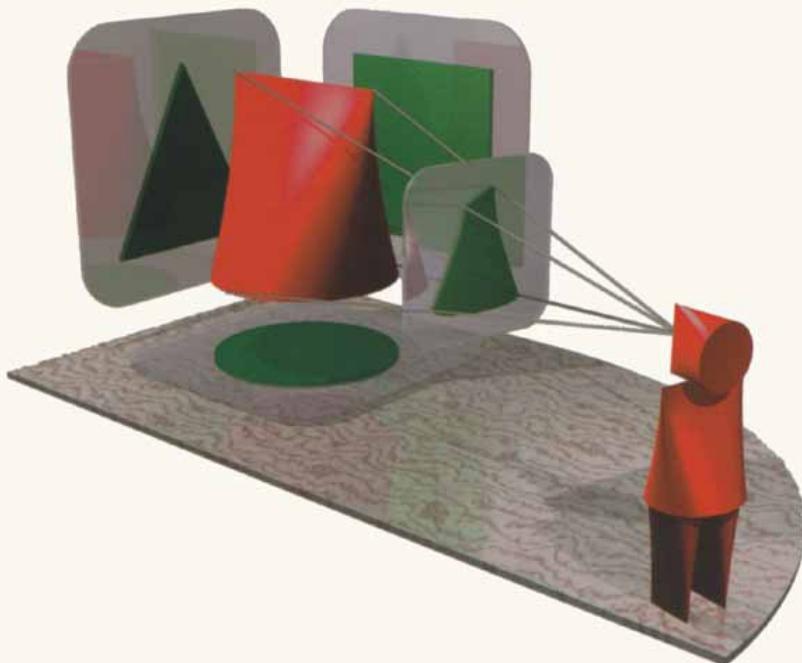


MONOGRAPHS IN COMPUTER SCIENCE

A THEORY OF OBJECTS



MARTÍN ABADI

LUCA CARDELLI



Springer

Monographs in Computer Science

Editors

David Gries
Fred B. Schneider

Springer Science+Business Media, LLC

Monographs in Computer Science

Abadi and Cardelli, **A Theory of Objects**

Brzozowski and Seger, **Asynchronous Circuits**

Selig, **Geometrical Methods in Robotics**

Nielson [editor], **ML with Concurrency**

Castillo, Gutiérrez, and Hadi, **Expert Systems and Probabilistic Network Models**

Martín Abadi Luca Cardelli

A Theory of Objects



Martin Abadi
Luca Cardelli
Systems Research Center
Digital Equipment Corporation
Palo Alto, CA 94301
USA

Luca Cardelli
(current address):
Microsoft Research
Cambridge, CB2 3NH
UK

Series Editors:

David Gries
Department of Computer Science
Cornell University
Upson Hall
Ithaca, NY 14853-7501
USA

Fred B. Schneider
Department of Computer Science
Cornell University
Upson Hall
Ithaca, NY 14853-7501
USA

With nine illustrations.

Library of Congress Cataloging-in-Publication Data
Abadi, Martin.

A theory of objects / Martin Abadi, Luca Cardelli.
p. cm. — (Monographs in computer science)
Includes bibliographical references and index.
ISBN 978-1-4612-6445-3 ISBN 978-1-4419-8598-9 (eBook)
DOI 10.1007/978-1-4419-8598-9
1. Object-oriented programming (Computer science) I. Cardelli,
Luca. II. Title. III. Series.
QA76.64.A22 1996
005.13'1—dc20

96-17038

Printed on acid-free paper.

© 1996 Springer Science+Business Media New York
Originally published by Springer-Verlag New York, Inc. in 1996
Softcover reprint of the hardcover 1st edition 1996

All rights reserved. This work may not be translated or copied in whole or in part without the written permission of the publisher (Springer Science+Business Media, LLC), except for brief excerpts in connection with reviews or scholarly analysis. Use in connection with any form of information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed is forbidden.

The use of general descriptive names, trade names, trademarks, etc., in this publication, even if the former are not especially identified, is not to be taken as a sign that such names, as understood by the Trade Marks and Merchandise Marks Act, may accordingly be used freely by anyone.

Production managed by Robert Wexler; manufacturing supervised by Jeffrey Taub.
Camera-ready copy prepared from the authors' files.
Cover illustration, "Object Subject," by Luca Cardelli.

9 8 7 6 5 4 3 2 (Corrected second printing, 1998)

ISBN 978-1-4612-6445-3

SPIN 10669864

Preface

This book develops a theory of objects as a foundation for object-oriented languages and programming. Our theory provides explanations for object-oriented notions in terms of a few basic primitives, and can be useful for the design and understanding of programming languages.

There is a well-established theory of functions, the λ -calculus, that serves as a foundation for procedural languages. Our treatment of object-oriented languages largely parallels existing literature on procedural languages. However, our theory of objects is self-contained; it is the first that does not require explicit reference to functions or procedures.

The material in this book is based on research that we carried out during the past three years; some of this material has appeared in conference and journal articles [2, 4, 5, 6, 7, 9]. In the development of our ideas, we built on previous work on the foundations of object-oriented programming as represented, for example, by the collection of papers edited by Gunter and Mitchell [68]. That body of work produced an understanding of subtyping and several explanations of objects in terms of functions and records. We discuss some of those previous results in the course of the book.

The chapters of this book are grouped in four main parts, bracketed between a Prologue and an Epilogue. The initial part, the Review, contains a general non-technical explanation of object-oriented concepts. Part I describes untyped and simply-typed formalisms for objects and classes. Part II discusses polymorphism, data abstraction, and the Self type, along with their connections to subtyping and subclassing. Part III investigates subtyping between type operators, which is more powerful than simple subtyping and supports more flexible inheritance (for example, for binary methods). The final sections of the book contain an Appendix summarizing our type rules, a list of figures, a list of tables, a list of notations, and a list of languages.

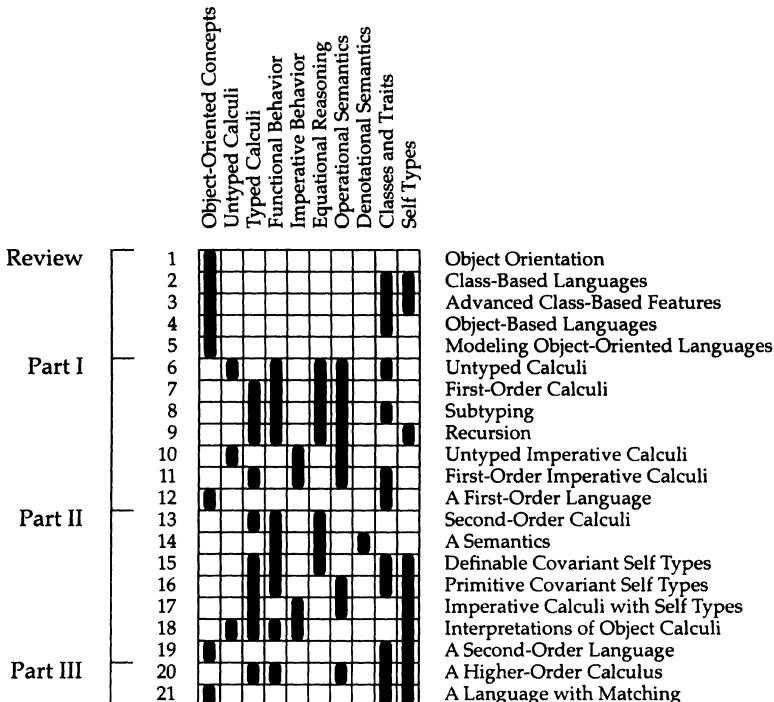
The intended audience of the book consists of computer scientists, professionals, and senior and graduate students with an interest in modern language issues. The Review is accessible to programmers familiar with at least one object-oriented language. It provides a general, concise, and language-independent perspective on object-oriented concepts, and should make up for lack of broad knowledge about objects. An acquaintance with the λ -calculus and with type systems is helpful background for the rest of the book. Part I is technical but relatively easy; it does not assume much expertise or skill in programming-language theory. Parts II and III are more challenging.

Subsets of this book are suitable for advanced courses on the theory, design, and structure of programming languages, even if not entirely devoted to object-oriented languages. The book can be used to complement standard object-oriented analysis,

design, and programming texts. The diagrams given next should help in selecting the relevant subsets.

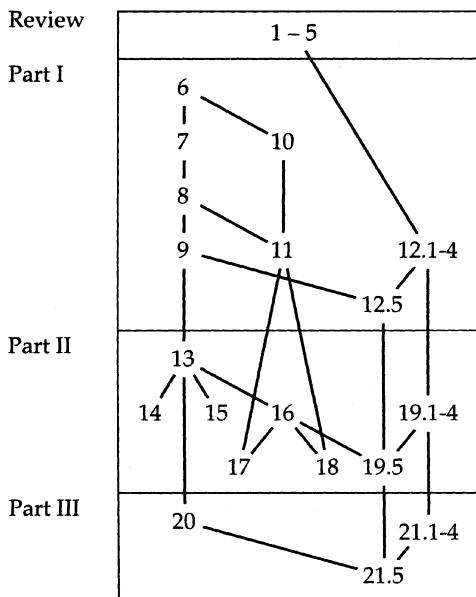
Concept Map

The first diagram shows the general organization of the book. Each row represents a chapter, and each column represents a theme. The black stripes indicate the occurrence of themes in chapters. A scan of a row indicates the major themes developed in a chapter. A scan of a column indicates the flow of a theme through the chapters.



Chapter Dependencies

The next diagram shows the dependencies between chapters: each chapter should be read after reading the chapters above it in the graph. Some sections of Chapters 12, 19, and 21 are singled out as depending only on the Review and on each other; those sections assume some familiarity with the notation used elsewhere in the book.



Acknowledgments

Many people have helped us in the preparation of this book.

We are grateful to Ramesh Viswanathan, with whom we wrote a paper that is the origin of Chapter 18. We are also grateful to Kim Bruce, Adriana Compagnoni, Bill Kal-sow, John Lamping, Paul-André Melliès, John Mitchell, Gordon Plotkin, and Scott Smith for fruitful discussions.

In addition, we received many valuable comments on drafts of this book from Felice Cardone, Antony Courtney, Dave Detlefs, Luis Dominguez, Kathleen Fisher, Andy Gordon, Jason Hickey, Brian Howard, Michael Isard, Ole Madsen, Jens Palsberg, Gareth Rees, Scott Smith, Valery Trifonov, David Ungar, and Peter Wegner.

Finally, we thank our colleagues at Digital's Systems Research Center and our manager, Bob Taylor, for their support.

Pointers

We expect to maintain a Web page for general information, sample sections, and auxiliary material about this book. At the time of writing, this page is located at:

```
<http://www.research.digital.com/SRC/personal/  
Luca_Cardelli/TheoryOfObjects.html>
```

Springer servers are located at:

<http://www.springer-ny.com>	(North America)
<http://www.springer.de>	(Europe)

Palo Alto, February 1996

Contents

PREFACE	v
PROLOGUE	1

REVIEW: Object-Oriented Features

1 Object Orientation	7
1.1 Objects	7
1.2 Reuse	8
1.3 Classifying Features	9
2 Class-Based Languages	11
2.1 Classes and Objects	11
2.2 Method Lookup	13
2.3 Subclasses and Inheritance	15
2.4 Subsumption and Dynamic Dispatch	17
2.5 Type Information, Lost and Found	19
2.6 Covariance, Contravariance, and Invariance	20
2.7 Method Specialization	22
2.8 Self Type Specialization	23
3 Advanced Class-Based Features	25
3.1 Object Types	25
3.2 Distinguishing Subclassing from Subtyping	27
3.3 Type Parameters	28
3.4 Subclassing without Subtyping	30
3.5 Object Protocols	32
4 Object-Based Languages	35
4.1 Objects without Classes	35
4.2 Prototypes and Clones	36
4.3 Inheritance by Embedding and by Delegation	38
4.4 Embedding	39
4.5 Delegation	42
4.6 Embedding versus Delegation	45
4.7 Dynamic Inheritance and Mode-Switching	46
4.8 Traits: From Prototypes back to Classes?	47
4.9 Types for Object-Based Languages	49

5 Modeling Object-Oriented Languages.....	51
5.1 Reduction to Basic Mechanisms	51
5.2 The Role of Method Update	52
5.3 The Scope of this Book	53
PART I: Untyped and First-Order Calculi	
6 Untyped Calculi.....	57
6.1 Object Primitives.....	57
6.2 The ζ -Calculus.....	60
6.3 Functions as Objects.....	66
6.4 Fixpoints.....	68
6.5 Examples	69
6.6 Traits, Classes, and Inheritance	73
6.7 Interpretations of Objects	76
7 First-Order Calculi	79
7.1 Formal Systems.....	79
7.2 The Object Fragment	80
7.3 Standard First-Order Fragments	82
7.4 Examples	84
7.5 Some Properties of \mathbf{Ob}_1	85
7.6 First-Order Equational Theories.....	89
7.7 Functions and Fixpoints.....	91
8 Subtyping.....	93
8.1 Subtyping.....	93
8.2 Examples	95
8.3 Some Properties of $\mathbf{Ob}_{1<\mu}$	95
8.4 First-Order Equational Theories with Subtyping	98
8.5 Classes and Inheritance	100
8.6 Objects versus Records.....	106
8.7 Variance Annotations.....	109
9 Recursion	113
9.1 Recursion	113
9.2 Recursion and Subsumption	115
9.3 Some Properties of $\mathbf{Ob}_{1<\mu}$	118
9.4 Examples	121
9.5 The Shortcomings of First-Order Typing	123
9.6 Towards the Type Self	125
9.7 Dynamic Typing	126

10 Untyped Imperative Calculi	129
10.1 Syntax	129
10.2 Fields	130
10.3 Procedures	131
10.4 Examples	133
10.5 Operational Semantics	135
11 First-Order Imperative Calculi	141
11.1 Typing	141
11.2 Examples of typings	142
11.3 Classes and Global Behavior Change	144
11.4 Subject Reduction	146
12 A First-Order Language	153
12.1 Features	153
12.2 Syntax	154
12.3 Examples	156
12.4 Typing	159
12.5 Translation	162

PART II: Second-Order Calculi

13 Second-Order Calculi	169
13.1 The Universal Quantifier	169
13.2 The Existential Quantifier	173
13.3 Variance Properties	177
13.4 Variant Product and Function Types, Encoded in $\mathbf{Ob}_<$	178
13.5 The Self Quantifier	179
14 A Semantics	185
14.1 The Untyped Universe	185
14.2 Types in the Untyped Universe	187
14.3 The Interpretation of Types and Typed Terms	196
15 Definable Covariant Self Types	201
15.1 ζ -Objects	201
15.2 Examples, with Typing	205
15.3 Binary Methods and the Covariance Requirement	208
15.4 Classes and Inheritance, with Self	209
15.5 Updating from the Outside	210
15.6 Recoup	212
15.7 Objects with Structural Invariants	215

16 Primitive Covariant Self Types	221
16.1 Primitive Self Types and Structural Rules	221
16.2 Objects with Structural Rules	222
16.3 Quantifiers	227
16.4 Subject Reduction	229
16.5 Examples	233
16.6 Classes and Inheritance, with Primitive Self	237
17 Imperative Calculi with Self Types	241
17.1 Syntax of Terms and Operational Semantics	241
17.2 Typing with Self	242
17.3 Quantifiers	243
17.4 Examples	245
17.5 Subject Reduction	247
18 Interpretations of Object Calculi	257
18.1 The Interpretation Problem	257
18.2 Untyped Interpretations	259
18.3 Typed Interpretations	264
19 A Second-Order Language	273
19.1 Features	273
19.2 Syntax	274
19.3 Examples	276
19.4 Typing	279
19.5 Translation	282
PART III: Higher-Order Calculi	
20 A Higher-Order Calculus	287
20.1 Syntax of $Ob_{\omega<\mu}$	287
20.2 Operational Semantics	289
20.3 Typing	290
20.4 Binary Methods	294
20.5 Basic Properties of $Ob_{\omega<\mu}$	298
20.6 Subject Reduction	301
21 A Language with Matching	305
21.1 Features	305
21.2 Syntax	306
21.3 Examples	307
21.4 Typing	311
21.5 Translation	315
EPILOGUE	325

APPENDIX: Rules and Proofs

A Fragments	329
A.1 Simple-Objects Fragments.....	329
A.2 Other Typing Fragments.....	330
A.3 Other Equational Fragments.....	333
B Systems.....	337
B.1 The $\text{Ob}_{1\leq}$ Calculus	337
B.2 The $F_{<\mu}$ Calculus	339
B.3 The ζOb Calculus	342
C Proofs	347
C.1 Proof of the Variance Lemma from Section 13.3	347
C.2 Proof of the Variance Lemma from Section 16.4	351
C.3 Deriving the Rules for ζ -Objects from Section 15.1.2	352
C.4 Denotational Soundness of Equational Rules.....	354
LIST OF FIGURES	363
LIST OF TABLES	365
LIST OF NOTATIONS.....	371
LIST OF LANGUAGES.....	381
BIBLIOGRAPHY.....	383
INDEX	391

Prologue

Procedural languages are generally well understood; their constructs are by now standard, and their formal underpinnings are solid. The fundamental features of these languages have been distilled into formalisms that prove useful in identifying and explaining issues of implementation, static analysis, semantics, and verification.

