seconds	runtime w/o arithmetics in seconds	runtime w/o arithmetics / runtime OzE w/o arithmetics
C++N	0.091	0.264	0.093
OCamlN	0.218	0.328	0.12
SmalltalkN	0.355	1.00	0.35
JavaN	0.392	1.29	0.45
OCamlE	1.86	1.68	0.59
OzE	3.16	2.84	1.00
JavaE	3.34	3.54	1.25
CLOSN	0.250	3.75	1.32
PrologN	1.41	8.83	3.11
PrologE	2.83	11.6	4.09

object-oriented constructs. We observe that only OzE and OCamlE spend more time on arithmetics than on object-oriented constructs. This indicates that object-oriented constructs are optimized well relative to arithmetics. On the contrary, arithmetics in SICStus Prolog is faster than in Oz, but SICStus Prolog spends most of the runtime in object-oriented constructs. We conjecture that objects in SICStus Prolog could benefit from the implementation techniques described in this chapter.

8.6.3 Performance Impact of Individual Optimizations

The impact of the individual optimizations on overall performance is hard to measure, since in a real implementation the optimizations are heavily intertwined and cannot be kept separate as neatly as in the above presentation. To a certain extent, the meta-object protocol described in Chapter 12 allows to undo the optimizations. Experiments with this system allow an estimation of about one order of magnitude in cumulative speedup by the described optimizations. The impact of lookup tables and inline caches on performance of object-oriented languages are studied in the literature [UP83, Ung86]. The optimization of **`self`** yields an estimated speedup factor of 1.2–1.4 for "pure" object-oriented programs like our case studies. Specific to Objects in Oz is the need to optimize first-class messages. The speedup of this optimization is impossible to measure in the current implementation since *all* other optimizations heavily rely on it. Easier to measure is the benefit in terms of memory consumption. We measured for the sieve program a memory consumption of 60.0 MBytes without the optimization compared to 266 kBytes with the optimization. This indicates that for languages with first-class messages, this optimization is crucial.

8.6.4 Summary of Performance Evaluation

Performance of DFKI Oz 2.0 lies in the range of state-of-the-art byte-code-based programming systems. Its object system performs well relative to arithmetics. Considering that most applications spend less time on object-oriented constructs than the benchmarks used, we conclude that further optimizing the object system becomes important only if other aspects are significantly improved.

8.7 Historical Notes and Related Work

The first abstract machine for a simple applicative programming language was Landin's SECD machine [Lan63], variants of which are used for the implementation of functional programming languages. Warren's Abstract Machine (WAM) [War83] forms the base of many Prolog implementations. For a good introduction to the WAM consider Aït-Kaci's tutorial reconstruction [AK91]. AMOZ was developed on the base of the WAM and retained some of its features.

The use of closures appeared with the lexically scoped language Algol. Sussman and Steele [SS75] describe environments—which they also call virtual substitutions—as an implementation technique for β-reduction in an implementation of an extended λ-calculus that became known as Scheme.

The possibility of using complete method tables to implement late binding was suggested by Steele [Ste76]. Lookup caches are introduced by Conroy and Pelegri-Lopart [CPL83], and inline caches by Deutsch and Schiffman [DS83], both in the context of Smalltalk-80 implementations.

As in Oz, the semantics of Smalltalk is based on first-class messages. However, the language is designed in such a way that the programmer can only get a hold of the messages upon error. Like in Oz, message creation is generally avoided in Smalltalk [GR83] and upon error, messages are reconstructed to get the right debugging behavior. Our optimization of first-class messages is related in spirit to *deforestation* [Wad90] in functional programming, where compile-time analysis is used to eliminate the need to building structures at runtime. In contrast to the situation in typed functional programming, we must prepare for reconstructing the structure (message) in case the callee needs it.

8.8 Discussion

We showed that with a few standard object-oriented implementation techniques and a careful treatment of first-class messages, we can bring the object system up to speed comparable to other byte-code-based object-oriented language implementations. The taken approach can be called *surgical* since we kept the general translation scheme prescribed by the semantics of the object system in the previous chapter and concentrated on speeding up individual critical aspects. There is virtually no compiler support for the object system. The compiler does not know about classes; translation of class definition is (with slight variations) as given in Chapter 7. We showed that support for objects in the runtime system alone, together with special treatment of messages, can yield performance comparable to other state-of-the-art object systems. The programmer can rely on the usual performance assumptions in object-oriented programming, including practically constant time access to attributes, features and methods and no memory consumption for messages. Performance of these operations relative to other basic operations in Oz such as arithmetics is acceptable.

An alternative implementation design would have been to provide support for class definition in the compiler, which would have incurred a significant implementation effort, but would have provided opportunities for optimization that our current design misses, as we shall see below.

Our surgical approach on the other hand allowed us flexibility that we found to be crucial during the design process of Oz, which contained dramatic design changes that also heavily affected the object system. So far there was no need to implement even central operations like method lookup or inheritance on the level of the abstract machine, and instead an easily maintainable high-level Oz implementation is still in use. The approach allowed us to concentrate on the performance-critical aspects and keep the implementation effort fairly low.

The following list contains possible further improvements that have not been pursued, partly because they would have incurred significant changes in the compiler or abstract machine, partly because the performance gain is hard to estimate.

Inlining. Method application that uses static binding could be optimized by *inlining* the method body in the calling code. For this optimization, the compiler needs to know which methods a given class has. Classes are first class citizens in Oz; we must provide for inheritance during compilation *where possible* to be able to optimize the case of static binding. When the issue of inlining is tackled for compiling Oz, inlining of methods should be considered.

Polymorphic Inline Caches. Hölzle, Chambers and Ungar [HCU91] show that it can be beneficial to extend the idea of inline caches to remembering several of the most common methods in late binding of object/method application. They call this technique *polymorphic inline cache*. For example, consider our solution to the n-queens problem in Program 6.5. The application

```
{self.neighbor canAttack(@row @column $)}
```

in method `testOrAdvance` calls the method `canAttack` of class `Queen` 1330 times and the method `canAttack` of class `NullQueen` 112 times during the search for the first solution of the 8-queens problem. Every call to `NullQueen` is preceeded and succeeded by a call to `Queen`. This means that we have 224 cache misses among 1442 calls, a miss rate of 15.5%. On the other hand, there are only two possible methods to call. Thus, if we extend our inline cache to contain two classes and the corresponding method addresses, we can avoid all cache misses at the expense of at most two equality tests per object application. This would surely improve runtime in the application at hand.

Free Variables. Free variables of methods are stored in the closure environment of the corresponding method closure. Typically methods share a significant part of these free variables. We could save heap space if we allocate for a class a *class closure environment* that can be shared by all methods.[10] This consideration becomes significant in applications where many classes are created at runtime.

---

[10]Steele [Ste76] already pointed out that minimal closures do not necessarily yield maximal efficiency.

Part III

Objects and Concurrency

This part investigates issues that arise when object-oriented programming concepts are used in the framework of concurrent programming. Chapter 9 shows that logic variables together with cells can express a wide variety of synchronization techniques for passive objects. We emphasize the ability to support these techniques by using object-oriented abstractions. Chapter 10 shows that Objects in Oz can readily express active objects. Chapter 11 discusses object-oriented concepts in concurrency models fundamentally different from concurrency by explicit threads. Chapter 12 presents a concurrent meta-object protocol for Oz.

The balloon was by this time tugging hard at the rope that held it to the ground, for the air within it was hot, and this made it so much lighter in weight than the air without that it pulled hard to rise into the sky.

"Come, Dorothy!" cried the Wizard. "Hurry up, or the balloon will fly away."

"I can't find Toto anywhere," replied Dorothy, who did not wish to leave her little dog behind. Toto had run into the crowd to bark at a kitten, and Dorothy at last found him. She picked him up and ran towards the balloon.

She was within a few steps of it, and Oz was holding out his hands to help her into the basket, when, crack! went the ropes, and the balloon rose into the air without her.

"Come back!" she screamed. "I want to go, too!"

"I can't come back, my dear," called Oz from the basket. "Good-bye!"

Chapter: How the Balloon Was Launched

Chapter 9

Synchronization Techniques

We saw in Chapter 4 that synchronized reduction with logic variables allows for data-driven synchronization of concurrent threads. In this chapter, we will explore the expressivity of Oz for more complex synchronization tasks. We start with data-driven synchronization in Section 9.1. Sections 9.2 through 9.7 show how objects together with logic variables can encode a variety of increasingly complex synchronization mechanisms. In Section 9.9, we integrate a common synchronization scheme in our object model. In Sections 9.10 and 9.11, we discuss more general issues in concurrent object-oriented programming.

9.1 Data-Driven Synchronization

We showed in Chapter 4 how threads can synchronize each other driven by the availability of data. If both producer and consumer of the data have a reference to a shared logic variable the producer can tell a basic constraint on the variable and the consumer can synchronize on the variable with a corresponding conditional. For example, consider the pair `Produce`/`Consume` in Program 9.1.

Program 9.1 Producer/Consumer

```
proc {Produce Xs}                proc {Consume Xs}
   X|Xr=Xs                           case Xs of X|Xr
in                                   then
   {ProduceItem X}                      {ConsumeItem X}
   {Produce Xr}                         {Consume Xr}
end                              end end
```

We run the procedures `Produce` and `Consume` in different threads.

```
declare Xs in
thread {Produce Xs} end          thread {Consume Xs} end
```

The producer thread synchronizes the consumer thread by telling a basic constraint right before it starts producing the item. Synchronization schemes in which the producer signals to the consumer that the data is (soon) available is called *push-based* [Lea97]. In this example, synchronization is done not on the data being produced but on the medium holding the data; `ConsumeItem` may be called with an unbound variable as argument and thus may need to synchronize on the data.

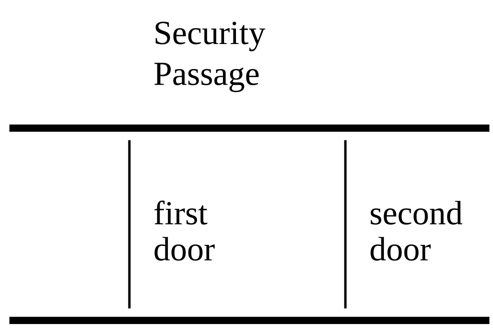
If the producer is likely to be faster, a *pull-based* scheme is a better choice. Here, the consumer asks the producer for the next item when it is ready for it. Interestingly, Program 9.1 can be used for pull-based control flow just by exchanging the roles of `ProduceItem` and `ConsumeItem`. Then, the consumer signals to the producer that it is ready for the next item by continuing the list from which it reads. The producer synchronizes on this list and delivers.

9.2 Mutual Exclusion

Stateless computation poses severe restrictions on the synchronization techniques that can be encoded. In particular, it seems impossible to express competition among threads for limited resources as required in mutual exclusion where we have to prevent that more than one thread executes a code segment at a time. We shall see in this section how to use cells for this task.

As an example take a double door in a security critical building (e.g. a bank vault). The figure below depicts the situation. A program to execute a passage of a person from left to right may look like this.

```
class Passage
   ⋮
   meth pass
      {self.first open}
      % person can enter
      {self.first close}
      {self.second open}
      % person can exit
      {self.second close}
   end
end
```

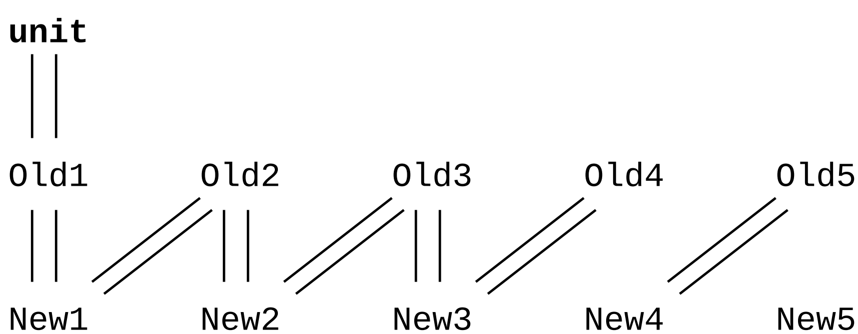


The safety requirement that never both doors should be open at the same time can be guaranteed, if no two threads can apply an instance of `Passage` concurrently to `pass` messages. Such mutual exclusion conditions occur frequently in stateful concurrent programming since typically stateful procedures go through transient violations of invariants that must be protected from observability by other threads. We implement mutual exclusion in Program 9.2. An application

Program 9.2 Mutual Exclusion

```
class SafePassage
   from Passage
   attr token:unit
   meth pass
      Old New in
      Old=token<-New
      {Wait Old}
      Passage,pass
      New=Old
   end
end
```

of a `SafePassage` object to a message `pass` will exchange the value in the attribute `token`. Atomicity of exchange guarantees that no two threads can get reference to the same value `Old`. Only one thread will be able to get **`unit`** out of `token` and enter the critical section `Passage,pass`. When the critical section is left, the tell statement `New=Old` will pass **`unit`** to the next waiting thread. Over time the exchange statements of different threads will implicitly build up a queue of variables of the form



Here, the edges represent equality constraints in the store; diagonal edges stem from exchange operations. The synchronization token **`unit`** is passed from left to right. In this example, five `pass` requests have been issued, and three of them have finished their critical section and executed the tell statement `Old=New`. The synchronization token is stuck at the fourth request which is currently executing `Passage,pass`.

9.3 Semaphores

The semaphore [Dij68] is the first widely used abstraction for synchronization. It merges mutual exclusion with the notion of a limited resource. To date, the semaphore is considered to be an essential albeit low-level synchronization mechanism and is mentioned as a minimal requirement a concurrent programming system has to provide [Boo94]. Operating systems with multitasking usually provide semaphores through library procedures.

A semaphore is an integer valued variable s on which the following operations are defined:

- $wait(s)$ If $s > 0$ then s is decremented by 1, else the executing thread is suspended. In this case, we say that the thread suspends *on s*.

- $signal(s)$ If there are threads that currently suspend on s, one of them is awoken, otherwise it is incremented by 1.

We require that both operations are atomic, and that s has a non-negative initial value n.

Semaphores can solve the mutual exclusion problem above by replacing the cell by a semaphore s of initial value 1, `{Wait Old}` by a wait operation on s and `Old=New` by a signal operation on s. If s is used in such a way, it can only assume the values 0 and 1. Such a semaphore is called *binary*. The integer value of the semaphore allows to express synchronization conditions coupled to a resource.

The semaphore is a higher-level abstraction for synchronization than the logic variable, since it enables continuous synchronization actions whereas, once a logic variable is bound, its synchronization capability is exhausted. Nevertheless, we can implement a semaphore with logic variables by dynamically creating new variables to refresh the synchronization capability.[1] In Program 9.3 we implement a semaphore class using synchronization with logic variables. A semaphore is represented by an object with the attributes `waitPointer` and `signalPointer`. Their values can be seen as pointers into a list. Initially, `waitPointer` points to a list of n **`unit`** values, and `signalPointer` to the tail of that list. For example, after

```
Sem={New Semaphore init(3)}
```

the attribute `waitPointer` of `Sem` holds a list **`unit`**`|`**`unit`**`|`**`unit`**`|Ur` and the attribute `signalPointer` holds the unbound variable `Ur`. Application of `Sem` to the message `wait` advances `waitPointer` and waits for the entry to

[1] Bill Silverman (quoted by Shapiro [Sha89]) compared the logic variable with a genie that grants you a single wish. Of course, the first thing to do when encountering such a genie is to wish to have two wishes!

Program 9.3 Semaphore Class

```
class Semaphore
   attr waitPointer signalPointer
   meth init(N)
      @waitPointer = {self listUnit(N $)}
   end
   meth listUnit(N $)
      case N of 0 then @signalPointer
      else unit|{self listUnit(N-1 $)}
      end
   end
   meth wait
      W|Wr = waitPointer <- Wr in
      {Wait W}
   end
   meth signal
      Sr in
      unit|Sr = signalPointer <- Sr
   end
end
```

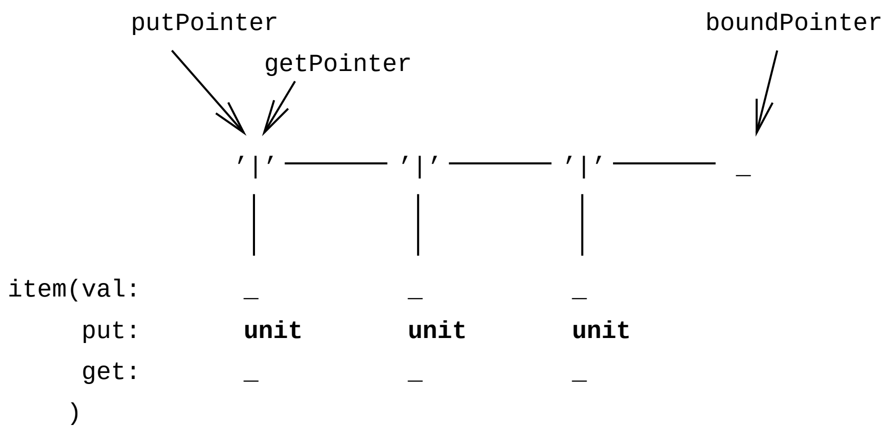
which it pointed before becoming bound. Application to `signal` advances `signalPointer` and binds the variable to which it pointed before, possibly waking up a thread in the `wait` method. If `signalPointer` points ahead of `waitPointer`, their distance represents the value of the semaphore. If `waitPointer` points ahead of `signalPointer`, the value of the semaphore is `0` and their distance represents the number of waiting threads.

9.4 Bounded Buffer

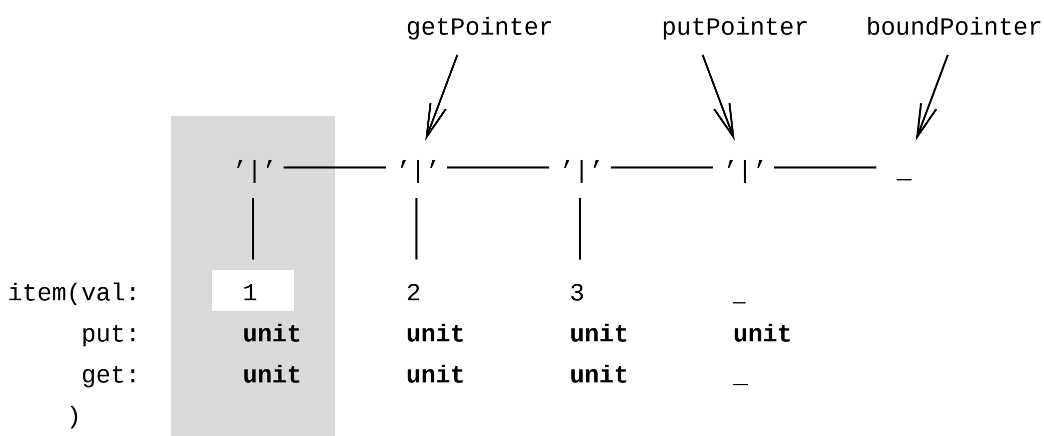
The producer/consumer example in Section 9.1 showed that a list can play the role of a buffer between communicating threads. The producer was not synchronized and thus the buffer could grow to arbitrary size. Dijkstra showed that semaphores can express *bounded* buffers [Dij68]. Instead of repeating his implementation, we show in Program 9.4 how to implement bounded buffers directly and more simply.

The attributes `putPointer`, `getPointer` and `boundPointer` represent three pointers into a list as depicted in Figure 9.1. The attribute `putPointer` points to the position where the next item can be put and the attribute `getPointer` points to where the next item can be gotten from. The items in the list hold the value and two variables on which the methods `put` and `get` synchronize. Fig-

Figure 9.1 Configurations of Bounded Buffer



(a) Initial Configuration



(b) Configuration after 3 Put and 1 Get

Program 9.4 Bounded Buffer

```
class BoundedBuffer
   attr putPointer getPointer boundPointer
   meth init(N)
      @putPointer = @getPointer = {self list(N $)}
   end
   meth list(N $)
      case N of 0 then @boundPointer
      else item(val:_ put:unit get:_)|{self list(N-1 $)}
      end
   end
   meth put(X)
      item(val:V put:P get:G)|Pr = putPointer <- Pr
   in
      {Wait P}
      G=unit X=V
   end
   meth get($)
      item(val:V put:_ get:G)|Gr = getPointer <- Gr
      item(val:_ put:P get:_)|Br = boundPointer <- Br
   in
      {Wait G}
      P=unit V
   end
end
```

ure 9.1(a) depicts the configuration of a buffer

```
BB={New BoundedBuffer init(3)}
```

right after initialization, and Figure 9.1(b) depicts the state of `BB` after

```
{BB put(1)} {BB put(2)} {BB put(3)}
declare X in {BB get(X)}
```

Note that the distance between `getPointer` and `boundPointer` is constant and represents the size of the buffer. The shaded area in Figure 9.1(b), i.e. the cons record, the item record and the two synchronization values, is not accessible anymore and thus subject to garbage collection, whereas the number `1` is accessible via `X`.

Program 9.5 Readers/Writer Problem

```
class ReadersWriters
   attr token: token(r:unit w:unit)
   meth read(Code)
      NewR NewW
      token(r:OldR w:OldW) = token <- token(r:NewR w:NewW)
   in
      NewR = OldR  % release read token
      {Wait OldR}  % wait for read token
      {Code}
      NewW = OldW  % release write token
   end
   meth write(Code)
      NewR NewW
      token(r:OldR w:OldW) = token <- token(r:NewR w:NewW)
   in
      {Wait OldW}  % wait for write token
      {Code}
      NewW = OldW  % release write token
      NewR = OldR  % release read token
   end
end
```

9.5 Readers/Writer

In the readers/writer problem [CHP71] several threads compete for a shared resource similar to mutual exclusion. The threads are divided into *reader* threads which are not required to exclude one another, and *writer* threads which are required to exclude every other thread, readers and writers alike. The problem is an abstraction of access to databases, where there is no danger in having several threads read concurrently, but writing or changing the data must be done under mutual exclusion to ensure consistency. Again, instead of using semaphores, we code the solution directly with logic variables as shown in Program 9.5. The program is a refinement of program 9.2 for mutual exclusion. Instead of using a single value synchronization token, the token is now a record. The `read` method waits for the `r` field of the synchronization token, whereas the `write` method waits for its `w` field. Overlapping of read requests is achieved by releasing the read token immediately.

Note that we use a record as token in order to be able to simultaneously "exchange" both fields. The program gets significantly more complex if only simple values are used.

9.6 Time Behavior

Often synchronization conditions occur in combination with time constraints. We extend Small Oz to provide primitives for soft real time programming as building blocks for more complex time behavior.

One essential building block is the procedure `Alarm`. It takes an integer *ms* as argument. `Alarm` suspends the current thread for at least *ms* milliseconds and when the thread is awoken, it returns **`unit`**.

A second essential component is called *simultaneous waiting*.

Simultaneous Waiting. Simultaneous waiting of the form

{`WhichFirst` *x y z*}

> is synchronized on *x* or *y* to be bound, i.e. it reduces if either one of the variables *x* and *y* get bound. When it reduces there are three possibilities.
>
> - The variable *x* is bound but not *y*; then *z* `= 1` is pushed.
> - The variable *y* is bound but not *x*; then *z* `= 2` is pushed.
> - Both variables are bound; then either one of *z* `= 1` and *z* `= 2` is pushed.

We show with two examples that `Alarm` and `WhichFirst` are versatile building blocks for time behavior.[2]

First consider the situation where concurrently produced computational events trigger graphical output. In order to avoid a too frequent display of the graphical output (flickering) we are introducing a time slack. An incoming event e_1 is not displayed immediately. Instead, if the next event e_2 happens within the time slack, then e_1 is simply ignored. Thus only the last one of a fast sequence of events will lead to a display.[3]

The method `doLazily` in class `Laziness` in Program 9.6 binds the current value of the attribute `token` to **`unit`** and replaces it by `NewVar`. It simultaneously waits on `NewVar` and the return value of `Alarm` to become bound. If `NewVar` gets bound faster—i.e. the same instance gets applied to `doLazily` within `Slack` milliseconds—then `lazyDisplay` ignores the message, and otherwise it executes `Code`.

[2] The procedure `Alarm` is included in the standard libraries of Oz, whereas the procedure `WhichFirst` can be programmed in Oz using `IsDet` and `==`.

[3] This example originates from the implementation of Ozcar, a debugger for Oz developed by Benjamin Lorenz. In Ozcar, the threads created by a program are displayed in a graphical tool. If many threads were created in the program being debugged, the resulting graphical output flickered.

Program 9.6 A Class for Laziness

```
class Laziness
   attr token
   meth doLazily(Code Slack)
      NewVar
   in
      unit = token <- NewVar
      case {WhichFirst NewVar {Alarm Slack}}
      of 1 then skip
      else {Code}
      end
   end
end
```

A second example provides a generic repetition functionality. An instance of the class `Repeat` shown in Program 9.7 can be made to repeatedly apply an `Action` procedure using the method `go`. The method `go` calls the private method `Go`, which implements a loop, which is at every iteration delayed by `@DelayTime` milliseconds. The loop is terminated, when the attribute `@Stop` becomes bound. This can be done by applying the `Repeat` object to the message `stop` in a concurrent thread or the procedure `Action`. Note that the method `stop` stops all `go` requests that have been issued after the last `stop` (or, if there was no `stop` yet, after object creation).

These two examples show that high-level time dependent abstractions can be defined using the primitives `Alarm` and `WhichFirst`.

9.7 Locks

Similar to the time abstractions `Laziness` and `Repeat` in the previous section, we extract the mutual exclusion functionality in Program 9.2 and provide it generically by the class `Lock` in Program 9.8.

We simply give to the lock as an argument the code to be executed in a critical section in the form of a nullary procedure. Using `Lock`, Program 9.2 becomes much simpler, as shown in Program 9.9.

9.8 Thread-Reentrant Locks

A problem arises when the same thread tries to enter a lock that it already holds. For example, in Program 9.2 it is reasonable that the door **`self`**`.first` is pro-

Program 9.7 A Repeater Class

```
class Repeat
   attr
      DelayTime: 1000
      Stop
   meth go(Action)
      Repeat,Go(Action)
   end
   meth stop
      unit = Stop <- _
   end
   meth Go(Action)
      S = {Alarm @DelayTime}
   in
      {Action}
      case {WhichFirst S @Stop}
      of   1 then Repeat,Go(Action)
      else 2 then skip
      end
   end
end
```

tected by the same lock as passage, such that the first door cannot be manipulated in any other way while someone passes the double door. Implementing the door's `open` method by

```
meth open
   {@lck lck(proc {$} ··· end)}
end
```

will lead to a deadlock, since `OpenFirst` waits for the lock that is never going to be released because `OpenFirst` is waiting for the lock... This situation led to the conception of *thread-reentrant* locks that do not require the lock if the current thread already holds it. The implementation of thread-reentrant locks in Program 9.10 uses the primitive `ThisThread` to be able to identify which thread currently holds the lock.

Thread identification. A thread identification of the form

$$\{\texttt{ThisThread}\,x\}$$

pushes the statement $x = \xi$ where ξ is a name that uniquely identifies the reducing thread. Thus, subsequent thread identifications issued by the same

Program 9.8 Lock

```
class Lock from BaseObject
   attr token: unit
   meth lck(Code)
      New Old = token <- New in
      {Wait Old}
      {Code}
      New = Old
   end
end
```

Program 9.9 Mutual Exclusion with Locks

```
class SafePassage
   from Passage
   feat lck
   meth init
      self.lck = {New Lock noop}
   end
   meth pass
      {self.lck proc {$} Passage , pass end}
   end
end
```

thread always yield the same name, and thread identifications issued by different threads always yield different names.