An analogous understanding has not yet emerged for object-oriented languages. There is no widespread agreement on a collection of basic constructs and on their properties. Consequently, practical object-oriented languages often support disparate features and programming techniques with little concern for orthogonality. This situation might improve if we had a better understanding of the foundations of object-oriented languages.

Various calculi of functions (λ -calculi) have been used as foundations for procedural languages. It is natural to attempt to use those same calculi for object-oriented languages. Unfortunately, there always seems to be a mismatch when one tries to model objects with functions. This mismatch is manageable for untyped languages, but becomes a serious obstacle when one attempts to explain typed object-oriented languages in terms of typed function calculi.

In this book, we take a different approach. Instead of taking functions as primitive and struggling with complex encodings of objects as functions, we take objects as primitive and concentrate on the intrinsic rules that objects should obey. We introduce *object calculi* and develop a theory of objects around them. These object calculi are as simple as function calculi, but represent objects directly.

Our theory of objects clarifies the general principles of object-oriented languages. Understanding those general principles can be helpful to those studying or designing programming languages. The theory suggests how to interpret existing constructions, and how to create and assess new ones. It also provides tools for reasoning about languages (existing or new), in particular for soundness proofs.

Using object calculi, we explain both the semantics of objects and their typing rules. We account for a range of object-oriented concepts, such as self, dynamic dispatch, classes, inheritance, prototyping, subtyping, covariance and contravariance, and method specialization. Because of the compactness of object calculi, we hope that they will serve as a convenient basis for further research on specification, verification, and analysis of object-oriented programs.

The book begins with an informal review of object-oriented concepts, where we aim to describe and categorize the fundamental features of object-oriented languages. The main distinction we make is between class-based and object-based languages. Object-based languages simplify and generalize class-based languages by reducing

classes to more primitive notions. Our object calculi are loosely modeled after object-based languages, but are even more primitive.

After the review, the main body of the book is divided into three parts where we discuss semantic, typing, and programming structures of increasing sophistication. At the end of each of the three parts, we present an object-oriented programming language that embodies the main features considered in that part. Given some familiarity with our notation, these languages should be understandable independently of the more formal material. For each language, we give an interpretation into one of our object calculi, establishing the correctness of the language's type system.

In Part I of the book we begin the development of our theory by introducing a tiny untyped object calculus. This is a calculus where the only entities are objects and object operations, with a built-in notion of self. In a sense, this calculus embodies the Smalltalk credo that anything can be modeled as an object. We go even further than Smalltalk by showing that objects can represent functions and numbers, and that classes can be programmed from objects. Several important variants of classes arise naturally.

After studying the untyped object calculus, we associate various type structures with it. We consider, in particular, object types, recursive types, subtyping, and dynamic types. These basic type constructions give us a type system rich enough to interpret a first object-oriented language. This language is, in rough terms, similar to languages such as Simula and Modula-3. Although lacking much of the convenience of common languages, it is remarkably expressive in that it integrates class-based and object-based notions.

In Parts II and III we move beyond basic type constructions and we investigate the Self type (the type of self). This topic brings us into the realm of polymorphism, data abstraction, and type operators. We review standard constructions that type theory provides to model these concepts, and incorporate them into our object calculi. We also give typing rules for the Self type and prove their correctness, in the context of both calculi and languages.

The Self type often occurs only as the type of the results of methods. We show that, in this special case, it can be understood through a combination of data abstraction and type recursion, and that method inheritance requires polymorphism. The general case where the Self type may occur in argument positions is also important; it arises in the presence of binary methods, which take arguments of the type of self. The programming and typing constructions related to binary methods are still in flux. Nonetheless, we show how to analyze them using fairly standard constructions from type theory. We conclude Parts II and III with two object-oriented languages that incorporate the Self type, and we discuss their subtle rules.

At the beginning of this Prologue we remarked on a mismatch between objects and functions. In the course of the book we develop a number of ideas and techniques that enable us to assess that mismatch more precisely. Specifically, we consider the problem of translating typed object calculi into typed function calculi. This is, in a sense, the problem of programming in object-oriented style within a typed procedural language. We show that this is possible in principle, given a sufficiently expressive procedural

language, but we argue that this would be too inconvenient in practice. Therefore we find that the expressiveness of object-oriented languages cannot be emulated easily by procedural languages.

Preview of Calculi and their Features

The following table summarizes the object calculi that we discuss in this book, and the main features that they contain explicitly. Although the names of the object calculi are not meaningful at this point, the table should give an idea of the general coverage of the various features, and will be helpful later to recall the relationships between the various calculi. Some calculi have no name (*nn*). The table does not have rows for the many features that can be expressed indirectly. A hollow bullet indicates that a feature can be encoded in a given calculus.

Object calculi and their features

We do not provide a systematic treatment of function calculi, because our main focus is on object calculi. However, in the course of the book we define in detail many of the common function calculi. The following table summarizes these calculi and their features.

Function calculi and their features

Calculus:	λ	F_1	$F_{1<}$	$F_{1\mu}$	$F_{1<\mu}$	$\text{imp}\lambda$	F	$F_{<}$	F_μ	$F_{<\mu}$
Defined in:	6.3	7.3	8.1	9.1	9.2	10.3	13.1– 13.2	13.1– 13.2	13.1– 13.2	13.1– 13.2
functions	•	•	•	•	•	•	•	•	•	•
function types		•	•	•	•		•	•	•	•
subtyping			•		•			•		•
recursive types				•	•				•	•
side-effects						•				
quantified types							•	•	•	•

Several combinations of function and object calculi are defined in the book but are not represented in these tables.

REVIEW

Object-Oriented Features

1 OBJECT ORIENTATION

In this Review we describe many of the standard concepts found in object-oriented programming, as a preparation for the development of our theory of objects. This first chapter introduces general notions, including the notion of object. In later chapters we discuss more concretely how these notions are represented in programming languages.

1.1 Objects

The *object-oriented* approach to programming is based on an intuitive correspondence between a software simulation of a physical system and the physical system itself. An analogy is drawn between building an algorithmic model of a physical system from software components and building a mechanical model of a physical system from concrete objects. By analogy, the software components are themselves called *objects*. In its purest form, the object-oriented approach recommends that every system be developed according to this analogy.

The development of a mechanical model includes analysis, design, and implementation aspects. Consider, for example, the task of building a mechanical model of the solar system. The analysis aspect consists in realizing that planets move along precise orbits. The design aspect consists in inventing a mechanism for moving spheres along such orbits. The implementation aspect consists in preparing and assembling spheres, gears, and springs.

All three aspects of constructing a model involve entities we may call objects, but with different connotations. In analysis, planets are seen as objects, but orbits clearly are not. In design, the intangible orbits are modeled by concrete objects such as orbital tracks and gears. Entirely new objects, such as power sources, may appear. In implementation, a spring may be an object that realizes the power source abstraction.

Object-oriented programming similarly involves analysis, design, and implementation aspects, with concrete objects replaced by software objects. The term *object* in programming refers to a component of a software model, not to a component of the system being modeled. The proper analogy is between software objects and objects in a “real model” that one may imagine building, rather than objects in the “real world”. For example, orbital tracks may be represented as software objects.

This approach to programming originated with Simula, which was initially dedicated to solving simulation (model building) problems [57]. Its main concepts were further refined in Smalltalk, where the concreteness of objects was originally intended as a pedagogical tool. Since then, the object-oriented methodology has been exploited in

a wide range of applications, including the construction of user interfaces, operating systems, and databases. Although these applications do not concern physical systems, the object-oriented methodology has carried over.

The properties that confer this broad range of applicability to the object-oriented approach can be summarized as follows [84, 86].

- The analogy between software models and physical models.
- The resilience of the software models.
- The reusability of the components of the software models.

The first point pertains to analysis. In the object-oriented analysis process, the analogy with a physical model can be useful in the development of a software model. Much effort has been invested in systematizing this analysis process; see for example [30].

The second point pertains to design. Resilience of design in the face of changes is a consequence of building abstractions. Objects form natural data abstraction boundaries and help focus a design on system structure instead of algorithms. Algorithms are factored into *methods* that are attached to objects, making the objects behaviorally autonomous. Because of object abstractions, a design can evolve with fewer pervasive reorganizations.

The third point pertains to implementation. Objects are naturally organized into taxonomies during analysis, design, and implementation. This hierarchical organization encourages the reuse of methods and data that are located higher in the hierarchy.

Object-oriented programming does not have an exclusive claim to all these good properties. Systems may be modeled by other paradigms, including ones based on traditional notions of algorithms and data structures, which were not well developed when Simula was invented [57]. Resilience can be achieved just as well by organizing programs around abstract data types [81], independently of taxonomies; in fact, data abstraction alone is sometimes taken as the essence of object orientation [19, 29]. Reusability can be achieved by modularization [101, 127] and parameterization [90, 91]. Hence it is possible that, as the availability and awareness of other techniques grow, the appeal of objects will fade away.

Still, the object-oriented approach has proven uniquely successful. It manages to integrate good analysis, design, and implementation techniques into a relatively intuitive and uniform framework. Moreover, some of its fundamental features are not easily explained as the union of other well-understood notions; witness the relative scarcity of formal techniques for analyzing object-oriented programs.

1.2 Reuse

It is often claimed that object-oriented languages allow reuse of software components better than traditional procedural languages. In support of this claim, we survey abstractly the reuse mechanisms that are provided exclusively by object-oriented languages, and the main implications of these mechanisms. A more detailed discussion is carried out in the following chapters.

A software component is reusable when it can be easily used in more than one context. For example, a module can be reused by importing it in several other modules, and a generic module can be reused by instantiating it with different parameters. In all cases, reuse entails the replacement of a component (or a placeholder) with another component in a context. In traditional procedural languages, such a replacement requires an exact agreement in type or interface.

In object-oriented languages there are two distinctive kinds of replacements. Objects can be replaced by other objects, and methods can be replaced by other methods. These forms of replacement do not require exact agreement of type or interface, only approximate agreement.

Various mechanisms allow replacing objects. In general terms, one may replace an object with a new one that has at least the same set of attributes. Any additional attributes of the new object remain as *invisible attributes*; they are preserved but are not directly accessible.

The replacement of methods is called *overriding*, while the reuse of existing methods is called *inheritance*. New objects, or new generators of objects, are *derived* from old ones by a mixture of reuse (inheritance) and variation (overriding). When a method is overridden by a new one, the new method must conform to the interface of the old one, but with some flexibility. In particular, the new method may be more specific than the old one, in interface or in behavior, and may use attributes that are available only in the derived object.

An almost inevitable consequence of these replacement mechanisms is the notion of *self*. By the special name *self*, a method can refer to its host object, and hence to its sibling methods (the other methods of the same object). Through inheritance and overriding, the siblings of a method may change. Therefore *self* has a dynamic meaning: it always gives access to the current siblings of a method. This dynamic notion of *self* is crucial, because it allows a method to exhibit a new behavior when inherited into a derived object, depending on the siblings it finds there. Without a notion of *self*, the behavior of methods is more rigid and method reuse is limited.

A further consequence of these replacement mechanisms is that methods are closely bound to objects. Because of the flexibility in object and method replacement, one cannot know statically the precise kind of all the objects that may be dynamically bound to a variable, and whether those objects have invisible attributes. Moreover, one cannot know statically what methods an object possesses, and whether those methods rely upon invisible attributes of their host object. Extracting a method from an object, and reusing it in a context that does not provide the appropriate invisible attributes, would be unsound. Thus, unlike procedures operating on data structures, methods are integral and inseparable parts of objects.

1.3 Classifying Features

In the remaining chapters of this Review we discuss the specific incarnations of object replacement and method replacement that are found in various kinds of programming

languages. The purpose of the Review is to establish basic terminology, and to describe, informally, the wide and somewhat bewildering spectrum of object-oriented features that one may want to formalize.

The description and classification of object-oriented notions is not an easy task. There is a great variety of object-oriented languages. Most languages have overlapping features, and each feature is present in varying degree in several languages. Therefore any attempt at categorization has many potential pitfalls. Describing individual languages leads to discussions about boring, irrelevant, and historically obsolete details. Comparing languages by their features leads to oversimplification, and is only practical for a small number of languages. Describing features in the abstract leads to concepts that, in their pure forms, may not exist in any language.

We choose the last option: describing disembodied features. In this way we are free to focus on the most interesting topics, and we can sidestep complex accidents of actual languages. To keep the discussion flowing, we relegate to footnotes specific comparisons with actual languages.

Throughout the Review we use a consistent pseudo-notation for object-oriented notions. Our pseudo-notation is intended to be suggestive rather than formal. It is not our goal to detail the semantics and typing rules of either the pseudo-notation or actual programming languages. We describe the meaning of the pseudo-notation informally, and we discuss, in general terms, how it relates to features of existing languages. Formal counterparts to the pseudo-notation appear throughout the rest of the book.

2 CLASS-BASED LANGUAGES

Class-based languages form the mainstream of object-oriented programming. Among the best-known class-based languages are Simula, the first object-oriented language, Smalltalk, the first dynamically typed object-oriented language, and C++, whose main class-based features are modeled after those of Simula. (Bibliographic references to these and other languages can be found in the List of Languages at the back of the book.)

In this chapter we discuss a kernel of properties that, with minimal controversy, characterize classical class-based languages above and beyond their countless peculiarities. More advanced class-based features are treated in Chapter 3.

2.1 Classes and Objects

Class-based languages are centered around the notion of *classes* as descriptions of objects. We begin with an example of a class declaration.

```
class cell is
    var contents: Integer := 0;
    method get(): Integer is
        return self.contents;
    end;
    method set(n: Integer) is
        self.contents := n;
    end;
end;
```

A class is intended to describe the structure of all the objects generated from the class. The class *cell* describes storage-cell objects having an integer field named *contents*, which is initialized to zero, and two methods named *get* and *set*, whose code operates on the *contents* field.¹ The *get* method has no parameters; when invoked, it fetches the *contents* field and returns it. The *set* method has an integer parameter; it stores the

¹ There is a wide spectrum of class notions in programming languages, from classes detailing values for all fields and code for all methods, to “abstract classes” with as yet uncoded methods, to classes with no specified values or code at all. We take classes to have fully specified values and code. The case where there are no values or code is better covered by object types, described in a later section.

parameter in the *contents* field and returns nothing. Within these methods, the special identifier **self** refers to the host object. The fields and methods of an object are collectively called its *attributes*.