The class `ReentrantLock` in Program 9.10 inherits from `Lock` and redefines the method `lck` such that it immediately executes `Code` if the current thread already holds the lock. Otherwise `Code` is protected by the inherited method `lck` making sure that the attribute `lockingThread` always refers to the thread that currently holds it or to **`unit`** if it is free.

The necessity for thread identification arises naturally in concurrent programming. Lopez and Lieberherr [LL94] mention this feature as one of the basic constructs that a reasonably expressive concurrent language must provide.

The idiom of thread-reentrant locks is so important that we syntactically support it such that instead of

```
{L proc {$} ··· end}
```

the following more pleasing syntax can be used

```
lock L then ··· end
```

The procedure `NewLock` is predefined as

Program 9.10 Reentrant Locks

```
class ReentrantLock from Lock
   attr lockingThread: unit
   meth lck(Code)
      This={ThisThread}
   in
      case @lockingThread==This
      then {Code}
      else Lock , lck(proc {$}
                         lockingThread <- This
                         {Code}
                         lockingThread <- unit
                      end)
      end
   end
end
```

```
fun {NewLock} {New ReentrantLock noop} end
```

Note that both in reentrant and non-reentrant locking, the lock does not affect threads that are created within the lock. The lock is released when the thread that entered the lock is finished with reduction of the locked statement.

```
lock
   L
then
   {P}
   thread {Q} end
   {R}
end
```

As soon as reduction of `{R}` is completed, the lock is released regardless whether `{Q}` is finished or not. If we wish to synchronize on `{Q}` as well, this needs to be programmed explicitly with an acknowledgment variable as in

```
lock
   L
then
   Ack in
   {P}
   thread {Q} Ack=unit end
   {R}
   {Wait Ack}
end
```

9.9 Objects with Reentrant Locks

In Program 9.9, mutual exclusion was achieved by binding the lock to an object feature upon initialization and accessing this feature for the lock in the methods. This idiom—in combination with thread-reentrant locks—is so important that we decided to support it syntactically. We allow to declare a property **prop** locking for a class which has the effect that a private feature of every instance is bound to a different lock. The property locking is inherited. Mutual exclusion can be enforced for statements *S* by writing within methods **lock** *S* **end**, which refers implicitly to the lock of the current object. Thus a reentrant version of Program 9.9 can be written as

```
class SafePassage
   from Passage
   prop locking
   meth pass
      lock Passage , pass end
   end
end
```

In the context of concurrency it becomes clear why we insist on an initial message for object creation. In a sequential context, we could instead simply first create the object and then apply it to the initialization message. In a concurrent context, however, it is possible that another thread has already a reference to the variable to which the newly created object is bound and tries to apply it. It could happen that this object application gets executed before the object gets applied to the initial message, which clearly defeats the purpose of *initial* messages. The definition of New in Program 7.3 prevents this by returning O only after the object application {O Message}.

Thread-reentrancy solves a problem that plagues languages with synchronized objects, namely that self application of such objects leads to immediate deadlock. Instead of introducing thread-reentrancy as in Oz and Java, the language POOL-T [Ame87] allows for direct method invocation that bypasses synchronization. The languages ConcurrentSmalltalk [YT87] and Obliq [Car95] change the semantics of **self** in an ad-hoc way such that object application that is statically identified as self application is not synchronized. However, indirect application of the current object or object application where the callee turns out to be self at runtime are synchronized, which can lead to unwanted deadlocks.

Note that making the lock feature public is generally unsafe since any reference to an object O can lock it forever as in

```
thread lock O.theLock then {Wait _} end end
```

A drawback of making the lock implicit is that it is not available to the programmer. Lea [Lea97] shows a number of programming techniques where this is essential. However, we can implement access to the object's lock with the following method `LOCK`.

```
meth !LOCK(P) lock {P} end end
```

In order to use the lock of an object `O` from outside for a statement *S*, we can write

```
{O LOCK(proc {$} S end)}
```

For security, the scope of `LOCK` can be controlled by the programmer. In the context of locking the situation may arise that several messages must be processed without releasing the lock. This can be achieved by the following method `batch` which employs first-class messages.

```
meth batch(Ms)
   lock {ForAll Ms proc {$ M} {self M} end} end
end
```

9.10 Inheritance and Concurrency

We pointed out in Sections 2.2.3 and 5.4 that non-conservative inheritance like any complex feature necessitates careful design and can be the source of programming errors. In the context of concurrent object-oriented programming, the dangers of non-conservative inheritance have been studied by Matsuoka and Yonezawa [MY93] who coined the term "inheritance anomaly". The source of these dangers is that synchronization often depends on non-local properties of objects.

Our conclusion of this observation is to propagate particularly careful usage of inheritance in the context of concurrency. Lea gives an overview of the issues that need to be kept in mind for concurrent programming in Java [Lea97]. Due to the close relation of the concurrency model of Oz and Java, his remarks are equally valid for Oz.

In languages like ABCL [Yon90] and POOL-T [Ame87], whose main tool of synchronization is synchronization code in the form of method guards, the problem is more urgent. Here, the "anomaly" pointed out by Matsuoka and Yonezawa consists mainly of the lack of a generally applicable method for inheritance of this synchronization code. This situation convinced the designers of these languages to abandon inheritance altogether.

Thread-reentrant locking of Java and Oz avoids the most urgent synchronization problem in the context of inheritance and concurrency, namely self and method application of synchronized methods as argued in Section 9.9.

Lea [Lea97] notes that immutable attributes provide strong invariants to the concurrent programmer. He argues that a particular danger in the context of inheritance lies in changing attributes in subclasses that were used in superclasses under the immutability assumption. Our radical decision to syntactically and semantically separate mutable components (attributes) from immutable components (features) helps to avoid errors of this kind.

9.11 Discussion

None of the presented synchronization techniques is new. We emphasize however the ease with which a wide variety of synchronization abstractions can be built in Oz from a small number of simple building blocks. To summarize, the programming concepts that we relied upon in this section include

1. thread-level concurrency,
2. object-oriented programming (inheritance, encapsulation),
3. logic variables for synchronization,
4. atomic attribute exchange as main indeterministic construct, and
5. first-class procedures.

It is the integration of these features in a coherent programming framework that turns Oz in a powerful concurrent language.

Both logic variables and the exchange operation were used in every program after Section 9.1. This justifies their prominent position in the language definition and the syntactic support for attribute exchange in the object system.

The concurrent functional language Multilisp [Hal85] supports some of these features. We are now in a better position to compare Multilisp's futures mentioned in Section 3.5 to logic variables. A future is statically tied to an expression that computes its value. Interestingly enough, after Section 9.1 every program relies on the fact that there is no such a restriction for logic variables. We attribute to this difference the fact that the implementation of these idioms with futures is much more complex in Multilisp than in Oz (see for comparison the semaphore given in [Hal85]). Similar to Multilisp's futures are ConcurrentSmalltalk's CBoxes [YT87] and ABCL's future objects [YBS86].

The language PCN [FOT92] features both thread-level concurrency and logic variables, however lacks lexically scoped higher-order programming and does not support object-oriented programming. PCN does not allow to access mutable variables (which correspond to Oz's attributes) from concurrent threads. The designers of PCN here obviously opted for security against expressivity, since this decision precludes the simple expression of essential synchronization mechanisms that are enabled by Oz's cells with atomic exchange. Communication of concurrent threads in PCN is stream-based which has proven to be difficult to use in practice in concurrent logic programming.

The language Java [AG96] supports thread-level concurrency and object-oriented programming. It provides a built-in notion of mutual exclusion in the form of `synchronized` methods and statements, with which atomic attribute exchange can be implemented. In its newest version 1.1, Java provides for lexically scoped higher-order programming in the form of "inner" classes. Java's main synchronization constructs `wait` and `notify` can only be used within synchronized methods and are similar to the constructs `wait` and `signal` in Hoare's monitors [Hoa72]. An advantage of synchronization in Oz over Java is the simplicity with which data-driven synchronization is provided, since the logic variable is supported as a basic notion.

In Java, any object can use a `synchronized` method or statement, whereas our corresponding syntactic support for locking requires the corresponding class to have the property `locking`. Without this requirement, every object must be prepared for synchronization, so either the object must be provided with a lock upon initialization, or a lock must be created upon the first attempt to synchronize. This is due to Oz's dynamic typing and general method application. In Java, it is statically known which classes have instances that may be locked and thus the locking property can be kept implicit without sacrificing simplicity or efficiency of sequential objects.

So he walked forward to the tree, but just as he came under the first branches they bent down and twined around him, and the next minute he was raised from the ground and flung headlong among his fellow travelers.

This did not hurt the Scarecrow, but it surprised him, and he looked rather dizzy when Dorothy picked him up.

"Here is another space between the trees," called the Lion.

"Let me try it first," said the Scarecrow, "for it doesn't hurt me to get thrown about." He walked up to another tree, as he spoke, but its branches immediately seized him and tossed him back again.

"This is strange," exclaimed Dorothy. "What shall we do?"

Chapter: Attacked by the Fighting Trees

Chapter 10

Active Objects

In the previous chapter, we saw how objects can exhibit specific concurrent behavior. Unlike purely sequential objects, concurrent objects can suspend their current thread via data flow synchronization. However, so far we maintained a clear separation between threads and objects. The only sources of computational activity are threads; objects are passive and can control threads via synchronization.

In Section 4.2, we showed how *active objects* are represented in concurrent logic programming. An active object is an object that is associated with a thread of its own that carries out the operations on the object. In this chapter, we will further explore active objects, leading to abstractions for many-to-one communication (Section 10.1) and servers (Section 10.2). A case study demonstrates how these abstractions can be used in the context of simulation (Section 10.3), and a performance analysis (Section 10.4) evaluates the practicality of using active objects as a central object-oriented programming concept.

10.1 Many-to-One Communication

We saw in Section 4.2 that active objects can be supported in Small Oz in the style of concurrent logic programming by installing a thread devoted to repeatedly reading messages from a stream. Messages are sent asynchronously by extending the stream. Here the metaphor of "message sending" is appropriate.[1] The sender thread does not wait for receipt of the message and the receiver thread is responsible for carrying out the requested computation.

A basic problem with such stream-based objects is that there is exactly one position in the stream where the next message can be entered, while there are usually many senders that can possibly do so. Thus the senders have to coordinate their writing activity in order to avoid that two senders attempt to enter a message at

[1]Compare with terminology discussion on page 17.

Program 10.1 Expressing Ports with Cells

```
proc {NewPort Ms P}
   {NewCell Ms P}
end
proc {Send P M}
   Ms Mr in
   {Exchange P Ms Mr}
   Ms = M|Mr
end
```

the same position. Janson, Montelius and Haridi [JMH93] survey the techniques for many-to-one communication in concurrent logic programming and point out their problems and limitations. None of the surveyed techniques has constant time and space complexity for many-to-one communication. Kahn [Kah89] notes that "there are many ways of attaining many-to-one communication in the framework of concurrent logic programming. All of these methods are fundamentally awkward, especially when compared with actors or objects that support many-to-one communication as a primitive notion." Janson, Montelius and Haridi decided to integrate in concurrent logic programming such a primitive notion, and call it *port*. A port is an opaque front-end to a stream, realized by the following two operations. The operation `{NewPort Ms P}` creates a port `P` and connects it to a stream `Ms`, and the operation `{Send P M}` sends a message `M` to a port, which will put it in the right place of its stream.

Ports can be implemented using cells as in Program 10.1, yielding a constant time and space mechanism for stream-based many-to-one communication. The idea is to represent the port as a cell holding the current tail of the stream. The send operation performs an exchange on the cell putting a new variable `Mr` in the cell. The old content `Ms` of the cell is bound to a list with head `M` and tail `Mr`. Since the exchange operation is atomic, a continuous stream of messages is built without two senders ever attempting to bind the same variable to a message.

Janson, Montelius and Haridi emphasize that the stream connected to a port should be closed with `nil` when the port cannot be referenced any longer. This provides a form of garbage collection and becomes important when active objects are used for fine-grained structuring of data.

10.2 Servers

Our goal is to support active objects by reusing the sequential object system. Specifically we want to be able to define a class with the usual syntax and create active objects from it.

Program 10.2 Creating Servers for Objects

```
fun {MakeServer Object}
   Stream
   proc {Serve M|Mr}
      {Object M} {Serve Mr}
   end
in
   thread {Serve Stream} end
   {NewPort Stream}
end
```

Creating Servers for Objects

The first step is to provide the ability to confine the computation resulting from an object application to a dedicated thread. To this aim the procedure `MakeServer` in Program 10.2 installs a `Port` in front of a given `Object`. The stream connected to the port is continuously read by the procedure `Serve`, which runs in its own thread. After

```
S={MakeServer O}
```

the programmer can decide to perform an operation on `O` using that thread by `{Send S M}` or using the current thread as usual with `{O M}`. Note that the operations on `O` issued by `S` are performed in strict sequential order. In `Serve`, `Object` is applied to the next message only after the previous object application has finished. This strict sequentiality provides strong invariants to the programmer at the expense of disallowing any concurrent interleaving where this would do no harm.

An alternative would be to spawn a new thread for every message using the following `Serve` procedure.

```
proc {Serve M|Mr}
   thread {Object M} end {Serve Mr}
end
```

Here object applications can concurrently interleave and the programmer must use synchronization techniques such as the ones presented in the previous chapter to enforce synchronization conditions. The execution of the server is not confined to a single thread, but scatters itself over as many threads as there are messages being processed concurrently at any point in time.

Program 10.3 Encapsulating Objects in Servers

```
fun {NewServer1 Class Init}
   Object={New Class Init}
in
   {MakeServer Object}
end
```

Encapsulating Objects in Servers

Often an object is intended for exclusive use in a server. For this purpose, the procedure `NewServer1` in Program 10.3 can be used instead of object creation via `New`. The procedure `NewServer1` defines a local `Object` and hands it to `MakeServer`. The only way to access `Object` is by sending messages to the port returned by `MakeServer`. Thus the port represents an active object to which we can send messages via `Send`.

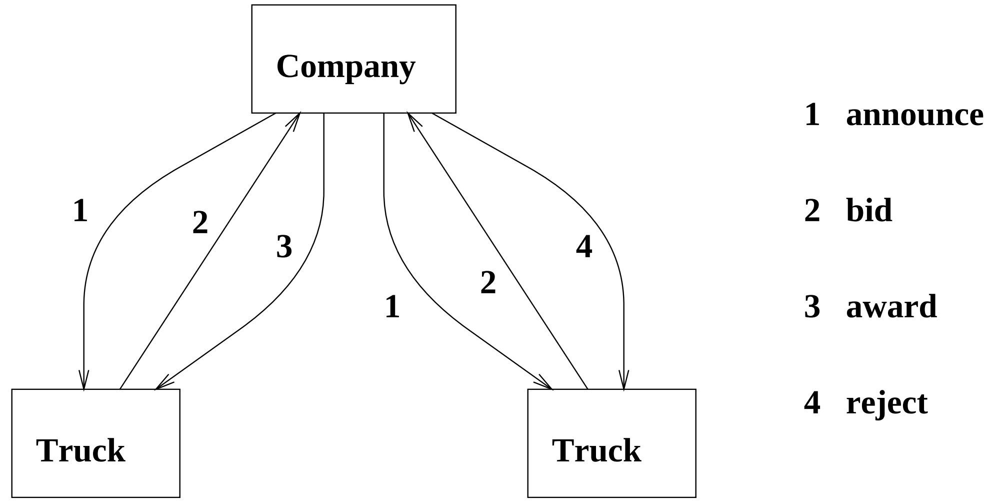
A Server Class

By construction, **`self`** in methods refers to the object and not the server. In order to enforce that the only way `Object` is accessed is through its port, we must make sure that **`self`** is not passed as argument to the outside. Instead, methods should be able to access and export the current *active* object, represented by its port. Therefore, we refine our server abstraction as shown in Program 10.4.

Program 10.4 A Server Class

```
local
   InitServer = {NewName}
in
   class Server
      attr server
      meth !InitServer(Port)
         @server = Port
      end
   end
   proc {NewServer Class Init ?Port}
      Object={New Class InitServer(Port)}
   in
      {Object Init}
      Port={MakeServer Object}
   end
end
```

Figure 10.1 A Simplified Contract Net Protocol



The argument `Class` of `NewServer` is assumed to inherit from `Server`. The object is initialized using the method `InitServer`, which is visible only in the procedure `Server` and in the class `NewServer`. The method `InitServer` binds the attribute `server` to the variable `Port` which is bound by `MakeServer` as in the procedure `NewServer1`. For self application to a message `M`, the programmer can now decide whether to use the active object via `{Send @server M}` or the passive object via `{self M}`. While the former possibility may decrease the latency for processing messages by the active object, it bears the danger of deadlocks. In particular, synchronization on return arguments in the thread of the active object as in

```
meth ··· {Send @server m(?X)} {Wait X} ··· end
```

immediately leads to a deadlock.

10.3 Case Study: Contract Net Protocol

As an example for a scenario in which active objects are useful, consider the simulation of the following distributed negotiation protocol. A transportation company has several trucks at its deposal. In order to fulfill incoming orders, it forwards them via mobile telephony to the trucks who reply with an estimated cost, depending on their current position and schedule. The company selects the most economic bid and awards the order to the corresponding truck. Figure 10.1 depicts the protocol.

Program 10.5 Negotiation Protocol in a Transportation Scenario

```
class Company from Server
   attr trucks
   meth order(Order)
      Announcements=
         {Map @trucks
          fun {$ Truck}
             Ann=ann(order:Order bid:_ award:_)
          in
             {Send Truck Ann} Ann
          end}
   in
      {FoldL Announcements
       fun {$ SmallestBid#SmallestAward
              ann(order:_ bid:Bid award:Award)}
          case Bid < SmallestBid                         %1
          then SmallestAward=false Bid#Award
          else Award=false SmallestBid#SmallestAward
          end
       end
       MaxCost#false}.2 = true
    end
end
class Truck from Server
   meth ann(order:Order bid:?Bid award:Award)
      Bid={self computeBid(Order $)}
      case Award then {self addToSchedule(Order)}    %2
      else skip end
   end
end
```

The contract net protocol [Smi80] was developed as a general computation framework for such negotiation protocols. We show here in principle how active objects can implement such a protocol and demonstrate the use of logic variables.[2]

We represent both the company and the trucks as active objects in order to model their distributed computational resources. The corresponding classes are given in Program 10.5. The protocol between an instance of `Company` and the instances of `Truck` referred to by `Company`'s attribute `trucks` is initiated by sending `order(Order)` to the company. The company forwards this order to

[2]After a feasibility study by Christian Schulte, Oz has been used as an implementation platform for a distributed collaborative transportation scenario in which variations of the contract net protocol were used [FMP95].

its trucks in the form of messages `ann(order:Order bid:_ award:_)` with new variables at the fields `bid` and `award` for each truck. In the course of the negotiation, these variables will be used for communication and synchronization between the company and its trucks. The trucks concurrently compute their bid on the order using the method `computeBid` not shown here and bind the field `bid` of the received announcement message. The company iterates through the list of all announcements to find the best bid and awards the order to the corresponding truck by binding the `award` field of the announcement message to **`true`**. The `award` field of all other announcements are bound to **`false`**. There is only one message sending per truck and order being sent. The synchronization required by the protocol is done through logic variables. At synchronization point *`%1`*, the company waits for each truck to have delivered the bid, and at synchronization point *`%2`*, the truck waits for the company to award or reject the order.

With conventional message passing a much more complex communication protocol would have to be used. For example, trucks must respond to the order via message passing. In general, the message must identify the order to which it refers so that the company knows to which order the truck responded. This example shows how natural data flow synchronization with logic variables can be used with active objects and that the server abstraction allows to create active objects from classes defined with the usual class notation.

10.4 Performance Analysis

Some concurrent object-oriented languages use active objects as the central programming idiom. Examples include the languages ABCL [Yon90], POOL-T [Ame87] and Eiffel|| [Car93]. In this section, we examine if this is desirable for Oz. The central requirement that we insist on is that the language should be practical for ordinary sequential object-oriented programming. In our experience this requirement is essential, since even for concurrent applications, typically large parts can and should be implemented with sequential programming concepts.

In order to get an impression of the consequences of active objects for the performance of sequential algorithms, we implemented the programs given in Sections 5.5 and 6.11 using active objects. In the case of the sieve of Eratosthenes, every filter is represented by an active object. Sending a number to the first filter must be synchronized such that the previous number made it through the chain or got filtered out. Thus a synchronization token is threaded through the methods `f`. No further change to the program was necessary. In the case of n-queens, every queen is an active object. No such synchronization was necessary here and thus migration was even simpler than for the sieve. The resulting programs are given in [Hen97a]. We note that migrating code from passive to active objects is often

Table 10.1 Performance Figures Passive vs Active Objects

Benchmark	runtime passive obj. in seconds	runtime active obj. in seconds	runtime active / runtime passive
Sieve	8.44	107.3	12.7
Queens	5.99	35.2	5.88
Queens w/o Arit.	2.84	32.0	11.3

much harder than these examples suggest. In particular, getting the synchronization conditions right can be tedious and error-prone.

Table 10.1 summarizes the performance results obtained under the conditions given in Section 8.6 and compares them with the sequential encoding. Active objects incur a runtime overhead of at least a factor of 10 (for n-queens after subtracting arithmetics). This overhead is incurred by switching control from one thread to another where the encoding with passive objects simply uses object application. Specifically, in sequential algorithms a message sending to an object *o* leads to waking up *o*'s thread and suspending the current thread. Given the fact that threads are light-weight in Oz and that the suspension mechanism is efficiently implemented, we conjecture that a slowdown of about one order of magnitude is intrinsic for a sequential implementation of Oz.

The active object version of the sieve used 70.3 MBytes of heap memory compared to 266 kBytes for the passive objects version. The active object version of the n-queens program used 24.0 MBytes compared to 634 kBytes for the passive objects version. The enormous memory consumption for active objects is due to the fact that messages must be built on the heap to store them in message streams, whereas messages to passive objects are usually not represented on the heap (see Section 8.5.2).

Unfortunately there are no performance figures on standard hardware for languages based on active objects given in the literature. McHale [McH94] vaguely suggests "several orders of magnitude overhead" for active objects depending on the expressivity of synchronization code. Even for parallel hardware, we found only a single performance analysis, carried out by Taura, Matsuoka and Yonezawa who evaluate the performance of ABCL on multicomputers [TMY93] using mathematical programming benchmarks. We conjecture that languages based on active objects so far failed to prove their practicality as general-purpose programming languages on standard hardware.

10.5 Discussion

We argued that active objects are a valuable programming idiom in applications and demonstrated the use of active objects in a simulation scenario. We showed that active objects can be expressed using passive objects and first-class messages. We showed that straightforward use of active objects for sequential algorithms incurs a significant overhead in our implementation. It has been shown that sophisticated compile-time analysis can significantly reduce this overhead [PZC95]. What still remains however is the conceptual burden of active objects for sequential programming. More generally, Lopez and Lieberherr [LL94] argue that treating concurrency and object-oriented organization of data as orthogonal issues increases the flexibility and expressiveness of a programming language.

In this chapter we made heavy use of first-class messages. In languages without first-class messages, such as Java, C++ and Smalltalk[3], it is much more difficult to combine active and passive objects. In retrospect, this observation justifies the introduction of first-class messages as a basic ingredient of our object model and the implementation effort incurred by them.

We conclude that while active objects provide a useful programming abstraction, a concurrent object system should be based on passive objects, which can—together with first class messages—support abstractions for active objects.

10.6 Historical Notes and Related Work

Streams are introduced by Landin [Lan65] who characterized them as functions $() \rightarrow \mathit{element} \times \mathit{stream}$. Gilles Kahn's process network [Kah74] is the first approach to modeling a distributed system using streams (which he calls channels). Kahn and MacQueen introduced stream merging for many-to-one communication [KM77].

Concurrent logic programming follows this approach by providing each potential sender with its own stream [ST83, KTMB86], and merging them into the stream that is read by the receiver object. Often, binary trees of merge agents are used. The most widely used programming idiom in concurrent logic programming to implement stream merging is *committed choice*, which was introduced in the Relational Language [CG81] and used in variations in every following concurrent logic language (for an overview see [Sha89]).

The communication structure in a concurrent object-oriented program is typically dynamic. At runtime, object references are passed around and new potential

[3] Smalltalk's computation model defines messages as objects, but the language is designed such that the user can access these message objects only in exception handling.

senders appear. A program that uses stream merging for many-to-one communication therefore must introduce a merge process each time a new potential sender is introduced. To relieve the programmer of this tedious and error-prone task, the object-oriented extensions to concurrent logic languages Vulcan [KTMB87], A'UM [YC88] and Polka [Dav89] automatically introduce sender streams and mergers. This leads to a proliferation of streams since many of them are not actually used. Communication with stream-merging cannot achieve constant-time behavior, which makes it problematic as a central computational idiom.

Atomic test unification, introduced by Saraswat [Sar85], allows many-to-one communication without the need for stream merging, but cannot achieve constant-time message sending either [JMH93].

Only after Janson, Montelius and Haridi [JMH93] introduced ports, active objects became a practically useful programming idiom in concurrent logic programming. We showed that the basic idea of ports can be implemented using cells. Vice versa, ports can express cells by modeling cells with active objects reading exchange requests from a stream connected to a port as shown in [Jan94].

The actors model of computation [Hew77, HB77] can be seen as the first programming model based on active objects. However, its underlying concurrency model is fine-grained and thus there is no one-to-one relation between an object and a thread of control. In the next chapter, we shall discuss fine-grained concurrency. Based on the actors model, Yonezawa developed the language ABCL [Yon90]. In contrast to the actors model, ABCL's concurrency is coarse-grained. The methods of active objects are defined by Lisp-like procedures, using suitable synchronization primitives.

The language POOL-T [Ame87] is based on the idea of explicit message reception. Object bodies define the processing of messages by active objects, including their synchronization behavior.

The language Eiffel|| [Car93] integrates active objects conservatively into the language Eiffel by adding a new base class called `PROCESS` and the corresponding compiler support. The instances of `PROCESS` exhibit the behavior of active objects. Furthermore, automatic data flow synchronization is provided similar to futures as described in Section 3.5.

ABCL, POOL-T and Eiffel|| support active objects as the only form of concurrent objects. It is not possible to define passive objects with synchronization behavior such as mutual exclusion.

The great spider was lying asleep when the Lion found him, and it looked so ugly that its foe turned up his nose in disgust. Its legs were quite as long as the tiger had said, and its body covered with coarse black hair. It had a great mouth, with a row of sharp teeth a foot long; but its head was joined to the pudgy body by a neck as slender as a wasp's waist. This gave the Lion a hint of the best way to attack the creature, and as he knew it was easier to fight it asleep than awake, he gave a great spring and landed directly upon the monster's back. Then, with one blow of his heavy paw, all armed with sharp claws, he knocked the spider's head from its body. Jumping down, he watched it until the long legs stopped wiggling, when he knew it was quite dead.