The intended behavior of objects can be understood in terms of the naive storage model of Figure 2-1. In that model, an object is internally represented as a reference to a record of attributes. An operation on an object attribute implicitly bypasses the reference in order to access the attribute. Program variables hold references, not attribute records. Parameter passing and assignment copy references, not the associated attribute records. Therefore parameter passing and assignment produce sharing of attribute records.²

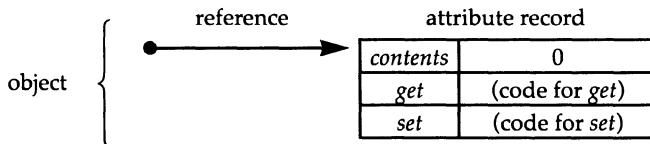


Figure 2-1. Naive storage model.

An object can be created from a class *c* by the construction **new c**. More precisely, **new c** allocates an attribute record and returns a reference to it. The attribute record contains the initial values and the method code specified by *c*. Distinct executions of **new c** produce distinct objects; that is, they produce references to distinct attribute records. An object generated from class *c* via **new** is colloquially called “an object of class *c*” or “an instance of class *c*”.

We indicate by *InstanceTypeOf(c)* the type of objects of class *c*. In the following example, a new cell is bound to the variable *myCell* with type *InstanceTypeOf(cell)*:

```
var myCell: InstanceTypeOf(cell) := new cell;
```

By introducing the type *InstanceTypeOf(cell)*, we have made an important distinction between classes and types. We could have chosen to consider *cell*, instead of *InstanceTypeOf(cell)*, as the type of the objects generated from *cell*. However, such an identification of classes with types would cause confusion later.

Given a cell named *myCell*, we can extract the value of the *contents* field by *myCell.contents*, and we can update it by *myCell.contents:=n*.³ The *contents* fields of distinct objects of class *cell* change independently upon updates. That is, each object of class *cell* has a separate *contents* field.

² Some languages, such as C++ and Oberon, expose the distinction between attribute records and references to attribute records in order to distinguish between stack-allocated and heap-allocated objects. Other languages, such as Simula, Smalltalk, and Modula-3, hide this distinction and deal exclusively with heap-allocated objects.

Given a cell, we can also invoke its methods. For example, `myCell.set(3)` is a method invocation that runs the code for `set` with the formal parameter `n` bound to 3 and with `self` bound to `myCell`. As a further example of method invocation, we define a procedure named `double` whose body doubles the contents of a cell:

```
procedure double(aCell: InstanceTypeOf(cell)) is
    aCell.set(2 * aCell.get());
end;
```

2.2 Method Lookup

Given a method invocation of the form `o.m(...)`, a language-dependent process called *method lookup* is responsible for identifying the appropriate method `m` of the object `o` that has to be executed. Method lookup is not completely trivial because of the standard storage model used for class-based languages, which is more complex than the naive model of Figure 2-1. We briefly discuss this storage model, since it is often presented as the semantics of class-based languages.

Although a class is meant to describe the structure of the objects constructed from it, methods are not directly *embedded* into objects as it might appear from the syntax of classes and from Figure 2-1. Instead, for space efficiency, they are factored into *method suites* that are shared by objects of the same class (Figure 2-2). Method lookup must access these method suites; we say that objects *delegate* method invocations to the method suites associated with their classes.

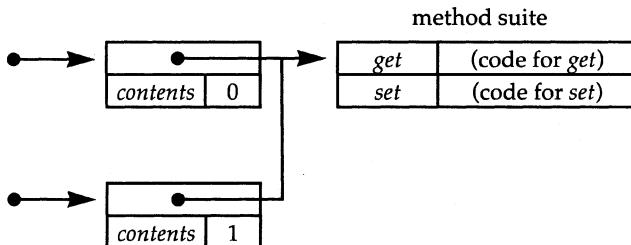


Figure 2-2. Method suites.

³ In some languages a field can be declared private (accessible only to the methods of its class) or protected (accessible only to the methods of its class and subclasses). Although representation hiding is important, it is not unique to object-oriented programming. Even within object-oriented programming, protection of fields and methods can be obtained by other mechanisms such as lexical hiding or subtyping (see Section 8.5.2). We do not enforce representation hiding for fields.

In the presence of inheritance, discussed shortly, method suites may be organized as a tree, and method lookup may require examining a chain of method suites. With multiple inheritance, the method suites may form a directed graph (sometimes including cycles); method lookup must include a strategy for examining such graphs. Depending on the language, some method lookups may be performed at compile-time, based on type information, and some at run-time, based on references to method suites contained in each object.

Although the description and implementation of method lookup can be quite complex, it is usually designed to produce the illusion that methods are, after all, embedded directly into objects, as in the naive storage model illustrated in Figure 2-1.⁴ When this illusion fails, confusion may result in both language semantics and programming.⁵

The illusion of embedding for both fields and methods is strengthened by the similarity of the standard notations $o.x$ for field extraction and $o.m(\dots)$ for method invocation. Moreover, in all implementations of method lookup, an occurrence of `self` within a method refers to the object that originally received the invocation of that method. Thus `self` invariably refers to the object that, according to the embedding interpretation, appears to contain the method.

It is interesting to notice that some features that would distinguish between embedding and delegation do not appear in class-based languages. Typical of class-based languages is the fact that methods, unlike fields, cannot be extracted from objects as functions, and cannot be updated within objects (by replacing them with other methods). Method extraction must be prohibited to preserve typing soundness, as we shall see. Method update is not necessarily unsound. With the embedding implementation, a method update would affect a single object. With the delegation implementation, however, it would affect all the objects of a class.⁶

Many class-based languages could be implemented equivalently by embedding or by delegation techniques. For simplicity, we assume the embedding model in the upcoming explanation of inheritance, and we only occasionally remark on delegation and method suites. The contrast between embedding and delegation will resurface in our discussion of object-based languages in Chapter 4.

⁴ The main class-based feature whose meaning depends on delegation of attributes is the notion of class variables (fields shared by all objects of a class), as it appears in Smalltalk and in Java; similar effects can be achieved with other mechanisms, such as global variables.

⁵ CLOS is an example of a language where method lookup is based on a complex delegation algorithm over a multiple-inheritance class hierarchy [113]. A detailed understanding of the automatic placement of methods in the hierarchy is necessary for predicting program behavior.

⁶ Beta is a class-based language that allows method update in individual objects. It syntactically distinguishes between methods that are embedded in objects (via “pattern variables”) and methods that are delegated to method suites. In fact, Beta has several features typical of object-based languages (see Chapter 4); method update is one of them.

2.3 Subclasses and Inheritance

Although the notion of class is rather interesting in its own right, it is a stepping stone for the notions of subclasses and inheritance. Like any class, a subclass describes the structure of a set of objects. However, it does so incrementally, by describing extensions and changes to its direct superclass. Fields from a superclass are implicitly replicated in a subclass, and new ones may be added. Methods from a superclass may be either replicated in a subclass, by default, or explicitly overridden by similarly named and typed methods.⁷ By the *subclass relation* we mean the partial order induced by the subclass declarations.

We extend our previous example with the declaration of a subclass *reCell* (restorable cell) of class *cell*. A restorable cell is a cell with an additional method that restores the contents to a previous state. The *set* method is overridden so that it makes a backup of the *contents* field before updating it; **super.set(n)** invokes the old version of *set* from the *cell* class.

```
subclass reCell of cell is
    var backup: Integer := 0;
    override set(n: Integer) is
        self.backup := self.contents;
        super.set(n);
    end;
    method restore() is
        self.contents := self.backup;
    end;
end;
```

Inheritance is the sharing of attributes between a class and its subclasses. This sharing includes the initial values of fields and the code of methods. So the initial value of the *contents* field and the code of the *get* method are inherited by *reCell* from *cell*. The *set* method could be inherited implicitly like the *get* method, but is instead overridden in *reCell*.

It is common to say, informally, that *c'* inherits from *c*, to mean that *c'* is a subclass of *c*. Note, though, that a subclass of a class with no fields may override all the methods of the superclass and therefore inherit nothing. Conversely, there are mechanisms for sharing method code that do not rely on the subclass relation (for example, straightforward procedural abstraction). Hence we are careful to distinguish inheritance from subclassing.

Without subclasses, an occurrence of **self** in a class declaration always refers to an object of that class. With subclasses, this is not the case. In a method that a subclass *c'*

⁷ Simula and C++ allow overriding only for those methods that have been declared overridable (virtual) in the superclass. We implicitly consider all methods to be virtual, as in Smalltalk and Modula-3.

inherits from a class c , `self` refers to an object of the subclass c' , not to an object of the original class c . In particular, through `self`, one can access methods redefined in c' , and cannot access the original methods from c .

The special identifier `super` in a new method can be used to invoke the old version of a method from a superclass. More precisely, `super.m(a)` stands for the invocation of the method *m* of the current direct superclass, with argument *a*, and with `self` still bound to the current `self`. The latter condition is critical; it ensures that any occurrence of `self` within the method *m* is interpreted with respect to the current subclass, and not with respect to any superclass.⁸

The implementation of method lookup must now be reconsidered in light of our discussion of subclasses. In the delegation-based implementation of inheritance there is a method suite for each (sub)class. In the case of dynamically typed languages, these suites are searched, from subclasses to superclasses, until an appropriate method is found (Figure 2-3). In the case of statically typed languages (where the hierarchy of

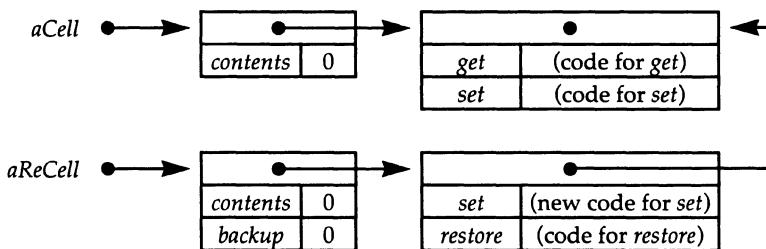


Figure 2-3. Hierarchical method suites.

method suites is known at compile-time) the suites are collapsed at each level so that iterative search is not necessary; in a sense, an embedding model is used after the first step of delegation (Figure 2-4).

Multiple inheritance is obtained when a class inherits attributes from more than one other class. (Otherwise one has *single inheritance*.) There would be nothing peculiar about multiple inheritance, except that it is usually specified by declaring a class as

⁸ Most object-oriented languages have some notion similar to **super** as described here, but details of its meaning vary greatly. We have described a particular version, closest to Smalltalk's **super** (whose meaning is described in terms of method lookup). See also **resend** in Self and **call-next-method** in CLOS. Many class-based languages do not provide a notion of **super** relative to the current class. Rather, they provide ways of extracting methods from a particular class; examples are selection from object types in Modula-3, qualified messages in C++, and "upward" **qua** in Simula. Beta uses a diametrically opposite mechanism, called **inner**, to allow virtual methods to invoke methods that override them. Bracha and Cook have proposed a general framework for method combination [33].

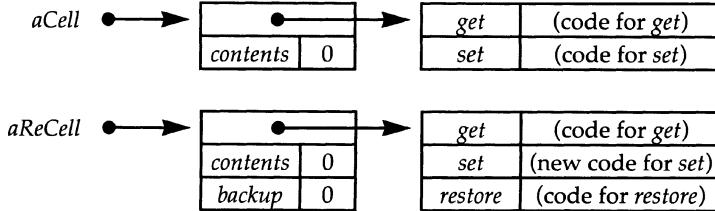


Figure 2-4. Collapsed method suites.

having multiple superclasses. Then one must decide what to do about conflicts and duplications among the attributes of the superclasses. The simplest solution is to identify all duplications of identical attributes, and to forbid any conflicts; many other solutions have been adopted. With multiple superclasses, there is no unique path upwards for method lookup to follow, and hence the meaning of **super** becomes rather delicate.⁹

2.4 Subsumption and Dynamic Dispatch

From what we have said so far, it may seem that subclasses are just a convenient mechanism to avoid rewriting definitions that already appear in superclasses. Much more comes into play with the associated notion of subsumption. Consider the definitions:

```
var myCell: InstanceTypeOf(cell) := new cell;
var myReCell: InstanceTypeOf(reCell) := new reCell;
procedure f(x: InstanceTypeOf(cell)) is ... end;
```

Consider also the following code fragments, in the scope of those definitions:

```
myCell := myReCell;
f(myReCell);
```

Here an instance of class *reCell* is assigned to a variable holding instances of class *cell*. Similarly, an instance of class *reCell* is passed to a procedure *f* that expects instances of class *cell*. Both code fragments would be illegal in a language like Pascal, because the types *InstanceTypeOf(cell)* and *InstanceTypeOf(reCell)* do not match. In object-oriented languages these code fragments are made legal by the following rule, which embodies what is often called (*subtype*) polymorphism:

If *c'* is a subclass of *c*, and *o'* is an instance of *c'*, then *o'* is an instance of *c*.

or, from the point of view of the typechecker:

⁹ See, for example, the mechanisms for multiple inheritance adopted in Eiffel, C++, and CLOS.

- (1) If c' is a subclass of c , and $o' : \text{InstanceTypeOf}(c')$, then $o' : \text{InstanceTypeOf}(c)$.

We analyze statement (1) further, by introducing a reflexive and transitive subtype relation ($<:$) between InstanceTypeOf types. This subtype relation is intended, intuitively, as set inclusion between sets of values. For now we do not define the subtype relation precisely, but we assume that it satisfies two properties:

- (2) If $a : A$ and $A <: B$, then $a : B$.
 (3) $\text{InstanceTypeOf}(c') <: \text{InstanceTypeOf}(c)$ if and only if c' is a subclass of c .

Together, (2) and (3) yield (1).

Property (2), called *subsumption*, is the characteristic property of subtype relations. By subsumption, a value of type A can be viewed as a value of a supertype B . We say that the value is subsumed from type A to type B .

Property (3), which we may call subclassing-is-subtyping, is the characteristic property of subclassing in classical class-based languages. Since inheritance is connected with subclassing, we may read (3) as an inheritance-is-subtyping property. More recent class-based languages adopt a different, inheritance-is-not-subtyping approach, as is shown later.

With the introduction of subsumption, we have to reexamine the meaning of method invocation. For example, given the code:

```
procedure g(x:  $\text{InstanceTypeOf}(\text{cell})$ ) is
  x.set(3);
end;
g(myReCell);
```

we should determine the meaning of *x.set(3)* during the invocation of *g*. The declared type of *x* is $\text{InstanceTypeOf}(\text{cell})$, while its value is *myReCell*, which is an instance of *reCell*. Because the method *set* is overridden in *reCell*, there are two possibilities:

<i>Static dispatch:</i>	<i>x.set(3)</i> executes the code of <i>set</i> from class <i>cell</i>
<i>Dynamic dispatch:</i>	<i>x.set(3)</i> executes the code of <i>set</i> from class <i>reCell</i>

Static dispatch is based on the compile-time type information available for *x*. Dynamic dispatch is based on the run-time value of *x*; we may say that $\text{InstanceTypeOf}(\text{reCell})$ is the *true type* of *x* during the execution of *g(myReCell)*, and that the true type determines the choice of method.

Dynamic dispatch is found in all object-oriented languages, to the point that it can be regarded as one of their defining properties. Dynamic dispatch is an important component of object abstraction: each object knows how to behave autonomously, so the context does not need to examine the object and decide which operation to apply.¹⁰

¹⁰ Some languages, including Simula, C++, and Modula-3 allow both kinds of dispatch. The static versions are based on distinct syntactic constructions.

An interesting consequence of dynamic dispatch is that subsumption should have no run-time effect on objects. For example, if an application of subsumption from *InstanceTypeOf(reCell)* to *InstanceTypeOf(cell)* were to “coerce” a *reCell* to a *cell* by cutting off its additional attributes (*backup* and *restore*), then a dynamically dispatched invocation of *set* would fail. The fact that subsumption has no run-time effect is both good for efficiency and semantically necessary.¹¹

2.5 Type Information, Lost and Found

Although subsumption has no run-time effect, it has the consequence of reducing static knowledge about the true type of an object. Imagine a root class with no attributes, such that all classes are subclasses of the root class. Then any object can be considered, by subsumption, as a member of the root class and can be regarded as a useless object with no attributes.

Less drastically, when subsuming an object from *InstanceTypeOf(reCell)* to *InstanceTypeOf(cell)*, the ability to access the field *backup* (as well as the method *restore*) is lost. This fact, however, does not make the field *backup* redundant because it is still used, through *self*, by the body of the overriding method *set*. So attributes forgotten by subsumption can still be used thanks to dynamic dispatch.

In a purist view of object-oriented methodology, dynamic dispatch is the only mechanism for taking advantage of attributes that have been forgotten by subsumption. This position is often taken on abstraction grounds: no knowledge should be obtainable about objects except by invoking their methods.¹² In the purist approach, subsumption provides a simple and effective mechanism for hiding private attributes. If we create a *reCell* and, by subsumption, give it to a client as a *cell*, we can be sure that the client cannot directly affect the *backup* field.

Most languages, however, provide some way of inspecting the type of an object and, consequently, of regaining access to its forgotten attributes [83, 95]. For example, a procedure with parameter *x* of type *InstanceTypeOf(cell)* could contain the following code. The **typecase** statement binds *x* to *c* or to *rc* depending on the true (run-time) type of *x*:

```
typecase x
  when rc: InstanceTypeOf(reCell) do ... rc.restore() ... ;
  when c: InstanceTypeOf(cell) do ... c.set(3) ... ;
end;
```

¹¹ Languages that support stack-based objects have to deal with such truncation effects on assignment and parameter passing. In these cases, either subsumption is forbidden (C++) or coercion is a series of field assignments (Oberon).

¹² C++ adopts this purist position, but uniquely on grounds of efficiency, and then resorts to unsafe casts for recovering type information [115]. A recent addition to C++ introduces a mechanism for examining the run-time type of objects.

Previously inaccessible attributes can now be used in the *rc* branch.¹³

The **typecase** mechanism is useful, but it is considered impure for several methodological reasons (and also for theoretical ones). First, it violates the object abstraction, revealing information that may be regarded as private. Second, it renders programs more fragile by introducing a form of dynamic failure when none of the branches apply. Third, and probably most important, it makes code less extensible: when adding another subclass of *cell* one may have to revisit and extend the **typecase** statements in existing code. In the purist framework, the addition of a new subclass does not require recoding of existing classes. This is a good property, in particular because the source code of commercial libraries may not be available.

Although **typecase** may be ultimately an unavoidable feature, its methodological drawbacks require that it be used prudently. The desire to reduce the uses of **typecase** has shaped much of the type structure of object-oriented languages. In particular, **typecase** on self is necessary for emulating objects in conventional languages by records of procedures; in contrast, the standard typing of methods in object-oriented languages avoids this need for **typecase**. More sophisticated typings of methods are aimed at avoiding **typecase** also on method results and on method arguments (using Self types, discussed later).

2.6 Covariance, Contravariance, and Invariance

In the rest of this chapter we study some flexible typing techniques for classes and methods that help in avoiding uses of **typecase**. To explain these techniques, we first review the basic subtyping properties of product and function types.

Let us consider product types. The type $A \times B$ is the type of pairs with left component of type A and right component of type B . The operations $fst(c)$ and $snd(c)$ extract the left and right components, respectively, of an element c of type $A \times B$.

We say that \times is a *covariant* operator (in both arguments), because $A \times B$ varies in the same sense as A or B :

$$A \times B <: A' \times B' \text{ provided that } A <: A' \text{ and } B <: B'$$

We can justify this property as follows.

Argument for the covariance of $A \times B$

A pair (a,b) with left component a of type A and right component b of type B , has type $A \times B$. If $A <: A'$ and $B <: B'$, then by subsumption we have $a : A'$ and $b : B'$, so that (a,b) has also type $A' \times B'$. Therefore any pair of type $A \times B$ has also type $A' \times B'$ whenever $A <: A'$ and $B <: B'$. In other words, the inclusion $A \times B <: A' \times B'$ between product types is valid whenever $A <: A'$ and $B <: B'$.

¹³ Instances of **typecase** appear in Modula-3 (as **narrow** and **typecase**), in Simula (as **qua** and **inspect**), and in Oberon (as type guards).

We examine function types next. The type $A \rightarrow B$ is the type of functions with argument type A and result type B .

We say that \rightarrow is a *contravariant* operator in its left argument because $A \rightarrow B$ varies in the opposite sense as A ; the right argument is instead covariant:

$$A \rightarrow B <: A' \rightarrow B' \text{ provided that } A' <: A \text{ and } B <: B'$$

Argument for the co/contravariance of $A \rightarrow B$

If $B <: B'$, then a function f of type $A \rightarrow B$ produces results of type B' by subsumption. If $A' <: A$, then f accepts also arguments of type A' , since these have type A by subsumption. Therefore every function of type $A \rightarrow B$ has also type $A' \rightarrow B'$ whenever $A' <: A$ and $B <: B'$. In other words, the inclusion $A \rightarrow B <: A' \rightarrow B'$ between function types is valid whenever $A' <: A$ and $B <: B'$.

In the case of functions of multiple arguments, for example of type $(A_1 \times A_2) \rightarrow B$, we have contravariance in both A_1 and A_2 . This is because product, which is covariant in both of its arguments, is found in a contravariant context.

Finally, consider pairs whose components can be updated; we temporarily indicate their type by $A * B$. Given $p : A * B$, $a : A$, and $b : B$, we have operations $getLft(p) : A$ and $getRht(p) : B$ that extract components, and operations $setLft(p, a)$ and $setRht(p, b)$ that destructively update components.

The operator $*$ does not enjoy any covariance or contravariance properties:

$$A * B <: A' * B' \text{ provided that } A = A' \text{ and } B = B'$$

We say that $*$ is an *invariant* operator (in both of its arguments).

Argument for the invariance of $A * B$

If $A <: A'$ and $B <: B'$, can we covariantly allow $A * B <: A' * B'$? If we adopt this inclusion, then from $p : A * B$ we obtain $p : A' * B'$ and we can perform $setLft(p, a')$ for any $a' : A'$. After that, $getLft(p)$ might return an element of type A' that is not an element of type A . Hence the inclusion $A * B <: A' * B'$ is not sound.

Conversely, if $A'' <: A$ and $B'' <: B$, can we contravariantly allow $A * B <: A'' * B''$? From $p : A * B$ we now obtain $p : A'' * B''$, and we can incorrectly deduce that $getLft(p) : A''$. Hence the inclusion $A * B <: A'' * B''$ is not sound either.

Much controversy in the object-oriented community revolves around the question of whether the types of arguments of methods should vary covariantly or contravariantly from classes to subclasses. We must stress that the properties of \times , \rightarrow , and $*$ follow inevitably from our assumptions; we cannot take \rightarrow to be covariant in its left argument any more than we can take π to be 3.0. In the following sections we examine how variance properties of method types follow inevitably by a similar analysis. We cannot take method argument types to vary covariantly, unless somehow we change

the meaning of covariance, subtyping, or subsumption (e.g., as in [48]). According to our definitions, covariance of method argument types is statically unsound: if left unchecked, it may result in unpredictable behavior.¹⁴

2.7 Method Specialization

In our discussion of subclasses we have taken the simplest approach to overriding, requiring that an overriding method has exactly the same type as the overridden method. This condition can be relaxed to allow *method specialization*, that is, to allow an overriding method to adopt different argument and result types, specialized for the subclass. We still do not allow overriding and specialization of field types: fields are updatable, like the components of the type $A \otimes B$, and therefore their types must be invariant.

Suppose we use different argument and result types, A' and B' , for an overriding method m :

```
class c is
    method m(x: A): B is ... end;
    method m1(x1: A1): B1 is ... end;
end;

subclass c' of c is
    override m(x: A'): B' is ... end;
end;
```

In determining the admissible A' and B' we are constrained by the possibility of subsumption between $InstanceTypeOf(c')$ and $InstanceTypeOf(c)$. When o' of type $InstanceTypeOf(c')$ is subsumed into $InstanceTypeOf(c)$, and $o'.m(a)$ is invoked, it is acceptable for the argument to have static type A and for the result to have static type B . Therefore it is sufficient to require that $B' <: B$ (covariantly) and that $A <: A'$ (contravariantly). This is called *method specialization on override*: the result type B is specialized to B' and the parameter type A is generalized to A' , with the net effect that $A \rightarrow B$ is specialized to $A' \rightarrow B'$.

There is another form of method specialization that happens implicitly by inheritance. The occurrences of **self** in the methods of c can be considered of type $InstanceTypeOf(c)$, in the sense that all objects bound to **self** have type $InstanceTypeOf(c)$ or a subtype of $InstanceTypeOf(c)$. When the methods of c are inherited by c' , the same occurrences of **self** can similarly be considered of type $InstanceTypeOf(c')$. Thus the type of **self** is silently specialized on inheritance (covariantly!).

¹⁴ Eiffel still favors covariance of method arguments; the issue is discussed in [55] and in [89], Chapter 22. In Beta, unsound behavior is caught by run-time checks. Covariance can be soundly adopted for multiple dispatch, but using a different set of type operators [48].

Another way to look at specialization is to consider methods as functions whose first parameter is *self*; we call such functions *pre-methods*. In the case of the method m_1 of c , this pre-method pm_1 would have type $(\text{InstanceTypeOf}(c) \times A_1) \rightarrow B_1$. This type is a subtype of $(\text{InstanceTypeOf}(c') \times A_1) \rightarrow B_1$, so pm_1 has this type too by subsumption. Then pm_1 has the type of a legal pre-method for c' , and it can be inherited by c' . More generally, an inheritable pre-method of type $(\text{InstanceTypeOf}(c) \times A_1) \rightarrow B_1$ has by subsumption the type $(\text{InstanceTypeOf}(c') \times A'_1) \rightarrow B_1$ for any $A'_1 <: A_1$ and $B_1 <: B'_1$. That is, the parameters, including *self*, may be specialized, and the results may be generalized.

Note that the argument and result inclusions we have derived for inheritance are opposite to the ones for overriding: for inheritance, parameter types may be specialized, and result types may be generalized; for overriding, parameter types may be generalized, and result types may be specialized. In any case, the sound rules for method specialization, both for inheritance and for overriding, are a direct consequence of the constraints on subtyping and subsumption.

2.8 Self Type Specialization

Method specialization adds flexibility to subclass definitions by allowing the result type of a method to vary. Another opportunity for flexibility arises when the result type of a method is, recursively, its class type. In this section we consider a language construct devised for specializing such methods.

Class definitions are often recursive, in the sense that the definition of a class c may contain occurrences of $\text{InstanceTypeOf}(c)$. For example, we could have a class c containing a method m with result type $\text{InstanceTypeOf}(c)$:

```
class c is
    var x: Integer := 0;
    method m(): InstanceTypeOf(c) is ... self ... end;
end;

subclass c' of c is
    var y: Integer := 0;
end;
```

On inheritance, recursive types are, by default, preserved exactly, just as other types are. For instance, for o' of class c' , we have that $o'.m()$ has type $\text{InstanceTypeOf}(c)$ and not, for example, $\text{InstanceTypeOf}(c')$. In general, adopting $\text{InstanceTypeOf}(c')$ as the result type for the inherited method m in c' is unsound, because m may construct and return an instance of c that is not an instance of c' .

Suppose, though, that m returns *self*, perhaps after modifying the field x . Then it would be sound to give the inherited method the result type $\text{InstanceTypeOf}(c')$, since *self* is always bound to the receiver of invocations. With this more precise typing, we would avoid subsequent uses of **typecase**: limiting the result type to $\text{InstanceTypeOf}(c)$ constitutes an unwarranted loss of information.

This argument leads to the notion of *Self types*. The keyword **Self** represents the type of **self**. Instead of assigning the result type *InstanceTypeOf(c)* to *m*, we now write:

```
class c is
    var x: Integer := 0;
    method m(): Self is ... self ... end;
end;
```

The typing of the code of *m* relies on the assumptions that **Self** is a subtype of *InstanceTypeOf(c)*, and that **self** has type **Self**. Field updates to **self** preserve its **Self** type. For soundness, the result of *m* must be shown to have type **Self**, and not just type *InstanceTypeOf(c)*.

When *c'* is declared as a subclass of *c*, the result type of *m* is still taken to be **Self**. However, **Self** is then regarded as a subtype of *InstanceTypeOf(c')*. Thus **Self**, as the result type of a method, is automatically specialized on subclassing.

There are no drawbacks to extending classical class-based languages with **Self** in covariant positions, for example as the result type of methods. This extension increases expressive power and prevents loss of type information at no cost other than properly keeping track of the type of **self**. We can even allow **Self** as the type of fields; this is sound as long as those fields are updated only with **self** or updated versions of **self**.

A natural next step is to allow **Self** in contravariant (argument) positions. This is what Eiffel set out to do [88]. Unfortunately, contravariant uses of **Self** are unsound for subsumption as can be demonstrated by simple counterexamples [55]. As we have seen, types in argument positions of inherited methods may be generalized, not specialized.

The proper handling of **Self** in contravariant positions is a major new development in class-based languages, and goes well beyond their classical features. We discuss it in Sections 3.4 and 3.5.

3

ADVANCED CLASS-BASED FEATURES

One of the main characteristics of classical class-based languages is the strict correlation between inheritance, subclassing, and subtyping. A great economy of concepts and syntax is achieved by identifying these three relations. The identification also confers much flexibility in the use of subsumption: an object of a subclass, some of whose methods may have been inherited, can always be used in place of an object of a superclass by virtue of subtyping.

There are situations, however, in which inheritance, subclassing, and subtyping conflict. Opportunities for code reuse, both by inheritance and by parameterization, turn out to be limited by the coincidence of these relations. Therefore considerable attention has been devoted to separating them. The separation of subclassing from subtyping is becoming commonplace; other separations are more tentative. In this chapter we examine various possible distinctions.

3.1 Object Types

In the original formulation of classes (in Simula, for example), the type description of objects is intermixed with the implementation of methods. This situation conflicts with the now widely recognized advantages of keeping specifications separate from implementations, particularly to enable separate code development within large teams of programmers.

A relatively recent variation on the classical model addresses this problem. Separation between object specification and object implementation can be achieved by introducing types for objects that are independent of specific classes [19, 113]. This approach is supported by Modula-3 and by other languages that provide both classes and interfaces.

When we first discussed the type *InstanceTypeOf(cell)*, its meaning appeared rather limited. That type seemed to indicate that all its elements were obtained by executing **new** on the *cell* class, and therefore that they all had identical methods. Later, by the combination of subclassing, method overriding, subsumption, and dynamic dispatch, we obtained objects of type *InstanceTypeOf(cell)* that had additional attributes and different method code. It is therefore peculiar that the type *InstanceTypeOf(cell)* should depend explicitly on an entity, the class *cell*, that describes some specific method code. None of the code from class *cell* is necessarily found in members of *InstanceTypeOf(cell)*.