Chapter: The Lion Becomes the King of Beasts

Chapter 11

Alternative Concurrency Models

Concurrency is introduced in Oz explicitly, using **thread** $\cdots$ **end**. The composition construct (juxtaposition) realizes sequential composition. Unless the programmer introduces threads, programs run strictly sequentially. We call this style of concurrent programming *coarse-grained concurrency*.

In this chapter, we contrast coarse-grained concurrency to different approaches, where the main composition construct is interpreted as concurrent (1) or potentially concurrent composition (2). In the first case, every statement runs concurrently by default, and sequentiality must be enforced explicitly (Section 11.1). Such *fine-grained concurrency* is the underlying concurrency model of concurrent languages as diverse as Hewitt's actors model [Hew77, HB77], data-flow languages [Den74] and concurrent logic (constraint) languages [Sha89]. In Section 11.2 we examine the impact of fine-grained concurrency on models for object-oriented programming. In the second case, concurrency is introduced on demand, i.e. for suspending statements on top of the stack. This model was used in a previous version of Oz and is explained in more detail in Section 11.3. Its impact on object models is discussed in Section 11.4.

11.1 Fine-Grained Concurrency

The underlying motivation for investigating fine-grained concurrent programs was the hope that massively parallel computer hardware would enable their efficient execution. Examples for fine-grained concurrent programming frameworks are data-flow languages [Den74], concurrent logic programming languages [Sha89] and Hewitt's actors model [HB77], which was further developed by Agha [Agh86].

As a gedankenexperiment we can turn Small Oz into a language with fine-grained concurrency by reinterpreting the composition construct. Instead of pushing both components on the stack of the reducing thread, we create a thread for each component.

Composition. A composition of the form S_1 S_2 is unsynchronized. Reduction creates two new threads and pushes S_1 on the stack of the first one and S_2 on the stack of the second.

Instead of "thread", we shall call such a concurrently active entity *actor*. In a framework of fine-grained concurrency, the active/passive dichotomy for objects given in Section 2.3 becomes blurred since any operation on an object represents an actor of its own, which typically computes by splitting up into many more actors.

11.2 Objects for Fine-Grained Concurrency

The conventional use of state in programming relies heavily on sequential execution order which must be imposed explicitly in the context of fine-grained concurrency. On the other hand, it is this sequential execution that was considered the "bottleneck" to be overcome by fine-grained concurrency.[1] Consequently, the notion of sequential state was abandoned by the designers of object-oriented languages on the base of fine-grained concurrency.

The actors model uses a special **`become`** statement to replace the current actor by a new actor on which subsequent operations are carried out. Stateful programming is obtained in this scheme by computing the new actor as an incremental modification of the old one.

This model of state was adopted by the object-oriented extension of the concurrent logic language Polka [Dav89], where syntactic support for incremental modification is provided. Specifically, all Polka expressions of the form

`a` **`becomes`** *e*

that get executed during the processing of a message together define the new state that is used in the processing of the next message. Similarly, the object-oriented extension of Concurrent Prolog, Vulcan [KTMB87], supports the syntax

`new` `a` **`is`** *e*

To preserve maximal concurrency and still meaningful stateful programming, the modification of the state does not take effect until the processing of the next message. In both languages it is implicitly assumed that only one such statement per attribute is executed during the processing of a message.

[1] Backus [Bac87] famously talked of the "von Neumann bottleneck" of imperative programming that needed to be overcome by (pure) functional programming, which lends itself naturally to fine-grained concurrency.

In these languages, the implementation of any non-trivial sequential algorithm becomes syntactically difficult and incurs a significant synchronization overhead. On the other hand, we must bear in mind that these languages were not designed to *support* sequential programming, but rather to *overcome* it. Objects in Vulcan and Polka are realized as syntactic extensions on the base of active objects as described in Section 4.2. A similar extension could be easily integrated in Small Oz.

11.3 Implicit Concurrency

A less radical approach to concurrency is to introduce concurrency implicitly when needed. This scheme was proposed by Smolka and was the concurrency model underlying the initial version of the Oz Programming Model [Smo95].

In Small Oz, if the topmost statement on the stack of a thread is synchronized and waits for a variable to become bound, then the whole thread suspends. In previous versions of Oz, which we adopt as another gedankenexperiment for this and the following section, the suspending statement is instead popped from the stack, and pushed on the stack of a newly created thread. The original thread can continue. Any synchronized statement can thus introduce concurrent computation. In particular, conditionals and procedure applications can produce a thread for a given statement *S* as in

```
local X in case X then S end X=true end
```

or in

```
local P in {P} proc {P} S end end
```

Therefore, we call this treatment of synchronized statements *implicit concurrency*. Implicit concurrency was (with slight variations) the underlying concurrency model of Oz 1. A fitting characterization of this strategy is given by Smolka [Smo95]: “The [...] reduction strategy tries to be as sequential as possible and as concurrent as necessary.” However, this strategy made it hard to control the concurrency created by a program and therefore was found to be inferior to the current model of *explicit concurrency* suppported by Oz 2.

11.4 Objects for Implicit Concurrency

For a model of object state with implicit concurrency, the following situation arises.

- The state notion for fine-grained concurrency described in the previous section becomes unnecessarily limiting. Typically, programs are written such that most synchronized statements do not lead to thread creation. Sequential execution is the default reduction technique in practice, which should be reflected in the state model.

- On the other hand, in the presence of synchronization, implicit concurrency generally makes stateful computation in the style of sequential programming hard to achieve.

The main idea for solving this dilemma is to impose a static order on the execution of stateful statements in methods. Since this static order does not necessarily coincide with the control flow, we need to enforce it at runtime, using data flow synchronization. As an example, consider the following method

```
meth m(X Y)
   case {P}
   then a <- X
   else a <- Y
   end
   b <- 2 * @a
end
```

To support the usual semantics of sequential state, we must make sure that the attribute access `@a` is executed after the assignment in the body of the conditional. This can be achieved by data-flow synchronization. For this purpose, the above method is compiled to the following procedure.

```
proc {$ Self Message In Out}
   Inter1 Inter2 A
in
   case Message of m(X Y) then
      case {P}
      then {StateAssign Self a X In Inter1}
      else {StateAssign Self a Y In Inter1}
      end
      {StateAccess Self a A   Inter1 Inter2}
      {StateAssign Self b 2*A Inter2 Out}
   end
end
```

Synchronization variables are "threaded" through the code in order to provide for data-flow synchronization. The procedures `StateAccess`, `StateAssign` and `StateExchange` get two further arguments for synchronization. The first one is used to wait for the previous state manipulation to be completed and the second one to signal completion to the next state manipulation. The corresponding procedure `StateAssign` is given in Program 11.1 (compare with the library procedure `StateAssign` given in Program 7.4).

Program 11.1 Data-Flow Synchronization for State Manipulation

```
proc {StateAssign Self Attr NewVal In Out}
   case In==unit andthen {IsLiteral Attr}
   then {Assign Self.OOState.Attr NewVal} In=Out
   end
end
```

Note that the primitive `IsLiteral` synchronizes on its argument to be bound and reduces to **`true`** if it is a literal.

Even if the procedure `P` in our method suspends, it is guaranteed that `@a` refers to the value of the attribute *after* execution of one of the bodies of the conditional. In this case, there will be three waiting actors, one for `StateAccess`, one for the multiplication and one for `StateAssign`.

We prevent concurrent object applications from manipulating the state; the operations on the state of an object are globally synchronized. This is achieved by equipping objects with a cell at feature `OOSync` that holds the synchronization token similar to the technique for mutual exclusion given in Section 9.2. The cell at feature `OOSync` is initialized with **`unit`** by object creation. The procedure `ObjectApply` threads the token through the message as shown in Program 11.2.

Program 11.2 Data-Flow Synchronization for Object Application

```
proc {ObjectApply Object Message}
   In Out in
   {Exchange Object.OOSync In Out}
   { {Lookup Object.OOClass {Label Message}}
     Object Message In Out}
end
```

In order to enforce sequentialization also for the non-state-using part of methods and to avoid a potential proliferation of actors, we can synchronize the application of methods on the synchronization token as in Program 11.3.

As in other languages with implictly synchronized objects, the problem of synchronized self application arises. Recall the discussion of this subject in Section 9.9 on page 142. The code fragment

```
{self m} a <- 1
```

always leads to a deadlock. We solve this problem by introducing a construct that allows to apply a method using the current synchronization token, as opposed to acquiring the token from **`self`**. Consider the following method

Program 11.3 Data-Flow Synchronization for Object Application (seq. version)

```
proc {ObjectApply Object Message}
   In Out in
   {Exchange Object.OOSync In Out}
   case In==unit
   then
      { {Lookup Object.OOClass {Label Message}}
        Object Message In Out}
   end
end
```

```
meth m(X Y)
   a <- X
   self; n(X)
   Y = @a
end
```

which is translated to

```
proc {$ Self Message In Out}
   case Message of m(X Y) then
      Inter1 Inter2 in
      {StateAssign Self a X    In     Inter1}
      {ThreadedApply Self n(X) Inter1 Inter2}
      {StateAccess Self a Y    Inter2 Inter3}
   end
end
```

We argue that the introduction of a new language construct for such a programming idiom is a cleaner solution than pretending that it is a special case object appliation as in Obliq. Note that in Oz 1, the construct `self;` was merged with the concept of method application.

11.5 Summary and Historical Perspective

We discussed the notion of concurrent objects in two alternative concurrency models, namely fine-grained concurrency and implicit concurrency. To match the spirit of fine-grained concurrent languages, a new notion of stateful programming is appropriate as shown by the languages Polka and Vulcan. For implicit concurrency, the notion of a sequential state can be recovered using data-flow synchronization.

In the 70s and 80s it was generally believed that soon massively parallel hardware would become widespread reality. Consequently, languages with fine-grained concurrency that could exploit such massive parallelism attracted considerable attention. However, in order to make effective use of available sequential or small-scale parallel hardware, the resulting programming concepts underwent considerable revision, eventually leading to coarse-grained concurrency. We give three examples for this pattern.

- The fine-grained concurrency of Hewitt's actors model of computation [Hew77] was revised by Yonezawa in the design of ABCL [Yon90], where the behavior of active objects is described by sequential Lisp-like procedures, using suitable synchronization primitives.

- Fine-grained concurrent data-flow languages [Den74] were merged with a more traditional computation framework by Iannucci [Ian88].

- Concurrent logic programming was conceived as a fine-grained concurrent programming model [Sha89]. Concurrent logic programming was adapted to existing small-scale parallel hardware by Foster and others in the development of PCN [FOT92], thus introducing thread-level concurrency.

Dorothy told the Witch all her story: how the cyclone had brought her to the Land of Oz, how she had found her companions, and of the wonderful adventures they had met with. "My greatest wish now," she added, "is to get back to Kansas, for Aunt Em will surely think something dreadful has happened to me, and that will make her put on mourning; and unless the crops are better this year than they were last, I am sure Uncle Henry cannot afford it."

Chapter: Glinda The Good Witch Grants Dorothy's Wish

Chapter 12

A Concurrent Meta-Object Protocol

Meta-object protocols allow to use object-oriented concepts not only to define the properties and behavior of objects, but also of their underlying object model. Meta-object protocols have been used in programming languages as tools for advanced application programming, for developing programming tools such as debuggers, and for language design. An example for a powerful meta-object protocol is provided by CLOS [KdRB91] where even central aspects of the object model such as the inheritance scheme can be customized using object-oriented programming.

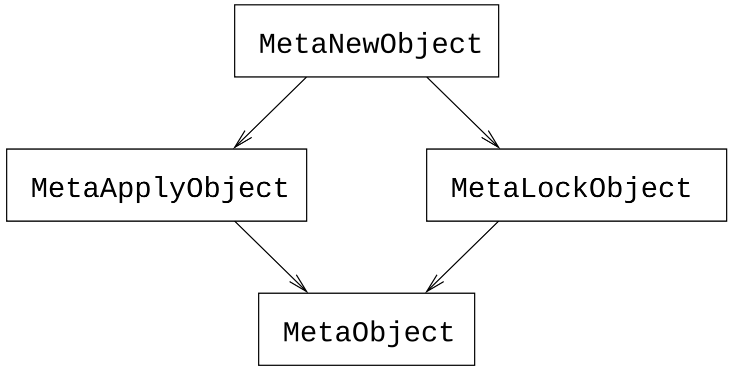
Our aim in this chapter is not to recreate such a protocol for Objects in Oz in general, but to show that it is possible to extend the ideas of a meta-object protocol to cover the concurrency model for objects. We show a simple meta-object protocol that allows to define classes for different concurrency models including implicitly synchronized objects and active objects. The intended applications for this protocol lie mainly in language design. The goal is to provide a system that allows to explore alternative concurrency concepts without manipulating the object library, compiler or runtime system.

12.1 Overall Design

The aim of our meta-object protocol is to let a class define which concurrency model its instances should adhere to. We let the class define what object creation, object/method application and locking means for its instances. Following a modular design, we define the following meta-classes.

- `MetaNewObject` allows to redefine object creation,
- `MetaApplyObject` provides access to the semantics of object and method application, and

Figure 12.1 The Structure of the Meta-Object Protocol



- `MetaLockObject` provides access to the locking mechanism.

The latter two classes depend in their functionality on the first class, and thus inherit from it. A forth class `MetaObject` inherits both classes `MetaApplyObject` and `MetaLockObject` and thus provides maximal flexibility. The inheritance relation between these meta-classes is depicted in Figure 12.1.

As a general design policy we insist on the following invariant. The meta-object protocol must not have any effect on classes that do not inherit from any of its meta-classes. This allows us to freely mix classes and objects of the standard model with objects and classes defined by meta-classes, and thus provide for maximal flexibility in experimentation.

Aspects that are less obviously related to concurrency such as class creation, inheritance, state and object features are not modifiable. For meta-object protocols that cover these aspects, we refer to CLOS and Smalltalk.