In general, elements of *InstanceTypeOf(cell)* have in common only the type signature of the attributes specified in *cell*; this is sometimes called the *object protocol* for cells.

It seems natural to take such a protocol as the type of instances of *cell*. Recall the classes *cell* and *reCell*:

```
class cell is
    var contents: Integer := 0;
    method get(): Integer is return self.contents end;
    method set(n: Integer) is self.contents := n end;
end;

subklass reCell of cell is
    var backup: Integer := 0;
    override set(n: Integer) is
        self.backup := self.contents;
        super.set(n);
    end;
    method restore() is self.contents := self.backup end;
end;
```

We introduce two object types *Cell* and *ReCell* that correspond to these classes. We write them as independent types, but we could introduce syntax to avoid repeating common components.

```
ObjectType Cell is
    var contents: Integer;
    method get(): Integer;
    method set(n: Integer);
end;

ObjectType ReCell is
    var contents: Integer;
    var backup: Integer;
    method get(): Integer;
    method set(n: Integer);
    method restore();
end;
```

These types list attributes and their types, but not their implementations. They are suitable to appear in interfaces, and to be implemented separately and in more than one way.

We use *ObjectTypeOf(cell)* as a meta-notation for the object type *Cell*. This type can be mechanically extracted from class *cell*. Therefore we may write either *o : ObjectTypeOf(cell)* or *o : Cell*. The main property we expect of *ObjectTypeOf* is that:

new c : ObjectTypeOf(c) for any class *c*

Different classes *cell* and *cell₁* may happen to produce the same object type *Cell*, equal to both *ObjectTypeOf(cell)* and *ObjectTypeOf(cell₁)*. Therefore objects having type *Cell* are required only to satisfy a certain protocol, independently of attribute implementation.

3.2 Distinguishing Subclassing from Subtyping

The subtype relation was previously based on the subclass relation. When object types are independent of classes, we must provide an independent definition. There are several choices: whether subtyping is determined by type structure or by type names in declarations, and in the former case what parts of the structure of types matter.

Structural subtyping (subtyping determined by type structure) has desirable properties, such as supporting type matching in distributed and persistent systems [15, 95]. A disadvantage is the possibility of accidental matching of unrelated types. However, one can avoid such accidents by imposing distinctions on top of structural subtyping [95]. In contrast, subtyping based on type names is hard to define precisely, and does not support structural subtyping.

For our present purposes we use a particularly simple form of structural subtyping. We assume, for two object types *O* and *O'*, that:

$$O' <: O \quad \text{if } O' \text{ has the same components as } O \text{ and possibly more}$$

where a component of an object type is the name of a field or a method and its associated type. So, for example, *ReCell* <: *Cell*.

This definition implies that object types are invariant in their component types, although we could extend it to allow method specialization as discussed in Section 2.7. When object types are defined recursively we need to be more careful about the definition of subtyping; we defer this discussion to Chapter 9 and beyond.

With this definition of subtyping, object types naturally support multiple subtyping, because components are assumed unordered. For example, consider the object type:

```
ObjectType ReInteger is
  var contents: Integer;
  var backup: Integer;
  method restore();
end;
```

Then we have both *ReCell* <: *Cell* and *ReCell* <: *ReInteger*.

As a consequence of the new definition of subtyping we have:

$$(4) \text{ If } c' \text{ is a subclass of } c, \text{ then } \text{ObjectTypeOf}(c') <: \text{ObjectTypeOf}(c).$$

This holds simply because a subclass can only add new attributes to a class, and because we require that overriding methods preserve the existing method types.

Property (4) is a reformulation for *ObjectTypeOf* of property (3) of Section 2.4, the subclassing-is-subtyping property. While property (3) is a double implication, the converse of (4) does not hold: there may be unrelated c and c' such that $\text{ObjectTypeOf}(c) = O$ and $\text{ObjectTypeOf}(c') = O'$, with $O' <: O$.

Therefore we have partially decoupled subclassing from subtyping by relaxing the double implication (3) to the single implication (4). Subclassing still implies subtyping, so all the previous uses of subsumption are still allowed. But, since subsumption is based on subtyping and not subclassing, we now have even more freedom in subsumption.

In conclusion, the notion of subclassing-is-subtyping can be weakened to subclassing-implies-subtyping without loss of expressiveness, and with a gain in separation between interfaces and implementations.

3.3 Type Parameters

Type parameterization is a general technique for reusing the same piece of code at different types. It is becoming common in modern object-oriented languages, in part independently of object-oriented features. We describe it here both because it is interesting in its own right, and because it is used for the treatment of binary methods later in this chapter.

In conjunction with subtyping, type parameterization can be used to remedy some typing difficulties due to contravariance, for example in method specialization. Consider the object types *Person* and *Vegetarian*, and assume *Vegetables* $<: \text{Food}$ (but not vice versa) [34]:

```
ObjectType Person is
...
    method eat( food: Food);
end;
ObjectType Vegetarian is
...
    method eat( food: Vegetables);
end;
```

The intention is that a vegetarian is a person, so we would expect *Vegetarian* $<: \text{Person}$. However, this inclusion cannot hold because of the contravariance on the argument of the *eat* method. If we erroneously assume *Vegetarian* $<: \text{Person}$, then a vegetarian can be subsumed into *Person*, and can be made to eat meat.

We can obtain some legal subsumptions between vegetarians and persons by converting the corresponding object types into type operators (functions from types to types) parameterized on the type of *food*:

```
ObjectOperator PersonEating[ $F <: \text{Food}$ ] is
...
    method eat( $\text{food}: F$ );
end;

ObjectOperator VegetarianEating[ $F <: \text{Vegetables}$ ] is
...
    method eat( $\text{food}: F$ );
end;
```

The mechanism used here is called *bounded type parameterization*.¹⁵ The variable F is a type parameter, which can be instantiated with a type. A bound such as $F <: \text{Vegetables}$ limits the possible instantiations of F to subtypes of *Vegetables*. So *VegetarianEating[Vegetables]* is a type; in contrast, *VegetarianEating[Food]* is not well-formed. The type *VegetarianEating[Vegetables]* is an instance of *VegetarianEating*, and is equal to the type *Vegetarian*.

There are no elements of operators such as *PersonEating*; there are only elements of types *PersonEating[F]* for some given F . Therefore we do not have any inclusion relation between the operators *VegetarianEating* and *PersonEating*. However, we do have that:

for all $F <: \text{Vegetables}$, *VegetarianEating[F]* $<: \text{PersonEating}[F]$

because, for any $F <: \text{Vegetables}$, the two instances are included by the usual rules for subtyping. In particular, we obtain *Vegetarian* = *VegetarianEating[Vegetables]* $<: \text{PersonEating[Vegetables]}$. This inclusion can be useful for subsumption: it asserts, correctly, that a vegetarian is a person that eats only vegetables.

Related to bounded type parameters are *bounded abstract types* (also called *partially abstract types*).¹⁶ Bounded abstract types offer a different solution to the problem of making *Vegetarian* a subtype of *Person*. We redefine our object types by adding the F parameter, subtype of *Food*, as one of the attributes:

```
ObjectType Person is
    type  $F <: \text{Food}$ ;
...
    var lunch:  $F$ ;
    method eat( $\text{food}: F$ );
end;
```

¹⁵ Languages that support bounded type parameters include Trellis/Owl, Sather, Eiffel, PolyTOIL, and Rapide. Other languages, such as C++, support simple type parameters without bounds.

¹⁶ Languages that support forms of bounded abstract types include Rapide, Modula-3 through opaque types at the module level, and Beta through virtual classes [85].

```
ObjectType Vegetarian is
  type F <: Vegetables;
  ...
  var lunch: F;
  method eat(food: F);
end;
```

The meaning of the type component $F <: Food$ in *Person* is that, given a person, we know that it can eat some *Food*, but we do not know exactly of what kind. The *lunch* attribute provides some food that a person can eat.

We can build an object of type *Person* by choosing a specific subtype of *Food*, for example $F = Dessert$, picking a dessert for the *lunch* field, and implementing a method with parameter of type *Dessert*. We have that the resulting object is a *Person*, by forgetting the specific F that we chose for its implementation.

Now the inclusion $Vegetarian <: Person$ holds. A vegetarian subsumed into *Person* can be safely fed the lunch it carries with it, because the vegetarian was constructed with $F <: Vegetables$. A limitation of this approach is that a person can be fed only the food it carries with it as a component of type F , and not food obtained independently.

3.4 Subclassing without Subtyping

We have seen in Section 3.2 how the partial decoupling of subtyping from subclassing increases the opportunities for subsumption. Another approach has emerged that increases the potential for inheritance by further separating subtyping from subclassing. This approach abandons completely the notion that subclassing implies subtyping (property (4)), and is known under the name *inheritance-is-not-subtyping* [52]. It is largely motivated by the desire to handle contravariant (argument) occurrences of **Self** so as to allow inheritance of methods with arguments of type **Self**; these methods arise naturally in realistic examples. The price paid for this added flexibility in inheritance is decreased flexibility in subsumption. When **Self** is used liberally in contravariant positions, subclasses do not necessarily induce subtypes.

Consider two types *Max* and *MinMax* for integers enriched with *min* and *max* methods. Each of these types is defined recursively:

```
ObjectType Max is
  var n: Integer;
  method max(other: Max): Max;
end;

ObjectType MinMax is
  var n: Integer;
  method max(other: MinMax): MinMax;
  method min(other: MinMax): MinMax;
end;
```

Consider also two classes:

```
class maxClass is
    var n: Integer := 0;
    method max(other: Self): Self is
        if self.n > other.n then return self else return other end;
    end;
end;

subclass minMaxClass of maxClass is
    method min(other: Self): Self is
        if self.n < other.n then return self else return other end;
    end;
end;
```

The methods *min* and *max* are called *binary* because they operate on two objects: **self** and *other*; the type of *other* is given by a contravariant occurrence of **Self** [36]. Notice that the method *max*, which has an argument of type **Self**, is inherited from *maxClass* to *minMaxClass*.

Intuitively, the type *Max* corresponds to the class *maxClass*, and *MinMax* to *minMaxClass*. To make this correspondence more precise, we must define the meaning of *ObjectTypeOf* for classes containing occurrences of **Self**, so as to obtain *ObjectTypeOf(maxClass) = Max* and *ObjectTypeOf(minMaxClass) = MinMax*. For these equations to hold, we map the use of **Self** in a class to the use of recursion in an object type. We also implicitly specialize **Self** for inherited methods; for example, we map the use of **Self** in the inherited method *max* to *MinMax*. In short, we obtain that any instance of *maxClass* has type *Max*, and any instance of *minMaxClass* has type *MinMax*.

Although *minMaxClass* is a subclass of *maxClass*, *MinMax* cannot be a subtype of *Max*. Consider the class:

```
subclass minMaxClass' of minMaxClass is
    override max(other: Self): Self is
        if other.min(self) = other then return self else return other end;
    end;
end;
```

For any instance *mm'* of *minMaxClass'* we have *mm' : MinMax*. If *MinMax* were a subtype of *Max*, then we would have also *mm' : Max*, and *mm'.max(m)* would be allowed for any *m* of type *Max*. Since *m* may not have a *min* attribute, the overridden *max* method of *mm'* may break. Therefore:

MinMax <: Max does not hold

Thus subclasses with contravariant occurrences of **Self** do not always induce subtypes.

3.5 Object Protocols

Even when subclasses do not induce subtypes, we can find a relation between the type induced by a class and the type induced by one of its subclasses. It just so happens that, unlike subtyping, this relation does not enjoy the subsumption property. We now examine this new relation between object types.

In the example of Section 3.4, we cannot usefully quantify over the subtypes of *Max* because of the failure of subtyping. A parametric definition such as:

ObjectOperator $P[M <: Max]$ is ... end;

is not very useful; we can instantiate P to $P[Max]$, but $P[MinMax]$ is not well-formed.

Still, any object that supports the *MinMax* protocol, in an intuitive sense, also supports the *Max* protocol. There seems to be an opportunity for some kind of *subprotocol* relation that may allow useful parameterization. In order to find this subprotocol relation, we introduce two type operators, *MaxProtocol* and *MinMaxProtocol*:

```
ObjectOperator MaxProtocol[X] is
    var n: Integer;
    method max(other: X): X;
end;

ObjectOperator MinMaxProtocol[X] is
    var n: Integer;
    method max(other: X): X;
    method min(other: X): X;
end;
```

Generalizing from this example, we can always pass uniformly from a recursive type T to an operator $T\text{-Protocol}$ by abstracting over the recursive occurrences of T . The operator $T\text{-Protocol}$ is a function on types; taking the fixpoint of $T\text{-Protocol}$ yields back T .

We find two formal relationships between *Max* and *MinMax*. First, *MinMax* is a post-fixpoint of *MaxProtocol*, that is:

$\text{MinMax} <: \text{MaxProtocol}[\text{MinMax}]$

Second, let $<$ denote the higher-order subtype relation between type operators:

$P <: P'$ iff $P[T] <: P'[T]$ for all types T

Then the protocols of *Max* and *MinMax* satisfy:

$\text{MinMaxProtocol} <: \text{MaxProtocol}$

Either of these two relationships can be taken as a basis for the notion of subprotocol:

S subprotocol T if $S <: T\text{-Protocol}[S]$

or

S subprotocol T if $S\text{-Protocol} <: T\text{-Protocol}$

The second relationship expresses a bit more directly the fact that there exists a subprotocol relation, and that this is in fact a relation between operators, not between types.

Whenever we have some property common to several types, we may think of parameterizing over these types. So we may adopt one of the following forms of parameterization:

ObjectOperator $P_1[X <: \text{MaxProtocol}[X]]$ is ... end;
ObjectOperator $P_2[P <: \text{MaxProtocol}]$ is ... end;

Then we can instantiate P_1 to $P_1[\text{MinMax}]$, and P_2 to $P_2[\text{MinMaxProtocol}]$.

These two forms of parameterization seem to be equally expressive in practice. The first one is called *F-bounded parameterization* [35, 38, 75]. The second form is *higher-order bounded parameterization*, defined via pointwise subtyping of type operators; we treat it formally in Part III. See [4] for a comparison between these two forms of parameterization.

Instead of working with type operators, a programming language supporting subprotocols may conveniently define a *matching* relation (denoted by $<\#$) directly over types.¹⁷ The properties of the matching relation are designed to correspond to the definition of subprotocol. Depending on the choice of subprotocol relation, we have:

$S <\# T$ if $S <: T\text{-Protocol}[S]$	(F-bounded interpretation)
or	
$S <\# T$ if $S\text{-Protocol} <: T\text{-Protocol}$	(higher-order interpretation)

With either definition we have $\text{MinMax} <\# \text{Max}$.

Matching does not enjoy a subsumption property (that is, $S <\# T$ and $s : S$ do not imply that $s : T$); however, matching is useful for parameterizing over all the types that match a given one:

ObjectOperator $P_3[X <\# \text{Max}]$ is ... end;

The instantiation $P_3[\text{MinMax}]$ is legal.

In summary, even in the presence of contravariant occurrences of **Self**, and in absence of subtyping, there can be inheritance of binary methods like *max*. Unfortunately, subsumption is lost in this context, and quantification over subtypes is no longer very useful. These disadvantages are partially compensated by the existence of a subprotocol relation, and by the ability to parameterize with respect to this relation.

¹⁷ The concept of matching and its name are taken from PolyTOIL; Theta and School have similar notions.

4 OBJECT-BASED LANGUAGES

The main features of class-based languages appeared fully formed in Simula. Object-based languages, in contrast, have emerged more gradually. Their main principles were first gathered in the Treaty of Orlando [114]; to a large extent, even their most basic notions are still evolving [118].

Object-based languages are intended to be both simpler and more flexible than traditional class-based languages. Many object-based languages originated in the Lisp, Smalltalk, and artificial intelligence communities, where extreme flexibility is highly valued [11, 31, 32, 78, 80]. As a consequence, little attention has been devoted to designing typed object-based languages, except for simple ones such as Emerald, and for recent ones such as Cecil and Omega. In this chapter we review the main features of object-based languages, considering issues of inheritance and typing.

4.1 Objects without Classes

Object-based languages are refreshing in their simplicity, compared with even the simplest class-based languages. In their minimal form, as in Emerald and in simple closure-based encodings [11], they support only the notions of objects and dynamic dispatch. When typed, they support only object types, subtyping, and subsumption.

The basic characteristics of object-based languages are the absence of classes and the existence of constructs for the creation of individual objects. Since there are no classes it is natural to introduce object types immediately, and to separate object interfaces from object implementations. For example, we may have:

```
ObjectType Cell is
    var contents: Integer;
    method get(): Integer;
    method set(n: Integer);
end;
object cell: Cell is
    var contents: Integer := 0;
    method get(): Integer is return self.contents end;
    method set(n: Integer) is self.contents := n end;
end;
```

This **object** definition creates a single object of type *Cell* and names it *cell*.

Even without classes, we can generate a uniform collection of objects by a procedure that, whenever invoked, returns a new object. Such a procedure can have parameters, for example to initialize fields:

```
procedure newCell(m: Integer): Cell is
  object cell: Cell is
    var contents: Integer := m;
    method get(): Integer is return self.contents end;
    method set(n: Integer) is self.contents := n end;
  end;
  return cell;
end;
var cellInstance: Cell := newCell(0);
```

Such object-generating procedures are uncannily similar to Simula classes, which also have parameters and an executable body. The role of **new** is taken over by procedure call.

The main insight of the object-based model is that class-based notions need not be assumed, but instead can be emulated by more primitive notions. Moreover, these more primitive notions can be combined in more flexible ways than in a strict class discipline. A similar reduction of classes to objects will appear in our formal treatment of object calculi.

4.2 Prototypes and Clones

As we have just seen, procedures may be used for generating objects. However, it may be difficult or inconvenient to anticipate all the possible ways in which objects should be parameterized. The so-called *prototype-based* languages adopt a different approach to object generation. Instead of parameterizing objects beforehand, they generate stock objects from prototypical objects, and customize the stock objects later [32, 79].

In class-based languages, object descriptions provide the templates from which object instances are generated. In prototype-based languages, instead, concrete and fully functional instances are built first. Some of these instances may be later interpreted as canonical representatives (*prototypes*) of classes of instances. Additional instances are derived from the prototypes. There is no entity that represents a class of instances, other than by convention.

The basic mechanism used for instantiating prototypes is *cloning*. Cloning produces a *shallow copy* of an object; that is, a copy of the outermost object structure, sharing all the attribute values of the original object, but with independent state.

```
var cellClone: Cell := clone cellInstance;
```

Cloning is a bit like **new**, but operates on objects instead of classes. Any object can act both as an instance and as a prototype for further cloning. An object in the role of a

prototype acts much like a class for the objects cloned from it, and much like a superclass for the prototypes cloned from it. Unlike a class, however, a prototype is a fully functional object in its own right.¹⁸

Cloning would not be of much use without some way of mutating (customizing) the clones so that they can differ from their prototypes. In the simplest case, we may customize a field:

```
cellClone.contents := 3;
```

More generally, we may want to customize the behavior of a clone by modifying its methods. The dynamic modification of the behavior of individual objects is a unique property of object-based languages. We could write:

```
cellClone.get :=
    method (): Integer is
        if self.contents < 0 then return 0 else return self.contents end;
    end;
```

replacing the method *get* of a particular object with a method that never returns negative numbers.

This method update operation can be seen as an elementary example of a rather rich collection of operations that are used for mutating clones [27]. With respect to those, method update has the advantage of being simple and statically typable, as we shall see in great detail.¹⁹

As a further illustration of method update, we reformulate the *reCell* example from Section 2.3 in object-based style:

```
ObjectType ReCell is
    var contents: Integer;
    method get(): Integer;
    method set(n: Integer);
    method restore();
end;

object reCell: ReCell is
    var contents: Integer := 0;
    method get(): Integer is return self.contents end;
```

¹⁸ There is an equivalent of *InstanceTypeOf(c)* for prototype-based languages. Let *DescendentOf(o)* be the collection of the objects cloned, directly or indirectly, from *o*. The concept of *DescendentOf* is used in Cecil to constrain the parameters of methods at run-time.

¹⁹ Method update is directly supported by some languages (Beta, NewtonScript, Obliq, Kevo, Garnet). Other languages allow dynamically removing and adding attributes (Self, Act1), or replacing groups of methods (Self, Ellie), much to the same effect.

```

method set(n: Integer) is
  let x = self.get();
  self.restore := method () is self.contents := x end;
  self.contents := n;
end;
method restore() is self.contents := 0 end;
end;

```

Here the *set* method updates the *restore* method; there is no need for a separate *backup* field.

In the rest of this chapter we discuss the various ways in which clones may be derived from prototypes, and how clones may mutate.

4.3 Inheritance by Embedding and by Delegation

Simple object-based languages such as Emerald use procedures for generating objects that share common behavior, but these languages have no inheritance mechanism.

Prototype-based languages also have a generation mechanism, plus a way of updating individual objects. Cloning can be seen, in part, as inheritance (in the sense of method reuse), and update can be seen as overriding. Cloning and update, however, do not provide sufficiently flexible forms of inheritance, because they always preserve the shape of a prototype. In general, we may want to inherit the behavior of an object when building one of a different shape.

A natural inheritance mechanism to consider is the extension of existing objects with new attributes. Object extension, however, is not a simple operation. One problem is the potential for clashes of the additional attributes with existing (but unknown) ones; careful handling is required in typed languages [43, 61, 70, 126]. Another problem is the necessity of modifying the representation of an object in order to extend it, while maintaining the object's identity (which is often determined by the memory address of the representation). Because of these difficulties, direct attribute extension is rarely provided as a language construct; it is either found in specialized object models [14, 16], or it is intended to be used in restricted contexts [12].²⁰

An alternative to extending existing objects is to build new objects that include some attributes of existing ones. This is in fact the most common inheritance mechanism for prototype-based languages, and the one we discuss in detail.

There are two orthogonal aspects to inheritance in prototype-based languages: obtaining the attributes of a *donor* object, and incorporating those attributes into a new *host* object. Along these dimensions, prototype-based languages can be separated into

²⁰ Omega and Kevo support object extension, but essentially as an environment operation that applies to all existing objects of a certain category. Sometimes the construction of extended prototypes is considered external to the language and relegated to the user interface [26, 32].

four categories. The attributes of a donor may be obtained *implicitly* or *explicitly* and, orthogonally, those attributes may be either *embedded* into a host or *delegated* to a donor.

In implicit inheritance, one or more objects are designated as the donors, and their attributes are implicitly inherited. In explicit inheritance, individual attributes of one or more donors are explicitly designated. An intermediate possibility is to explicitly designate a named collection of attributes that, however, does not form a whole object: this is *mixin* inheritance [33]. We generally consider only implicit and explicit inheritance as the extreme points of a spectrum.

More important is the distinction between embedding and delegation inheritance. With embedding, the attributes inherited from a donor become part of the host (in principle, at least; various optimizations can be applied) [32, 117, 118]. With delegation, the inherited attributes remain part of the donor, and are accessed via an indirection from the host.

In both embedding and delegation the meaning of **self** is the usual one: it is the original receiver of an invocation. The semantic differences between the two forms of inheritance concern only how changes to a donor object affect the host objects that inherit its attributes. In embedding, host objects are independent of their donors. In delegation, complex webs of dependencies may be created.

4.4 Embedding

One possible interpretation of inheritance is that host objects contain copies of the attributes of donor objects. We call this the embedding interpretation of inheritance, or simply embedding (Figure 4-1). The embedding interpretation is appealing because it provides the simplest explanation of the standard semantics of **self** as the receiver of an invocation. The invocation of an inherited method, under the embedding interpretation, works exactly like the invocation of an original method.

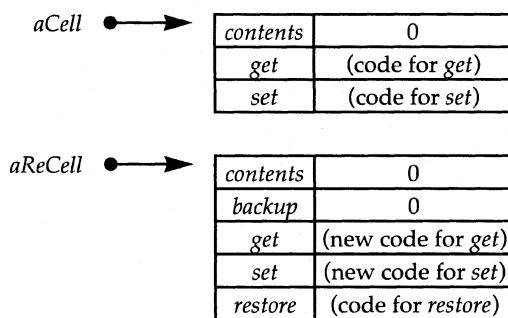


Figure 4-1. Embedding.

Embedding was described as part of one of the first proposals for prototype-based languages [31]. Only recently, though, it has been adopted by languages such as Kevo and Obliq. We call these languages *embedding-based*.

As we discussed in the previous section, inheritance can be specified explicitly or implicitly. Explicit forms of embedding inheritance can be understood as reassembling parts of old objects into new objects. Implicit forms of embedding inheritance can be understood as ways of extending copies of existing objects with new attributes.

In the explicit version of embedding, we designate methods and fields to be copied from other specific objects. For instance, we might write our restorable cell example as follows:

```
object cell: Cell is
    var contents: Integer := 0;
    method get(): Integer is return self.contents end;
    method set(n: Integer) is self.contents := n end;
end;

object reCellExp: ReCell is
    var contents: Integer := cell.contents;
    var backup: Integer := 0;
    method get(): Integer is
        return embed cell.get();
    end;
    method set(n: Integer) is
        self.backup := self.contents;
        embed cell.set(n);
    end;
    method restore() is self.contents := self.backup end;
end;
```

We write **embed** *o.m(...)* to embed the method *m* of object *o* into the current object; this embedding construction is used to replace both the implicit inheritance of the *get* method and an occurrence of *super.set(n)* in the *set* method. The meaning of **embed** *cell.set(n)* is to execute the *set* method of *cell* with **self** bound to the current **self**, and not with **self** bound to *cell* as in a normal invocation *cell.set(n)*. Moreover, the code of *set* is embedded in *reCellExp*, so it is not affected by updates to the *cell* object.

The code for the *get* method is rather verbose, given that its only purpose is to redirect an invocation. In practice, we could adopt an abbreviation such as:

```
method get copied from cell;
```

However, we still need the general **embed** construct for the *set* method.

To avoid typing problems, we need to assume that the object that follows **embed** is statically known (the object itself, not just its type). In this sense, the situation is anal-

ogous to that in a subclass definition, where we assume that the superclass is statically known.

In the implicit version of embedding, which we discuss next, we may designate a particular object as the donor of methods and fields to be copied into a new object. For instance, we may write the restorable cell example as follows:

```
object cell: Cell is
    var contents: Integer := 0;
    method get(): Integer is return self.contents end;
    method set(n: Integer) is self.contents := n end;
end;

object reCellImp: ReCell extends cell is
    var backup: Integer := 0;
    override set(n: Integer) is
        self.backup := self.contents;
        embed cell.set(n);
end;
    method restore() is self.contents := self.backup end;
end;
```

These definitions are not much different from the corresponding class definition of Sections 2.1 and 2.3. The **extends** declaration designates the donor object *cell* for *reCellImp*; we assume that the object that follows **extends** is statically known. As a consequence of this declaration, *reCellImp* is an object containing a copy of the attributes of *cell*, with independent state. The effect of **override** is to replace a method of the donor in the host; note that the **embed** construct is still needed.

Since objects are independent of their donors, the definitions of both *reCellImp* and *reCellExp* can be seen as convenient abbreviations for a stand-alone definition:

```
object reCell: ReCell is
    var contents: Integer := 0;
    var backup: Integer := 0;
    method get(): Integer is return self.contents end;
    method set(n: Integer) is
        self.backup := self.contents;
        self.contents := n;
end;
    method restore() is self.contents := self.backup end;
end;
```

Alternatively, we could define an equivalent object by a pure extension of *cell* followed by a method update.

```

object reCellImpl1: ReCell extends cell is
    var backup: Integer := 0;
    method restore() is self.contents := self.backup end;
end;
reCellImpl1.set :=
    method (n: Integer) is
        self.backup := self.contents;
        self.contents := n;
    end;

```

This code works because, with embedding, method update affects only the object to which it is applied.

4.5 Delegation

Prototype-based languages that permit the sharing of attributes across objects are called delegation-based. In operational terms, delegation is the redirection of field access and method invocation from an object or prototype to another, in such a way that an object can be seen as an extension of another.

In the most common forms of implicit delegation inheritance [49, 121], a child object (the host) designates another object as its parent (the donor). In implementation terms, the child contains a *parent link* (a reference to its parent) through which the attributes of the parent can be obtained (Figure 4-2). When an attribute is not found in an object, its parent is searched.

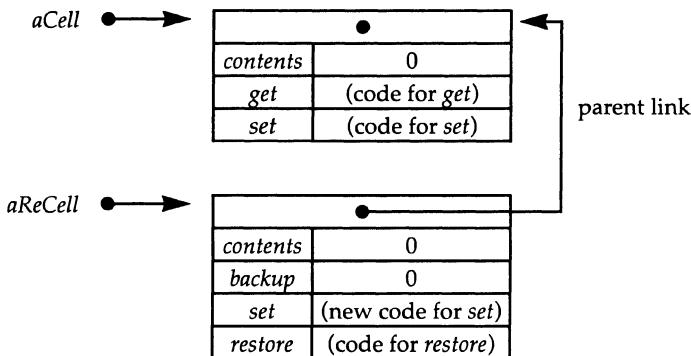


Figure 4-2. Delegation.

As a first attempt towards a proper definition of *reCellImpl* in delegation style, we define *reCellImpl'* as a child of *cell*:

```

object cell: Cell is
  var contents: Integer := 0;
  method get(): Integer is return self.contents end;
  method set(n: Integer) is self.contents := n end;
end;

object reCellImp': ReCell child of cell is
  var backup: Integer := 0;
  override set(n: Integer) is
    self.backup := self.contents;
    delegate cell.set(n);
  end;
  method restore() is self.contents := self.backup end;
end;

```

Instead of writing `super.set(n)` we now write `delegate cell.set(n)`. The meaning of `delegate cell.set(n)` is to execute the `set` method of `cell` with `self` bound to the current `self`. The difference between `delegate` and `embed` is that the former obtains the method from the parent at the time of invocation, whereas the latter obtains it earlier, at the time of object creation.

This definition of `reCellImp'` is similar to the definition of `reCellImp` for embedding, in Section 4.4. The meaning is, however, different: `reCellImp'` does not contain a copy of the attributes of `cell`, but instead it contains a parent link. Because of parent links, the attributes of the `cell` object are shared by all its children. Therefore the preceding definition of `reCellImp'` is wrong: a single `contents` field becomes shared by all children of `cell` and, in particular, by all clones of `reCellImp'`.

A better definition of a restorable cell would include a local copy of the `contents` field, overriding the `contents` field of the parent. Hence we define `reCellImp` as follows:

```

object reCellImp: ReCell child of cell is
  override contents: Integer := cell.contents;
  var backup: Integer := 0;
  override set(n: Integer) is
    self.backup := self.contents;
    delegate cell.set(n);
  end;
  method restore() is self.contents := self.backup end;
end;

```

A crucial aspect of delegation inheritance is the interaction of parent links with the binding of `self`. On an invocation of `reCellImp.get()`, the `get` method is found only in the parent `cell`, but the occurrences of `self` within the code of `get` refer to the original

receiver, *reCellImp*, and not to the parent, *cell*. Hence the result of *get()* is, as desired, the integer stored in the *contents* field of *reCellImp*.