12.2 Object Creation

The meta-class `MetaNewObject` allows to redefine object creation. To this aim, we obviously need to redefine the procedure `New`. The class `MetaNewObject` and the new procedure `New` are given in Program 12.1.

The new procedure `New` checks if the class `C` defines the feature `MetaNew`. If so, it creates a standard object using the procedure `MakeInstance` given in Section 7.2 and applies the procedure at `C`'s feature `MetaNew` to this object, `C` and the initial message. If not, the standard object creation procedure `Object.new` is called. The class `MetaNewObject` represents the standard behavior of object creation.

For maximal flexibility, the object creation functionality of `MetaNewObject` is split between object creation as such and object initialization. We generally

Program 12.1 A Meta-Class for Object Creation

```
declare
MetaNew ={NewName}
MetaInit={NewName}
class MetaNewObject from BaseObject
   feat
      !MetaInit: proc {$ C WithMessage NewObject}
                    {NewObject WithMessage}
                 end
      !MetaNew : fun {$ C WithMessage NewObject}
                    {NewObject.MetaInit C
                     WithMessage NewObject}
                    NewObject
                 end
end
fun {New C WithMessage}
   case {Class.hasFeature C MetaNew}
   then
      {NewObject.MetaNew C WithMessage {MakeInstance C}}
   else
      {Object.new C WithMessage}
   end
end
```

use features for the meta-object protocol. A more obvious choice would be to use meta-methods, but we want to avoid interference of the meta-object protocol with object and method application that we are going to modify as well. Since we do not intend to modify the feature mechanism, features provide more stable grounds. Nevertheless, we are going to call these features *meta-methods*.

As a simple first example, consider the task to write a meta-class that counts how many instances are created from it and any class that inherits from it. We simply redefine the feature `MetaInit` in the following meta-class `Counting` to send a message to a `Counter` each time it is called.

```
class Counting from MetaNewObject
  feat
    !MetaInit: proc {$ C WithMessage NewObject}
                 {Counter inc}
                 {{Class.getFeature MetaNewObject MetaInit}
                  C WithMessage NewObject}
               end
end
```

The rest of the meta-method calls `MetaNewObject`'s meta-method `MetaInit`, and thus represents a (somewhat verbose) kind of super-call. The record `Class` represents a library module from which the procedure `Class.getFeature` is retrieved that provides access to non-free features of classes.

A more interesting task consists in modifying object creation such that `New` returns—instead of an ordinary object—a port which represents an active object serving messages sent to the port. The following meta-class `PortObject` does the job.

```
declare
class PortObject from MetaNewObject
   feat
      !MetaNew
      :
         fun {$ C WithMessage NewObject}
            Ms P={NewPort Ms}
         in
            {NewObject WithMessage}
            thread
               {ForAll Ms NewObject}
            end
            P
         end
end
```

Instead of returning the argument `NewObject` as in the the original meta-method `MetaNew`, this `MetaNew` returns a port. Thus, every instance created from `PortObject` is a port to an encapsulated object.

12.3 Method and Object Application

Similar to object creation, we provide appropriate library (and compiler) support to enable redefinition of object and method application. The meta-class `MetaApplyObject` in Program 12.2 represents the standard behavior of object and method application that can be modified through overriding.

Observe that the meta-method `MetaInit` is redefined such that it sets the private feature `MyClass` to the object's class. The meta-method `MetaObjAppl` implements object application by calling the meta-method for method application with the feature `MyClass` as class argument. The procedure `Lookup` is explained in Section 7.5.

We use the meta-class `MetaApplObject` to refine the meta-class `PortObject` such that object application uses the port, but method application uses the embed-

Program 12.2 A Meta-Class for Object and Method Application

```
declare
MetaMethAppl={NewName}
MetaObjAppl ={NewName}
class MetaApplObject from MetaNewObject
   feat
      MyClass
      !MetaObjAppl
      : proc {$ Self Message}
           {Self.MetaMethAppl Self.MyClass Self Message}
        end
      !MetaMethAppl
      : proc {$ C Self Message}
           {{Lookup C {Label Message}}
            Self Message}
        end
      !MetaInit
      : proc {$ C WithMessage NewObject}
           NewObject.MyClass = C
           {{Class.getFeature MetaNewObject MetaInit}
            C WithMessage NewObject}
        end
end
```

ded object. Furthermore, **`self`** in methods refers to the port instead of to the object. Program 12.3 shows the corresponding meta-class `Agent`.

Note that in contrast to the meta-class `PortObject`, we avoid using object application in the body of `MetaNew`; we leave to the reader to find out why.

12.4 Object Locking

To open up object locking, the standard object library is modified such that the feature `TheLock` is used as lock in **`lock`** ··· **`end`** instead of the usual implicit lock, if this feature is defined in the class of the current object. The class `MetaLockingObject` in Program 12.4 modifies object initialization such that this feature is bound to a lock, if the class has the property `locking`.

In the first example for the use of `MetaLockingObject` we provide for general implicit locking of object application. The meta-class `AlwaysLocking` in Program 12.5 inherits from `MetaObject` which in turn inherits both from `MetaApplObject` and `MetaLockingObject` as shown in Figure 12.1. The meta-method for object application is modified such that it acquires `TheLock`

Program 12.3 A Meta-Class for Agents

```
class Agent
   from MetaApplObject
   feat
      MyPort
      !MetaObjAppl
      : proc {$ Self Message}
           {Send Self.MyPort Message}
        end

      !MetaNew
      : fun {$ C WithMessage NewObject}
           Mr Ms=WithMessage|Mr
           P={NewPort Mr}
        in
           thread
              {ForAll Ms
               proc {$ M}
                  {Self.MetaMethAppl C NewObject M}
               end}
           end
           P
        end
end
```

Program 12.4 A Meta-Class for Locking

```
declare
TheLock={NewName}
class MetaLockingObject from MetaNewObject
   prop locking
   feat
      !TheLock
      !MetaInit
      : proc {$ C WithMessage NewObject}
           NewObject.TheLock=case {Class.isLocking C}
                             then {NewLock}
                             else ´no property locking´
                             end
        end
end
```

Program 12.5 A Meta-Class for Implicit Locking

```
class AlwaysLocking from MetaObject
   feat !MetaObjAppl
        : proc {$ Self Message}
              lock Self.TheLock
              then
                 {{Class.getFeature
                   MetaApplyObject MetaObjApply}
                  C WithMessage NewObject}
end        end end
```

before the inherited meta-method is called.

For the second example, consider a situation where several objects must be locked to perform a complex operation on them. If locking is done without care, deadlocks can occur when two threads start locking objects that both operations use. A classical deadlock prevention technique is to impose a total ordering on the objects involved in such operations and make sure that the process of acquiring the locks follows this ordering [Hav68]. This technique is called *hierarchical locking*. We provide a meta-class that encapsulates this protocol in Program 12.6. The meta-class `HierarchicalLocking` refines the meta-method `MetaNew` such that an integer is bound to the feature `LockId`. The integers are generated by a `IdServer` in increasing order. The procedure `LockAll` takes `Objects` that must inherit from `HierarchicalLocking`, sorts them according to their `LockId`, locks them in this order and applies a given procedure `P`.

12.5 Discussion and Related Work

We showed that a modified object library can support a powerful meta-object protocol, geared towards experimenting with alternative or additional concurrency models for concurrent objects. Surprisingly, this is the first meta-object protocol that covers a variety of concurrency models ranging from completely sequential to active objects. Again, it is the feature of first-class messages that allows us to integrate active objects. The meta-classes given in this chapter are idealized versions of an experimental meta-object system for Oz described in [Hen97a].

The meta-object protocols of CLOS and Smalltalk allow to integrate synchronization techniques in the object system, but fail to provide for active objects due to the lack of first-class messages. Watanabe and Yonezawa [WY90] describe a system based on ABCL that allows to reflect components of active objects, such as their message queue, back into another active object which they call its “meta-object”. Such reflection techniques provide a subset of the techniques possible for

Program 12.6 A Meta-Class for Hierarchical Locking

```
local
    LockId={NewName}
    proc {LockAll1 Os P}
       case Os
       of O|Or then
          lock O.TheLock then {LockAll1 Or P} end
       [] nil then {P}
       end
    end
    IdServer={New class from BaseObject
                     attr id:0
                     meth id(NewId)
                        Id = id <- NewId in NewId=Id+1
                 end end
              noop}
in
    proc {LockAll Objects P}
       {LockAll1
        {Sort Os fun {$ X Y} X.LockId < Y.LockId end}
        P}
    end
    class HierarchicalLocking from MetaLockingObject
        feat !LockId
             !MetaNew
             : fun {$ C WithMessage NewObject}
                  NewObject.LockId={IdServer newId($)}
                  {{Class.getFeature MetaNewObject MetaNew}
                  C WithMessage NewObject}
end end        end
```

meta-object protocols. They are restricted by their inherent descriptive nature, as opposed to the prescriptive nature of meta-object protocols.

The locking scheme of Java is similar to ours. Java provides first-class access to the implicit lock of objects, and thus allows to express the techniques described in Section 12.4. Java does not provide any meta-object facilities; all Java classes are instances of a fixed class `Class` which only provides some debugging and self-documentation functionality.

Extensions of Eiffel such as Eiffel|| [Car93] and Karaorman and Bruno's Eiffel extension [KB93] achieve active objects using a combination of library and compiler support. A suitably modified Eiffel compiler recognizes inheritance from a fixed class (`PROCESS` in Eiffel|| and `CONCURRENCY` in Karaorman and Bruno's

Eiffel dialect) and modifies object creation of instances of this class to yield active objects. However, it is not possible to modify these classes through inheritance.

POOL-T [Ame87] allows to customize the behavior of active objects by defining suitable "object bodies", but enforces explicit message receipt, which excludes the possibility to encode passive objects.

All you have to do is to knock the heels together three times and command the shoes to carry you wherever you wish to go."

"If that is so," said the child joyfully, "I will ask them to carry me back to Kansas at once."

She threw her arms around the Lion's neck and kissed him, patting his big head tenderly. Then she kissed the Tin Woodman, who was weeping in a way most dangerous to his joints. But she hugged the soft, stuffed body of the Scarecrow in her arms instead of kissing his painted face, and found she was crying herself at this sorrowful parting from her loving comrades.

Glinda the Good stepped down from her ruby throne to give the little girl a good-bye kiss, and Dorothy thanked her for all the kindness she had shown to her friends and herself.

Dorothy now took Toto up solemnly in her arms, and having said one last good-bye she clapped the heels of her shoes together three times, saying:

"Take me home to Aunt Em!"

Chapter: Glinda The Good Witch Grants Dorothy's Wish

Chapter 13

Conclusion

In this dissertation, we showed that the investment into a particular collection of advanced language and system features pays off by providing a base for a powerful object-oriented programming and system. We briefly review these investments and the corresponding returns in Section 13.1. Section 13.2 gives a summary of the dissertation.

13.1 The Investments and their Returns

Higher-Order Programming. Lexically scoped first-class procedures are provided by functional programming languages, and object-oriented languages such as Smalltalk (in the form of blocks) and Java (by "inner" classes in its recent version). The use of first-class procedures to define an object system was pioneered by object-oriented Lisp-extensions such as Flavors and CLOS. In this presentation, lexically scoped higher-order programming provided the key to advanced object-oriented techniques such as full compositionality of classes and classes as first-class values (Section 6.9), and higher-order programming with state (Section 6.7). Furthermore, first-class procedures allowed a simple reduction of objects to Small Oz in Chapter 7 which can be seen as their semantic foundation.

Cells. Stateful data in the form of cells are an obvious ingredient of object-oriented programming. It is surprising that concurrent logic programming languages struggled for a long time to achieve concurrent object-oriented programming concepts without cells, leading to awkward semantic and syntactic constructions and failing to express basic sequential object-oriented programming as discussed in Sections 3.5 and 10.1. In fact, Oz is the first concurrent language with logic variables that readily supports sequential object-oriented programming. From the perspective of concurrent logic programming, the main prerequisite for this were cells and the replacement of concurrent by sequential composition as the main composition operator.

Names. Names bound to lexically scoped variables allowed in Section 6.5 to express important object-oriented idioms such as private and protected methods.

Logic Variables. Logic variables powered concurrent programming techniques such as data-driven synchronization (Section 10.3), and provided—together with cells—a wide variety of other synchronization techniques (Chapter 9).

Thread-Level Concurrency. Part II describes a conventional object-oriented system. In Chapter 11, we showed that conventional object-oriented programming is much harder to obtain in alternative concurrency models such as fine-grained concurrency.

Abstract Machine. The abstract machine for Oz proved in Chapter 8 to provide the flexibility needed to efficiently integrate object-oriented programming in an existing implementation.

13.2 Summary

Lea [Lea97] notes that "research on concurrency sometimes relies on models and techniques that are ill-suited for everyday object-oriented software development." So far, this was certainly the case for object-oriented programming in concurrent logic languages (see [JMH93] for a thorough discussion). This dissertation can be seen as an attempt to bridge this apparent gap between concurrent language research and programming practice. To this aim, we proceeded in two steps. Firstly, we designed a conventional object system for the thread-based concurrent constraint language Oz, enabling sequential object-oriented programming in a concurrent constraint language. We described this object system, its semantic foundation and implementation (Part II). Secondly, we explored concurrent programming from the perspective of this conventional object system, making use of thread-level concurrency and logic variables (Part III).

13.2.1 Conventional Objects in Concurrent Constraint Programming

In Chapters 5 and 6, we presented a simple yet powerful object-system on the base of Small Oz, a variant of Oz. First-class messages, attribute and method identifiers allowed to express a variety of interesting programming techniques. The semantic foundation of Objects in Oz was provided by a reduction to Small Oz in Chapter 7. We showed that names together with lexical scoping readily support concepts such as private and protected identifiers.

We showed in Chapter 8 that surgical operations on the instruction set of an abstract machine for Oz can yield an efficient implementation of Objects in Oz. We gave the first detailed account of how to integrate object-oriented concepts efficiently into the abstract machine of a high-level language.

13.2.2 Concurrent Programming with Objects in Oz

In Chapter 9, we exploited the expressivity of logic variables together with cells for a wide variety of synchronization techniques, ranging from data-driven synchronization over mutual exclusion, readers/writer synchronization, to thread-reentrant locks. We showed in Chapter 10 that active objects can be readily supported by making use of first-class messages. First-class messages also play a crucial role in the concurrent meta-object protocol presented in Chapter 12, a language design tool in which object-oriented programming can be used to define the concurrency model of objects. Chapter 11 examined the impact of alternative concurrency models on the design of suitable object systems.

13.3 Beyond Objects in Oz

To a researcher in programming languages, more interesting than the fate of individual languages such as Oz is the fate of their underlying concepts. We provided strong evidence that a conventional object system in a language with thread-level concurrency can benefit greatly from synchronization with logic variables and from first-class messages. On the base of this evidence we conclude that these features should be included in future concurrent object-oriented languages.

Bibliography

[AC96] Martín Abadi and Luca Cardelli. *A Theory of Objects*. Monographs in Computer Science. Springer-Verlag, Berlin, 1996.

[AG96] Ken Arnold and James Gosling. *The Java Programming Language*. The Java Series. Addison-Wesley, Reading, MA, 1996.

[Agh86] Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. The MIT Press, Cambridge, MA, 1986.

[AK91] Hassan Aït-Kaci. *Warren's Abstract Machine: A Tutorial Reconstruction*. Logic Programming. The MIT Press, Cambridge, MA, 1991.

[Ame87] Pierre America. POOL-T: A parallel object-oriented language. In Akinori Yonezawa and Mario Tokoro, editors, *Object-Oriented Concurrent Programming*, pages 199–220. The MIT Press, Cambridge, MA, 1987.

[AS96] Harold Abelson and Gerald Sussman. *Structure and Interpretation of Computer Programs*. The MIT Press, Cambridge, MA, second edition, 1996.

[AWY93] Gul Agha, Peter Wegner, and Akinori Yonezawa. *Research Directions in Concurrent Object-Oriented Programming*. The MIT Press, Cambridge, MA, 1993.

[Bac87] John Backus. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. In Robert Ashenhurst and Susan Graham, editors, *ACM Turing Award Lectures : The First 20 Years (1966-1985)*, Anthology series. The ACM Press, New York, 1987. 1977 Turing Award Lecture.

[Bak93] Henry Baker. Equal rights for functional objects or, the more things change, the more they are the same. *ACM SIGPLAN OOPS Messenger: Object-Oriented Programming Systems*, 4(4):2–27, 1993.

[Bal91] Henri Bal. A comparative study of five parallel programming languages. In *EurOpen Spring Conference on Open Distributed Systems in Perspective*, pages 209–228, Tromso, Norway, 1991.

[Bau00] Frank Baum. *The Wonderful Wizard of Oz*. Reilly and Lee, Chicago, 1900.

[BC93] Frédéric Benhamou and Alain Comerauer, editors. *Constraint Logic Programming*. The MIT Press, Cambridge, MA, 1993. Selected Research.

[BH77] Henry Baker and Carl Hewitt. The incremental garbage collection of processes. A.I. Memo No. 454, Massachusetts Institute of Technology, Cambridge, MA, December 1977.

[BHJL86] Andrew Black, Norman Hutchinson, Eric Jul, and Henry Levy. Object structure in the Emerald system. In Norman Meyrowitz, editor, *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages and Applications*, pages 78–86, Portland, Oregon, 1986. ACM SIGPLAN Notices 21(11).

[Boo94] Grady Booch. *Object-Oriented Analysis and Design with Applications*. Benjamin/Cummings Publishing, Redwood City, CA, second edition, 1994.

[C$^+$93] Derek Coleman et al. *Object-Oriented Development: The Fusion Method*. Prentice Hall, Englewood Cliffs, NJ, 1993.

[Car93] Denis Caromel. Toward a method of object-oriented concurrent programming. In Concurrent Object-Oriented Programming [Con93], pages 90–102.

[Car94] Luca Cardelli. Obliq: A language with distributed scope. Technical Report 122, Digital Equipment Corporation, Systems Research Center, Palo Alto, CA, June 1994.

[Car95] Luca Cardelli. A language with distributed scope. *Computing Systems*, 8(1):27–59, January 1995.

[CG81] Keith Clark and Steve Gregory. A relational language for parallel programming. In *Symposium on Functional Languages and Computer Architecture*, pages 171–178, Aspenäs, Sweden, 1981. University of Göteborg.

[CHP71] Pierre-Jacques Courtois, F. Heymans, and David Parnas. Concurrent control with readers and writers. *Communications of the ACM*, 14(10):667–668, October 1971.

[CKPR73] Alain Colmerauer, Henri Kanoui, R. Psero, and Philippe Roussel. Un système de communication homme-machine en français. Technical report, Université Aix-Marseille II, Marseille, France, 1973.

[Con93] Concurrent Object-Oriented Programming. Special Issue of *Communications of the ACM*, 36(9), September 1993.

[CPL83] Thomas Conroy and Eduardo Pelegri-Llopart. An assessment of method-lookup caches for Smalltalk-80 at Hewlett-Packard. In Krassner [Kra83], pages 239–248.

[CW85] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–522, December 1985.

[Dav89] Andrew Davison. *Polka: A Parlog Object-Oriented Language*. PhD thesis, Imperial College, London, 1989.

[Den74] Jack Dennis. First version of a data flow procedure language. In Bernhard Robinet, editor, *Proceedings Colloque sur la Programmation*, Lecture Notes in Computer Science 19, pages 362–376, Paris, 1974. Springer-Verlag, Berlin.

[Dij68] Edsger Dijkstra. Cooperating sequential processes. In F. Genuys, editor, *Programming Languages*. Academic Press, London, 1968.

[DN66] Ole-Johan Dahl and Kristen Nygaard. Simula, an Algol-based simulation language. *Communications of the ACM*, 9(9):671–678, 1966.

[Dri93a] Karel Driesen. Method lookup strategies in dynamically typed object-oriented programming languages. Master's thesis, Vrije Universiteit Brussel, 1993.

[Dri93b] Karel Driesen. Selector table indexing and sparse arrays. In Andreas Paepcke, editor, *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages and Applications*, pages 259–270, Washington, D.C., 1993. ACM SIGPLAN Notices 28(10).

[DS83] Peter Deutsch and Allan Schiffman. Efficient implementation of the Smalltalk-80 system. In Alan Demers, editor, *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 297–302, Austin, TX, 1983. The ACM Press, New York.

[FMP95] Klaus Fischer, Jörg Müller, and Markus Pischel. A model for cooperative transportation scheduling. In *Proceedings of the International Conference on Multiagent Systems (ICMAS)*, San Francisco, 1995. The MIT Press, Cambridge, MA.

[FOT92] Ian Foster, Robert Olson, and Steven Tuecke. Productive parallel programming: The PCN approach. *Scientific Programming*, 1(1):55–66, 1992.

[FWH92] Daniel Friedman, Mitchell Wand, and Christopher Haynes. *Essentials of Programming Languages*. The MIT Press, Cambridge, MA, 1992.

[GK76] Adele Goldberg and Alan Kay, editors. Smalltalk-72 instructional manual. Technical Report SSL-76-6, Xerox PARC, Palo Alto, CA, March 1976.

[GR83] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, Reading, MA, 1983.

[Hal85] Robert Halstead. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, 1985.

[Hav68] J. W. Havender. Avoiding deadlock in multitasking systems. *IBM Systems Journal*, 7(2):74–84, 1968.

[HB77] Carl Hewitt and Henry Baker. Laws for communicating parallel processes. In Bruce Gilchrist, editor, *Proceedings of the World Computer Congress of the IFIP*, pages 987–992, Toronto, Canada, August 1977. North-Holland, Amsterdam.

[HCU91] Urs Hölzle, Craig Chambers, and David Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In Pierre America, editor, *Proceeedings of the European Conference on Object-Oriented Programming*, Lecture Notes in Computer Science 512, pages 21–38, Geneva, Switzerland, 1991. Springer-Verlag, Berlin.

[Hen97a] Martin Henz. Objects in Oz—the programs of the thesis. `http://www.ps.uni-sb.de/~henz/diss/oz/`, 1997.

[Hen97b] Martin Henz. The Oz notation. DFKI Oz documentation series, German Research Center for Artificial Intelligence (DFKI), Stuhlsatzenhausweg 3, D-66123 Saarbrücken, Germany, 1997.

[Hew77] Carl Hewitt. Viewing control structures as patterns of message passing. *Journal of Artificial Intelligence*, 8(3):323–364, 1977.

[Hoa72] Charles A. R. Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, 17(10):549–557, October 1972.

[HS94] Martin Henz and Gert Smolka. Objects in higher-order concurrent constraint programming with state. In *Workshop on Coordination Models and Languages for Parallelism and Distribution (in connection with ECOOP 94)*, Bologna, Italy, 1994.

[HSW93] Martin Henz, Gert Smolka, and Jörg Würtz. Oz—a programming language for multi-agent systems. In Ruzena Bajcsy, editor, *Proceedings of the International Joint Conference on Artificial Intelligence*, volume 1, pages 404–409, Chambéry, France, 1993. Morgan Kaufmann Publishers, San Mateo, CA.

[HW66] Charles A. R. Hoare and Nikolaus Wirth. Contribution to the development of ALGOL 60. *Communications of the ACM*, 9(6):413–432, 1966.

[Ian88] Robert Iannucci. *Dataflow/von Neumann Hybrid Architecture*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, 1988.

[Ing78] Dan Ingalls. The Smalltalk-76 programming system design and implementation. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, Tuscon, AZ, 1978. The ACM Press, New York.

[Jan94] Sverker Janson. *AKL—A Multiparadigm Programming Language*. PhD thesis, SICS Swedish Institute of Computer Science, SICS Box 1263, S-164 28 Kista, Sweden, 1994.

[JM94] Joxan Jaffar and Michael Maher. Constraint logic programming—a survey. *Journal of Logic Programming*, 19/20:503–582, 1994.

[JMH93] Sverker Janson, Johan Montelius, and Seif Haridi. Ports for objects. In *Research Directions in Concurrent Object-Oriented Programming*. The MIT Press, Cambridge, MA, 1993.

[Kah74] Gilles Kahn. The semantics of a simple language for parallel processes. In *Proceedings of the World Computer Congress of the IFIP*, pages 471–475, Stockholm, Sweden, 1974. North-Holland, Amsterdam.

[Kah89] Kenneth Kahn. Objects: A fresh look. In Stephen Cook, editor, *Proceeedings of the European Conference on Object-Oriented Programming*, pages 207–223, Nottingham, UK, 1989. Cambridge University Press, Cambridge, MA.

[Kah96] Kenneth Kahn. ToonTalk—an animated programming environment for children. *Journal of Visual Languages and Computing*, June 1996.

[KB93] Murat Karaorman and John Bruno. Introducing concurrency to a sequential language. In Concurrent Object-Oriented Programming [Con93], pages 103–116.

[KdRB91] Gregor Kiczales, Jim des Rivières, and Daniel Bobrow. *The Art of the Metaobject Protocol*. The MIT Press, Cambridge, MA, 1991.

[KM77] Gilles Kahn and David MacQueen. Coroutines and networks of parallel processes. In Bruce Gilchrist, editor, *Proceedings of the World Computer Congress of the IFIP*, pages 993–998, Toronto, Canada, August 1977. North-Holland, Amsterdam.

[KMMPN83] Bent Bruun Kristensen, Ole Lehrmann Madsen, Birger Moller-Pedersen, and Kristen Nygaard. Abstraction mechanisms in the Beta programming language. In Alan Demers, editor, *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 285–298, Austin, TX, 1983. The ACM Press, New York.

[Kow74] Robert Kowalski. Predicate logic as a programming language. In Jack Rosenfeld, editor, *Proceedings of the World Computer Congress of the IFIP*, Stockholm, Sweden, 1974. North-Holland, Amsterdam.

[Kra83] Glenn Krassner, editor. *Smalltalk-80: Bits of History and Words of Advice*. Addison-Wesley, Reading, MA, 1983.

[KTMB86] Kenneth Kahn, Eric Tribble, Mark Miller, and Daniel Bobrow. Objects in concurrent logic programming languages. In Norman Meyrowitz, editor, *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages and Applications*, pages 242–257, Portland, Oregon, 1986. ACM SIGPLAN Notices 21(11).

[KTMB87] Kenneth Kahn, Eric Tribble, Mark Miller, and Daniel Bobrow. Vulcan: Logical concurrent objects. In Ehud Shapiro, editor, *Concurrent Prolog*, Series in Logic Programming, pages 275–303. The MIT Press, Cambridge, MA, 1987.

[Lan63] Peter Landin. The mechanical evaluation of expressions. *Computer Journal*, 6(4):308–320, 1963.

[Lan65] Peter Landin. A correspondence between ALGOL 60 and Church's lambda-notation: Part I. *Communications of the ACM*, 8(2):89–101, 1965.

[Lea97] Doug Lea. *Concurrent Programming in Java*. The Java Series. Addison-Wesley, Reading, MA, 1997.

[LL94] Cristina Videira Lopez and Karl Lieberherr. Abstracting process-to-function relations in concurrent object-oriented applications. In Paolo Ciancarini, Oscar Nierstrasz, and Akinori Yonezawa, editors, *Proceeedings of the European Conference on Object-Oriented Programming*, Lecture Notes in Computer Science 821, pages 81–99, Bologna, Italy, 1994. Springer-Verlag, Berlin.