As with embedding, we assume that, when a child is constructed by delegation, its parent is statically known. This restriction, adopted by statically-typed delegation-based languages such as Cecil, is important for checking the compatibility of the child with the parent. Knowing the type of the parent is not sufficient for soundness. For example, if the *contents* field of *cell* is first forgotten by subsumption, and a child is defined with a *contents* field having type *Boolean*, say, then the inherited *get* method returns a *Boolean* when invoked via the child because of the interpretation of *self* as the original receiver. This result is wrong because the declared result type of the *get* method is *Integer*.

More generally, a parent may be an unknown object as long as its true type is statically known, somehow. Alternatively, it must be statically known that the parent does not have certain attributes. These options can be handled in sophisticated type systems with negative information [43, 61, 70, 126]. For simplicity, we assume that parent objects are statically known.

We turn briefly to explicit delegation inheritance, where an object delegates individual methods explicitly to other objects [80] and where there are no parent declarations.²¹ We may write:

```
object reCellExp: ReCell is
    var contents: Integer := cell.contents;
    var backup: Integer := 0;
    method get(): Integer is return delegate cell.get() end;
    method set(n: Integer) is
        self.backup := self.contents;
        delegate cell.set(n);
    end;
    method restore() is self.contents := self.backup end;
end;
```

Explicit delegation provides a clean way of delegating operations to multiple objects. Multiple delegation is sometimes allowed in the implicit delegation model, by listing multiple parents. As in the case of implicit multiple inheritance, this is problematic because it introduces ambiguities in attribute lookup. Multiple parents can be more cleanly represented in terms of explicit delegation.²²

²¹ Ellie has both explicit embedding and explicit delegation. Self is primarily based on implicit delegation, but also supports a primitive for explicit delegation.

²² A mechanism similar to explicit multiple delegation can be found in Ellie. However, this mechanism does not preserve *self* as the original receiver of an invocation, and therefore it is redirection rather than delegation.

4.6 Embedding versus Delegation

In embedding inheritance, a freshly created host object contains copies of donor attributes. Access to the inherited donor attributes is then no different from access to original attributes.

The situation is not so simple in delegation inheritance, where a host object contains links to external donor objects. During method invocation, the attribute-lookup procedure must preserve the binding of `self` to the original receiver, even while following the donor links. This requirement complicates both the implementation and the formal modeling of method lookup.

In class-based languages the embedding and delegation models are normally equivalent. In object-based languages they are distinguishable. In particular, the sharing of donors typical of delegation can be exposed in several ways.

- In delegation, donors may contain fields, which may be updated; the changes are seen by the inheriting hosts.
- Similarly, the methods of a donor may be updated, and again the changes are seen by the inheriting hosts.
- It is often permitted to replace a donor link with another one in an object; then all the inheritors of that object may change behavior.
- Cloning is still taken to perform shallow copies of objects, without copying the corresponding donors. So all clones of an object come to share its donors and therefore the mutable fields and methods of the donors.

Thus embedding and delegation are two fundamentally distinct ways of achieving inheritance with prototypes; interesting languages exist that explore both possibilities.

Delegation is the most common of the two. An advantage of delegation is convenience in performing pervasive changes to all inheritors of an object. This mechanism for change fits well with an approach to programming where language and environment are integrated, and where program execution and development are unified.

Arguments have been proposed in favor of embedding over delegation [59, 117]. Delegation can be criticized because it creates dynamic webs of dependencies that lead to fragile systems. Embedding is not affected by this problem because objects remain autonomous. In embedding-based languages such as Kevo and Omega, pervasive changes are achieved even without donor hierarchies.

The space efficiency obtained by sharing is one of the main motivations for delegation mechanisms [114]. On the other hand, it has been argued that space efficiency, while essential, is best achieved behind the scenes of the implementation [32, 59]. Even delegation-based languages optimize cloning operations by transparently sharing structures [50]; the same techniques can be used to optimize the space utilization of embedding-based languages [59, 118].

In conclusion, the question of what is the best model of inheritance for object-based languages is still open.

4.7 Dynamic Inheritance and Mode-Switching

Delegation inheritance is called *static* when parent links are fixed for all time, and *dynamic* when parent links can be updated dynamically. In the delegation-based language Self, for example, a parent link can be declared as variable; then the parent of an object can be dynamically replaced with any another object (Figure 4-3).²³ Dynamic delegation is also called *dynamic inheritance*.

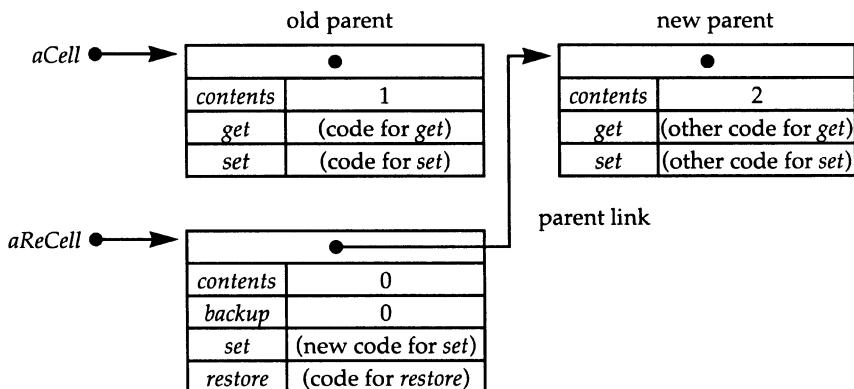


Figure 4-3. Reparenting.

Although dynamic inheritance is in general a dangerous feature, it enables rather elegant and disciplined programming techniques. In particular, *mode-switching* is the special case of dynamic inheritance where a parent is replaced with an object having the same attributes.²⁴ Mode-switching provides a way of modularly switching the behavior of an object from one mode or role to another [119, 121], and has been proposed also for class-based languages [119].

As an instance of mode-switching, consider an object representing a window with an iconized mode and an open mode. The window canvas size and the icon picture are preserved across iconization and deiconization operations. In both modes the window records the pictures drawn into it. The window reacts differently to some operations depending on the mode; for example, drawing primitives may switch between painting on-screen and off-screen. Switching between modes is obtained by reparenting; each parent encapsulates the state and behavior typical of a single mode.

²³ In contrast, Cecil is a delegation-based language that supports only static delegation. This is due at least in part to the desire to statically type that language, although some forms of dynamic inheritance are compatible with static typing.

²⁴ When the parent hierarchy is used as a hierarchical name space for storing programs, as in the Self environment [12], there is no necessary relation between an old parent and a new one.

In principle, one could achieve the same mode-switching behavior by inserting a flag in the code of each method; flipping the flag would then change the mode. However, this solution may become unwieldy when flags and modes multiply and interact. Moreover, the flag solution relies on the closed-world assumption that all present and future modes are known. With the reparenting technique, new modes can be added modularly, without modifying existing code.

Mode-switching, as used in delegation-based languages, rarely relies on the full power of dynamic reparenting. Most often it involves updating the parent links of individual leaf objects, not of shared parents. In this case, the mode-switching technique is not sensitive to the difference between delegation and embedding. In embedding-based languages, some clean and interesting forms of dynamic inheritance and mode-switching can be obtained just by attribute update.

As an example of mode-switching, we can give a new parent, *cell'* : *Cell*, to the object *reCellImp* of Section 4.5 (cf. the language Self):

```
repARENT reCellImp TO cell';
```

This means that the parent link of *reCellImp* is modified so that *cell'* replaces *cell*. A call to *reCellImp.get()* now invokes the *get* method of *cell'* instead of the one of *cell*.

An effect similar to that of mode-switching can be obtained by method update. For example, in the embedding model of Section 4.4, the reparenting of *reCellImp* to *cell'* can be written, in a somewhat clumsy but explicit form (cf. Ellie):

```
reCellImp.get :=  
  method () : Integer is return embed cell'.get() end;  
  
reCellImp.set :=  
  method (n : Integer) is  
    self.backup := self.contents;  
    embed cell'.set(n);  
  end;
```

Thus method update in an embedding model has the ability to emulate some common applications of dynamic inheritance, including ones that are based on delegation.

4.8 Traits: From Prototypes back to Classes?

Prototypes and clones were initially intended to replace classes and instances. Several prototype-based languages, however, seem to be moving towards a more traditional approach based on class-like structures. Delegation-based languages like Self and Cecil have evolved a distinction between *traits objects*, *prototype objects*, and *normal objects*.²⁵

²⁵ The terminology is from Self. Cecil has the similar notions of abstract, template, and concrete objects, respectively. Since Cecil is statically typed, the distinction is better enforced there than in Self. Omega also has a sharp distinction between prototypes and clones.

In Self and Cecil, traits and prototypes together are used to emulate classes. Traits typically contain methods. Prototypes typically contain fields and have traits for parents. Normal objects are cloned from prototypes, and thus share the traits of their prototypes (Figure 4-4). Several traits may be used by a single prototype in a form of multiple inheritance.

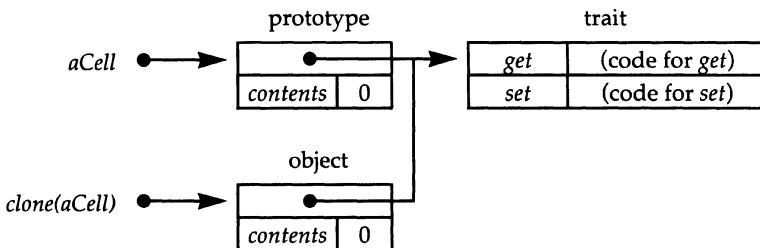


Figure 4-4. Traits.

In the spirit of classless languages, traits and prototypes are still ordinary objects. However, there is a separation of roles. Traits are intended only as the shared parents of normal objects; they should not be used directly or cloned. Prototypes are intended only as object (and prototype) generators via cloning; they should not be used directly or modified. Normal objects are intended only to be used and to carry local state; they should rely on traits for their methods. These distinctions may be purely methodological, or may be enforced; for example, some operations on traits and prototypes may be forbidden to protect these objects from accidental damage.

The separation of roles between traits and prototypes, although natural and useful in a delegation context, violates the spirit of the prototyping approach [59]. Traits normally contain only part of the intended attributes of normal objects. In particular, methods in traits refer to fields contained in prototypes. Traits, being themselves incomplete, cannot function on their own. This is a departure from the original prototyping idea that every object is a self-contained exemplar of behavior.²⁶

Therefore it is hard to regard traits as ordinary objects, even when they live in a universe where "everything is an object". Traits cannot be used as normal objects and, in typed languages, they need to be typed specially in the context of their associated prototypes. In this respect, traits have many of the properties of classes.

With the separation between traits and other objects, we seem to have come full circle back to class-based languages and to the separation between classes and instances. In fact, the traits-prototypes partition in delegation-based languages looks exactly like an implementation technique for classes. A similar traits-prototypes partition in embedding-based languages corresponds to a different implementation tech-

²⁶ Omega enforces a distinction between prototypes and clones, but remains faithful to the idea of prototypes as complete objects.

nique for classes that trades space for access speed. Cecil and Omega come even closer to class-based languages by abandoning dynamic inheritance.

Thus class-based notions and techniques are not totally banned in object-based languages. Rather, they naturally resurface as programming conventions that may be cast into language constructs. The design space between object-based and class-based languages should be seen as a continuum [114]. The achievement of object-based languages is to make clear that classes are just one of the possible ways of generating objects with common properties. Objects are more primitive than classes and should be understood and explained before classes.

4.9 Types for Object-Based Languages

Many object-based languages rely on dynamic features, such as dynamic delegation, that are hard to capture fully in a static type system. As with untyped procedural languages, this extreme plasticity has methodological drawbacks that can be remedied by the addition of appropriate type systems. A few typed prototype-based languages are now emerging.

The typings associated with prototype-based languages such as Cecil and Omega strongly restrict the dynamic features, to the point that there is not much difference between those typings and the ones used for class-based languages. Therefore, much of the discussion of types for class-based languages, carried out in Chapters 2 and 3, applies to statically typed object-based languages.

However, some peculiar typing issues remain. First, mode-switching, as opposed to unrestricted dynamic inheritance, can be typed rather easily.²⁷ Second, the treatment of method specialization and **Self** types for object-based languages requires consideration.

We have seen in Sections 2.7 and 2.8 how method specialization and covariant **Self** work in class-based language. Contravariant **Self** is more problematic; in Section 3.5 we had to recast recursive types as protocols and to abandon subsumption.

In object-based languages with dynamic inheritance, even covariant **Self** is reason for concern, because methods may be updated (e.g., for mode-switching). It may come as a surprise, then, that it is still possible to allow updating methods with result type **Self**, in the presence of subsumption. Contravariant **Self** must again be treated via protocol mechanisms. We discuss these issues extensively in Parts II and III.

²⁷ The possibility of typechecking mode-switching in object-based languages seems to be a novel idea we introduce; mode-switching is ruled out in Cecil and Omega.

5 MODELING OBJECT-ORIENTED LANGUAGES

In the previous chapters we have presented a relatively unbiased description of a large (although not exhaustive) collection of object-oriented features. We now begin to discuss our way of modeling those features.

5.1 Reduction to Basic Mechanisms

In our review of object-oriented languages we have examined differences and similarities between various object-oriented features. It should be apparent that, despite all the variations, many characteristics of class-based and object-based languages are just different presentations of a few general ideas. A natural question is whether these different presentations can be unified into a single conceptual and formal framework.

We can imagine two possible ways of encompassing the variety of object-oriented languages within a single framework. One approach is to create an object model so general that it can express directly all the conceivable variations of object classification and inheritance. Although such a model could probably be built, it would not be very useful because of either its complexity or its excessive generality. A different approach is to create a simple model that is flexible enough to represent more complex notions, but that does not directly embody any particular one. This is the approach that has resulted in a better understanding of procedural languages, via λ -calculi. This is also the approach that we adopt for object-oriented languages in this book. As we discuss later in Section 6.7 and Chapter 18, λ -calculi are not completely satisfactory as a foundation for object-oriented languages. By substituting object calculi for λ -calculi, we obtain the desired modeling power.

The object calculi that we develop are formalisms at the same level of abstraction as λ -calculi, but based exclusively on objects rather than functions. Much as in the development of λ -calculi, we start with a minimum untyped kernel and enrich it with derived constructs and with increasingly sophisticated type systems until language features can be realistically modeled. In the course of this book we investigate untyped, simply-typed, and polymorphic calculi, considering both functional and imperative execution models. A common kernel of object primitives provides conceptual unity to the whole family of calculi.

Object calculi can be seen as the result of reducing class-based and object-based languages to basic mechanisms. Class-based languages integrate many ideas into a single construct: the class. Much of the effort in understanding and analyzing class-based languages goes into distinguishing the contributions of those ideas. Object-based languages decompose class-based features in order to simplify them, and reconstruct

them in different ways. The reconstruction is not always as comfortable as the original. Still, this decomposition provides an explanation for class-based mechanisms and suggests possible variations. In our technical development we decompose object-based languages further into object calculi. The emulation of original object-based or class-based features may not be straightforward and convenient. The raw expressive power is, however, the same. By reduction to a few powerful primitives we gain a clearer semantics and we simplify the task of reasoning formally.

5.2 The Role of Method Update

In our search for minimal object calculi, it seems natural to take objects to be collections of methods. Fields are important too, but they can be seen as a derived concept; for example, a field can be viewed as a method that does not use its self parameter. The hiding of fields from public view has been advocated as a means of concealing representation choices, and thereby allowing flexibility in implementation; identifying fields with methods confers much of the same flexibility. When fields and methods are identified it is trivial to convert one into the other, conceptually turning passive data into active computation, and vice versa.