[Löh93] Klaus-Peter Löhr. Concurrency annotations for reusable software. In Concurrent Object-Oriented Programming [Con93], pages 81–89.

[Mah87] Michael Maher. Logic semantics for a class of committed-choice programs. In Jean-Louis Lassez, editor, *Proceedings of the International Conference on Logic Programming*, pages 858–876, Melbourne, Australia, 1987. The MIT Press, Cambridge, MA.

[McH94] Ciaran McHale. *Synchronization in Concurrent Object-Oriented Languages: Expressive Power, Genericity and Inheritance*. PhD thesis, University of Dublin, Trinity College, Dublin, Ireland, 1994.

[Mes93] José Meseguer. Solving the inheritance anomaly in concurrent object-oriented programming. In Oscar Nierstrasz, editor, *Proceeedings of the European Conference on Object-Oriented Programming*, Lecture Notes in Computer Science 707, pages 220–246, Kaiserslautern, Germany, 1993. Springer-Verlag, Berlin.

[Mey92] Bertrand Meyer. *Eiffel: The Language*. Object-Oriented Series. Prentice Hall, Englewood Cliffs, NJ, 1992.

[Mey93] Bertrand Meyer. Systematic concurrent object-oriented programming. In Concurrent Object-Oriented Programming [Con93], pages 56–80.

[Mil93] Robin Milner. The polyadic π-calculus: A tutorial. In Friedrich Bauer, Wilfried Brauer, and Helmut Schwichtenberg, editors, *Proceedings of the Summer School on Logic and Algebra of Specification*, Marktoberndorf, Germany, 1993. Springer-Verlag, NATO ASI Series, Berlin.

[Moo86] David Moon. Object-oriented programming with Flavors. In Norman Meyrowitz, editor, *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages and Applications*, pages 1–8, Portland, Oregon, September 1986. ACM SIGPLAN Notices 21(11).

[Mos94] Chris Moss. *Prolog++. The Power of Object-Oriented and Logic Programming*. International Series in Logic Programming. Addison-Wesley, Reading, MA, 1994.

[MSS95] Michael Mehl, Ralf Scheidhauer, and Christian Schulte. An abstract machine for Oz. In Manuel Hermenegildo and Doaitse Swierstra, editors, *International Symposium on Programming Languages: Implementations, Logics and Programs*, Lecture Notes in Computer Science 982, pages 151–168, Utrecht, The Netherlands, 1995. Springer-Verlag, Berlin.

[MY93] Satoshi Matsuoka and Akinori Yonezawa. Analysis of inheritance anomaly in object-oriented concurrent programming languages. In Gul Agha, Peter Wegner, and Akinori Yonezawa, editors, *Research Directions in Object-Based Concurrency*. The MIT Press, Cambridge, MA, 1993.

[N$^+$63] Peter Naur et al. Revised report on the algorithmic language ALGOL 60. *Communications of the ACM*, 6(1):1–20, 1963.

[Nie94] Joachim Niehren. *Funktionale Berechnung in einem Uniform Nebenläufigen Kalkül mit Logischen Variablen*. Doctoral Dissertation, Universität des Saarlandes, Fachbereich Informatik, Im Stadtwald, 66041 Saarbrücken, Germany, December 1994.

[Ped89] Claus Pedersen. Extending ordinary inheritance schemes to include generalization. In Norman Meyrowitz, editor, *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages and Applications*, pages 407–418. The ACM Press, New York, 1989.

[Pop95] Konstantin Popov. An exercise in concurrent object-oriented programming: Oz Browser. In *WOz'95, International Workshop on Oz Programming*, Institut Dalle Molle d'Intelligence Artificielle Perceptive, Martigny, Switzerland, 1995.

[Pop97] Konstantin Popov. *Exploiting Thread-Level Concurrency for Parallelism in a Constraint Language*. Doctoral Dissertation, Universität des Saarlandes, Fachbereich Informatik, Im Stadtwald, 66041 Saarbrücken, Germany, 1997. In preparation.

[PS94] Jens Palsberg and Michael Schwartzbach. *Object-Oriented Type Systems*. John Wiley & Sons, New York, 1994.

[PZC95] John Plevyak, Xingbin Zhang, and Andrew Chien. Obtaining sequential efficiency for concurrent object-oriented languages. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 311–321, San Francisco, CA, 1995. The ACM Press, New York.

[Ren82] Tim Rentsch. Object-oriented programming. *SIGPLAN Notices*, 17(12):51, 1982.

[RMS96] Peter Van Roy, Michael Mehl, and Ralf Scheidhauer. Integrating efficient records into concurrent constraint programming. In *International Symposium on Programming Languages, Implementations, Logics, and Programs*, Aachen, Germany, 1996. Springer-Verlag, Berlin.

[RR93] Anthony Ralston and Edwin Reilly, editors. *Encyclopedia of Computer Science*. Van Nostrand Reinhold, New York, third edition, 1993.

[RV97] Didier Rémy and Jérôme Vouillon. Objective ML: A simple object-oriented extension of ML. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 40–53, Paris, January 1997. The ACM Press, New York.

[Sar85] Vijay Saraswat. Partial correctness semantics for CP$[\downarrow, |, \&]$. In S.N. Meheshwari, editor, *Proceedings of the Conference on Foundations of Software Technology and Theoretical Computer Science*, Lecture Notes in Computer Science 206, pages 347–368, New Delhi, India, 1985. Springer-Verlag, Berlin.

[Sar93] Vijay Saraswat. *Concurrent Constraint Programming*. The MIT Press, Cambridge, MA, 1993.

[Sch97] Christian Schulte. Oz Explorer: A visual constraint programming tool. In Lee Naish, editor, *Proceedings of the International Conference on Logic Programming*, Leuven, Belgium, 1997. The MIT Press, Cambridge, MA. To appear.

[SG93] Guy Steele and Richard Gabriel. The evolution of lisp. In Richard Wexelblat, editor, *Proceedings of the ACM SIGPLAN History of Programming Languages Conference*, pages 231–270, Cambridge, MA, 1993. ACM SIGPLAN Notices 28(3).

[Sha89] Ehud Shapiro. The family of concurrent logic programming languages. *ACM Computing Surveys*, 21(3):413–511, September 1989.

[SHW93] Gert Smolka, Martin Henz, and Jörg Würtz. Object-oriented concurrent constraint programming in Oz. Research Report RR-93-16, German Research Center for Artificial Intelligence (DFKI), Stuhlsatzenhausweg 3, D-66123 Saarbrücken, Germany, April 1993.

[SHW95] Gert Smolka, Martin Henz, and Jörg Würtz. Object-oriented concurrent constraint programming in Oz. In Pascal van Hentenryck and Vijay A. Saraswat, editors, *Principles and Practice of Constraint Programming*, pages 29–48. The MIT Press, Cambridge, MA, 1995.

[SIC96] Intelligent Systems Laboratory, SICS, PO Box 1263, 164 28 Kista, Sweden. *SICStus Prolog User's Manual, Release 3 #5*, October 1996.

[Sin94] Ghan Bir Singh. Single versus multiple inheritance in object oriented programming. *ACM SIGPLAN OOPS Messenger: Object-Oriented Programming Systems*, 5(1):34–43, 1994.

[Smi80] Reid Smith. The contract net protocol: High-level communication and control in a distributed problem solver. *IEEE Transactions on Computers*, C-29(12):1104–1113, December 1980.

[Smo91] Gert Smolka. Hydra: Constraint-based computation and deduction. Project Proposal, German Research Center for Artificial Intelligence (DFKI) Stuhlsatzenhausweg 3, D-66123 Saarbrücken, Germany, April 1991.

[Smo95] Gert Smolka. The Oz programming model. In Jan van Leeuwen, editor, *Computer Science Today*, Lecture Notes in Computer Science 1000, pages 324–343. Springer-Verlag, Berlin, 1995.

[Smo97] Gert Smolka. An Oz primer. DFKI Oz documentation series, German Research Center for Artificial Intelligence (DFKI), Stuhlsatzenhausweg 3, D-66123 Saarbrücken, Germany, 1997.

[SO93] Heinz Schmidt and Stephen Omohundro. CLOS, Eiffel, and Sather: A comparison. In Andreas Paepcke, editor, *Object-Oriented Programming: The CLOS Perspective*, pages 181–213. The MIT Press, Cambridge, MA, 1993.

[SS75] Gerald Sussman and Guy Steele. Scheme: An interpreter for extended lambda calculus. A.I. Memo No. 349, Massachusetts Institute of Technology, Cambridge, MA, December 1975.

[SS94] Christian Schulte and Gert Smolka. Encapsulated search in higher-order concurrent constraint programming. In Maurice Bruynooghe, editor, *Proceedings of the International Symposium on Logic Programming*, pages 505–520, Ithaca, NY, 1994. The MIT Press, Cambridge, MA.

[SSW94] Christian Schulte, Gert Smolka, and Jörg Würtz. Encapsulated search and constraint programming in Oz. In Alan Borning, editor, *Second Workshop on Principles and Practice of Constraint Programming*, Lecture Notes in Computer Science 874, pages 134–150, Rosario, Orcas Island, WA, 1994. Springer-Verlag, Berlin.

[ST83] Ehud Shapiro and Akikazu Takeuchi. Object oriented programming in Concurrent Prolog. *New Generation Computing*, 1:24–48, 1983.

[ST94] Gert Smolka and Ralf Treinen. Records for logic programming. *Journal of Logic Programming*, 18(3):229–258, April 1994.

[ST97] Gert Smolka and Ralf Treinen. DFKI Oz documentation series. `http://www.ps.uni-sb.de/oz/`, Programming Systems Lab, German Research Center for Artificial Intelligence (DFKI), Stuhlsatzenhausweg 3, D-66123 Saarbrücken, Germany, 1997.

[Ste76] Guy Steele. Lambda: The ultimate declarative. A.I. Memo No. 379, Massachusetts Institute of Technology, Cambridge, MA, November 1976.

[Ste90] Guy Steele. *Common LISP: The Language*. Digital Press, second edition, 1990.

[Str87] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, MA, 1987.

[Tai96] Antero Taivalsaari. On the notion of inheritance. *ACM Computing Surveys*, 28(3):438–479, September 1996.

[TMY93] Kenjiro Taura, Satoshi Matsuoka, and Akinori Yonezawa. An efficient implementation scheme of concurrent object-oriented languages on stock multicomputers. In Marina Chen, editor, *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 218–228, Charleston, SC, 1993. The ACM Press, New York.

[Ung86] David Ungar. *The Design and Evaluation of a High Performance Smalltalk System*. The MIT Press, Cambridge, MA, 1986.

[UP83] David Ungar and David Patterson. Berkeley Smalltalk: Who knows where the time goes? In Krassner [Kra83], pages 189–206.

[US87] David Ungar and Randall B. Smith. Self: The power of simplicity. In Norman Meyrowitz, editor, *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages and Applications*, pages 227–242, Orlando, FL, 1987. The ACM Press, New York.

[VH94] Jan Vitek and Nigel Horspool. Taming message passing: Efficient method look-up for dynamically typed languages. In Paolo Ciancarini, Oscar Nierstrasz, and Akinori Yonezawa, editors, *Proceeedings of the European Conference on Object-Oriented Programming*, Lecture Notes in Computer Science 821, pages 432–449, Bologna, Italy, 1994. Springer-Verlag, Berlin.

[Wad90] Philip Wadler. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science*, 73:231–248, 1990.

[War83] David H. D. Warren. An abstract Prolog instruction set. Technical Note 309, SRI International, Artificial Intelligence Center, Menlo Park, CA, October 1983.

[WL96] Gregory Wilson and Paul Lu. *Parallel Programming Using C++*. The MIT Press, Cambridge, MA, 1996. Foreword "A Perspective on Concurrency and C++" by Bjarne Stroustrup.

[WY90] Takuo Watanabe and Akinori Yonezawa. Reflection in an object-oriented concurrent language. In Akinori Yonezawa, editor, *ABCL: An Object-Oriented Concurrent System*, pages 45–70. The MIT Press, Cambridge, MA, 1990.

[YBS86] Akinori Yonezawa, Jean-Pierre Briot, and Etsuya Shibayama. Object-oriented concurrent programming in ABCL/1. In Norman Meyrowitz, editor, *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages and Applications*, pages 258–268, Portland, Oregon, 1986. ACM SIGPLAN Notices 21(11).

[YC88] Kaoru Yoshida and Takashi Chikayama. A'UM—a stream-based concurrent object-oriented language. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 638–649, Tokyo, Japan, 1988. Springer-Verlag, Berlin.

[Yon90] Akinori Yonezawa, editor. *ABCL: An Object-Oriented Concurrent System*. Series in Computer Systems. The MIT Press, Cambridge, MA, 1990.

[YT87] Yasuhiko Yokote and Mario Tokoro. Concurrent programming in ConcurrentSmalltalk. In Akinori Yonezawa and Mario Tokoro, editors, *Object-Oriented Concurrent Programming*, pages 129–158. The MIT Press, Cambridge, MA, 1987.

Index

Colophon

This document was typeset using the typesetting system LaTeX 2ε. The typeset document was translated into PostScript using `dvips` and printed on an HP LaserJet 4Si. The typeface used for the main text is twelve point Times. The bibliography was prepared with BibTeX. The Oz programs were converted to LaTeX using the tool `raw2tex` written by Denys Duchier. All figures were drawn using the tool `xfig`, which also translated them into PostScript. The pictures on the title pages of the three parts were drawn by Andreas Schoch. The quotations in front of every chapter are taken—you may have guessed it—from the book "The Wizard of Oz" [Bau00]. We used the online version of the book provided by Project Gutenberg at `http://www.promo.net/pg/` and printed the excerpts in typeface twelve point Computer Modern Fibonacci.