The unification of fields with methods has the advantage of simplicity: both objects and object operations assume a uniform structure. In contrast, the separation of fields from methods induces a corresponding separation of object operations, and leads to the implicit or explicit splitting of self into two components. Unifying fields with methods gives more compact and therefore more elegant calculi.

This unification, however, has one debatable consequence. The natural operation on methods is method invocation, and the natural operations on fields are field selection and field update. By unifying fields with methods, we can collapse field selection and method invocation into a single operation. To complete the unification, though, we are led to generalize field update to method update.²⁸ At the lowest level of formalization, the choice is therefore between distinguishing methods from fields or adopting method update. We choose the latter.

The reliance on method update is one of the most unusual aspects of our formal treatment, since this operation is not so common in programming languages. However, method update can be seen as a form of dynamic inheritance: it supports the dynamic modification of object behavior, allowing objects, in a sense, to change their class dynamically. Thus method update gives us an edge in modeling object-based constructs, in addition to allowing us to model the more traditional class-based constructs where fields and methods are sharply separated.

A further justification for method update can be found in the desire to tame dynamic inheritance. Dynamic inheritance is a unique feature of object-based lan-

²⁸ The language Self unifies field update with method invocation; clients perform only method invocations. But in objects there is still a distinction between so-called data slots and assignment slots.

guages, not yet found in class-based ones. Its potentially unpredictable effects have led to the search for better-behaved, restricted, dynamic inheritance mechanisms [119]. Method update is one of these better-behaved mechanisms. It is statically typable, and can be used to emulate mode-switching. With method update we avoid the amorphous and dangerous aspects of dynamic inheritance [59, 119] while maintaining its dynamic specialization aspects originally advocated by the Treaty of Orlando [114].

5.3 The Scope of this Book

In this book, we aim to cover a wide, but not universal, range of object-oriented constructions. We consider class-based and object-based languages, both functional and imperative, typed and untyped.

In previous chapters, we have identified embedding and delegation as two fundamental ideas underlying many object-oriented features. In our formalization of objects, we choose embedding over delegation as the principal object-oriented paradigm. This approach is unconventional, but we have found that it yields the simplest formal systems and, as we shall see, it allows us to represent many features.

For example, our calculi can model classes well, although they are not class-based (since classes are not built-in). They can also model embedding-based features directly, including dynamic inheritance in the form of mode-switching. As for delegation-based features, our calculi can model traits just as easily as classes, and they can model dynamic delegation based on traits. Interpreting traits as ordinary objects, though, would require significant formal complications, because of the complexity of method lookup in delegation. Therefore we cannot account easily for this particular aspect of certain delegation-based languages. At any rate, the use of traits as objects does not seem to be essential, and is gradually being abandoned, particularly in typed languages [49].

There are other aspects of object-oriented programming that we do not consider in this book. In particular, our formal treatment is restricted to sequential languages and to languages where each method is associated with a unique object (*single-dispatch*) rather than multiple objects (*multiple-dispatch*).

The exclusion of certain execution models is not very important for our purposes. In studying the typing properties of object-oriented languages, the execution model, whether functional, relational, imperative, or concurrent, is largely irrelevant. We demonstrate this by treating both functional and imperative calculi with similar type structures. There is evidence that the same type structures can also be adapted to relational languages [13] and to concurrent languages [20]. The execution models of functional and imperative languages are of course different, and we treat them separately.

Although we develop imperative calculi extensively, our principal emphasis is on functional calculi. This may seem odd, since one of the key ideas of object-oriented programming is the hiding of private state, a very non-functional notion. Moreover, in simple situations, typing for imperative models is easier, because the effects of methods need not be reflected in result types. However, every imperative language contains

a functional language, hence the typing of imperative programs is at least as hard as that of functional programs. In realistic situations, imperative models exhibit all the typing problems of functional models. Therefore it is sensible to study functional calculi; the additional type rules peculiar to imperative calculi do not cause significant complications.

The restriction to single-dispatch languages is more significant. Multiple-dispatch features have a large impact on the type structure as well as the semantics of object-oriented languages [48]. The techniques required by the two flavors of dispatch are so different that we do not attempt to treat them coherently. Currently, the large majority of object-oriented languages are based on single-dispatch. Among the typed object-oriented languages, only a few experimental ones adopt multiple-dispatch [49].

As we proceed with the formal treatment of objects in the following chapters, we return periodically to the problem of modeling various features of object-oriented languages.

PART I

Untyped and First-Order Calculi

6 UNTYPED CALCULI

In this chapter we develop an untyped calculus of objects, while keeping in mind some future typing requirements. We view objects as primitive and define a direct semantics for them. In order to illustrate the expressiveness of the calculus, we develop some small but challenging examples and show how to represent classes and inheritance.

The object calculus does not include functions, but we show how to write functions and fixpoint operators in terms of objects. Conversely, we describe attempts to encode our object primitives in untyped λ -calculi.

6.1 Object Primitives

Like the pure λ -calculus, our basic calculus of objects, the ς -calculus, consists of a minimal set of syntactic constructs and computation rules. In this section we explain its primitives informally.

6.1.1 The Primitives of the ς -Calculus

Objects are the only computational structures in our calculus. An object is simply a collection of named attributes; all attributes are methods. Each method has a bound variable that represents self and a body that produces a result. The only operations on objects are method invocation and method update. Except for variables, there is no other construct in the calculus.

The next table summarizes the notation we use for objects and methods:

Objects and methods

$\varsigma(x)b$	method with self parameter x and body b
$[l_1=\varsigma(x_1)b_1, \dots, l_n=\varsigma(x_n)b_n]$	object with n methods labeled l_1, \dots, l_n
$o.l$	invocation of method l of object o
$o.l \Leftarrow \varsigma(x)b$	update of method l of object o with method $\varsigma(x)b$

In more detail, an object $[l_1=\varsigma(x_1)b_1, \dots, l_n=\varsigma(x_n)b_n]$ is a collection of components $l_i=\varsigma(x_i)b_i$, for distinct labels l_i and associated methods $\varsigma(x_i)b_i$, for $i \in 1..n$; the order of these components does not matter. The object containing a given method is called the method's *host* object. The letter ς (sigma) is used as a binder for the self parameter of a method; $\varsigma(x)b$ is a method with self parameter x , to be bound to the host object, and body b .

A method invocation is written $o.l$, where l is a label of o . The intent is to execute the method named l of o with the object o bound to the self parameter, and to return the result of the execution.

A method update is written $o.l \leftarrow \zeta(y)b$. The semantics of update is functional: an update produces a copy of o where the method l is replaced with $\zeta(y)b$. In order to make the formal treatment easier at first, we defer investigating imperative update until Chapter 10.

For example, the following is an object that contains two methods:

$[l_1 = \zeta(x_1)[], l_2 = \zeta(x_2)x_2.l_1]$

The first method, named l_1 , returns an empty object $[]$. The second method, named l_2 , invokes the first method through the self parameter x_2 . Less trivial examples appear in Section 6.5.

In addition to methods, we often discuss fields. Fields are not part of the calculus, but a method that does not use its self variable can be regarded as a field. For emphasis, a method that uses its self variable is then called a proper method. With these conventions, method invocation doubles as field selection, and method update doubles as field update. Note that it is possible to update a field with a proper method, transparently converting passive data into active code, and conversely it is possible to update a proper method with a field.

The following table summarizes our terminology for object attributes and object operations:

Terminology

		object attributes	
object operations	selection	fields	methods
	update	field selection	method invocation
		field update	method update

Notation

- $[\dots, l=b, \dots]$ stands for $[\dots, l=\zeta(y)b, \dots]$, for an unused y . Here $l=b$ is a *field*.
- $o.l:=b$ stands for $o.l \leftarrow \zeta(y)b$, for an unused y . Here $o.l:=b$ is a *field update*.

6.1.2 The Primitive Semantics of the ζ -Calculus

The execution of a ζ -calculus term can be expressed as a sequence of reduction steps. We call this the *primitive semantics*: it represents as simply and directly as possible the intended semantics of objects.

In order to define the primitive semantics, we introduce the following notation.

Notation

- We use indexed notation of the form $\Phi_i^{i \in 1..n}$ to denote sequences Φ_1, \dots, Φ_n .
- The notation $b \rightarrow c$ means that b reduces to c in one step.

- The substitution of a term c for the free occurrences of x in b is written $b[x \leftarrow c]$. Substitution is defined precisely later on.

The reduction steps for ζ -calculus operations are as follows.

Primitive semantics

Let $o \equiv [l_i = \zeta(x_i)b_i]_{i \in 1..n}$	$(l_i \text{ distinct})$	object
$o.l_j \rightarrow b_j[x_j \leftarrow o]$	$(j \in 1..n)$	invocation reduction
$o.l_j = \zeta(y)b \rightarrow [l_j = \zeta(y)b, l_i = \zeta(x_i)b_i]_{i \in (1..n) - \{j\}}$	$(j \in 1..n)$	update reduction

A method invocation $o.l_j$ reduces to the result of the substitution of the host object for the self parameter in the body of the method named l_j . Note that the semantics of invocation is defined abstractly by substitution and not, for example, by function application.

A method update $o.l_j = \zeta(y)b$ reduces to a copy of the host object where the updated method has been replaced by the updating one.

6.1.3 Examples of Reductions

Our first example of reduction is for an object with a single method named l ; the body of this method is the empty object $[]$. Hence the method is in fact a field, and method invocation and update produce the effects expected of field selection and update. We use “ \triangleq ” for “equal by definition” and “ \equiv ” for “syntactically identical”.

$$\begin{array}{ll} \text{let } o_1 \triangleq [l = \zeta(x)[]] & \text{then } o_1.l \rightarrow [] \{x \leftarrow o_1\} \equiv [] \\ & \text{and } o_1.l = \zeta(x)o_1 \rightarrow [l = \zeta(x)o_1] \end{array}$$

With our notation for fields, we can write o_1 as $[l = []]$ and $o_1.l = \zeta(x)o_1$ as $o_1.l := o_1$.

Self-substitution is at the core of the primitive semantics of method invocation as we have defined it. It is thus easy to program non-terminating computations without explicit use of recursion:

$$\text{let } o_2 \triangleq [l = \zeta(x)x.l] \quad \text{then } o_2.l \rightarrow x.l \{x \leftarrow o_2\} \equiv o_2.l \rightarrow \dots$$

Self-substitution also allows a method to return or to modify self:

$$\begin{array}{ll} \text{let } o_3 \triangleq [l = \zeta(x)x] & \text{then } o_3.l \rightarrow x \{x \leftarrow o_3\} \equiv o_3 \\ \text{let } o_4 \triangleq [l = \zeta(y)(y.l = \zeta(x)x)] & \text{then } o_4.l \rightarrow (o_4.l = \zeta(x)x) \rightarrow o_3 \end{array}$$

We place particular emphasis on the ability to modify self, as illustrated by this last example. In class-based languages, it is common for a method to modify attributes of self, although these attributes are fields and not proper methods. As indicated in Section 5.2, we allow a method to update other methods of self, or even itself, in the spirit of prototype-based languages. This feature does not significantly complicate the problems that we address. Method update is exploited in rather interesting examples that seem difficult to emulate in other calculi and in class-based languages.

6.1.4 Other Primitives

We should stress that our choice of primitives is not without alternatives.

For example, we could have tried to provide operations to add and to remove methods from objects, from which update could be defined [43]. However, we feel that update is an important operation characterized by peculiar typing rules; we prefer to study it directly and not to explain it away at an early stage.

Similarly, the choice of objects with a fixed number of components, instead of extensible ones [1, 61, 92, 125], is a conscious one. Fixed-size objects are easier to handle, particularly in the later stages of our type-theoretical development.

Finally, we do not provide an operation to extract a method from an object as a function. As we shall see, such an operation is incompatible with object subsumption in typed calculi. Methods are inseparable from objects and cannot be recovered as functions; this consideration inspired the use of a specialized ς -notation instead of the familiar λ -notation for parameters.

6.2 The ς -Calculus

In this section we formalize the untyped object calculus just described. In later chapters, this calculus is the basis for typed calculi.

We define a formal syntax, a reduction relation with an associated equational theory, and an operational semantics. These definitions provide more precise and refined formulations of the primitive semantics of the ς -calculus. For understanding the rest of this chapter, however, it suffices to have an intuitive grasp of the primitive semantics.

6.2.1 Syntax

The following grammar describes our pure untyped object calculus. A ς -term is an expression generated by this grammar.

Syntax of the ς -calculus

$a, b ::=$	terms
x	variable
$[l_i = \varsigma(x_i)b_i]^{i \in 1..n}$	object formation (l_i distinct)
$a.l$	field selection / method invocation
$a.l = \varsigma(x)b$	field update / method update

Notation

- The notation $a, b ::= \dots$ inductively describes a set of expressions, where a and b range over the expressions being defined. The preceding definition means: a term a or b is a variable x , or an expression of the form $[l_i = \varsigma(x_i)b_i]^{i \in 1..n}$ (where the l_i are labels, the x_i are variables, and the b_i are terms), and so on. Parentheses can be added at will to group subexpressions.

- Here and in the sequel, a construct binding a variable in a scope has the form $\zeta(x)b$; other examples are $\lambda(x)b$, $\lambda(x:A)b$, and $\forall(X)B$. In syntactically ambiguous situations, the body of such a construct extends to the right as much as possible, for example up to a closing parenthesis. For example, $\lambda(x)b(x)$ means $\lambda(x)(b(x))$ and not $(\lambda(x)b)(x)$.

To complete the formal syntax of the ζ -calculus we give the definitions of free variables (FV) and substitution ($b\{x \leftarrow a\}$) for ζ -terms.

Object scoping

$FV(\zeta(y)b)$	$\triangleq FV(b) - \{y\}$
$FV(x)$	$\triangleq \{x\}$
$FV([l_i=\zeta(x_i)b_i]_{i \in 1..n})$	$\triangleq \cup_{i \in 1..n} FV(\zeta(x_i)b_i)$
$FV(a.l)$	$\triangleq FV(a)$
$FV(a.l \Leftarrow \zeta(y)b)$	$\triangleq FV(a) \cup FV(\zeta(y)b)$

Object substitution

$(\zeta(y)b)\{x \leftarrow c\}$	$\triangleq \zeta(y')(b\{y \leftarrow y'\}\{x \leftarrow c\})$	for $y' \notin FV(\zeta(y)b) \cup FV(c) \cup \{x\}$
$x\{x \leftarrow c\}$	$\triangleq c$	
$y\{x \leftarrow c\}$	$\triangleq y$	for $y \neq x$
$[l_i=\zeta(x_i)b_i]_{i \in 1..n}\{x \leftarrow c\}$	$\triangleq [l_i=(\zeta(x_i)b_i)\{x \leftarrow c\}]_{i \in 1..n}$	
$(a.l)\{x \leftarrow c\}$	$\triangleq (a\{x \leftarrow c\}).l$	
$(a.l \Leftarrow \zeta(y)b)\{x \leftarrow c\}$	$\triangleq (a\{x \leftarrow c\}).l \Leftarrow ((\zeta(y)b)\{x \leftarrow c\})$	

Notation

- A *closed term* is a term without free variables.
- We write $b\{x\}$ to highlight that x may occur free in b .
- We write $b\{c\}$, instead of $b\{x \leftarrow c\}$, when $b\{x\}$ is present in the same context.
- We identify $\zeta(x)b$ with $\zeta(y)(b\{x \leftarrow y\})$, for any y not occurring free in b . (For example, we view $\zeta(x)x$ and $\zeta(y)y$ as the same method.)
- We identify any two objects that differ only in the order of their components. (For example, we view $[l_1=\zeta(x_1)b_1, l_2=\zeta(x_2)b_2]$ and $[l_2=\zeta(x_2)b_2, l_1=\zeta(x_1)b_1]$ as the same object.)

6.2.2 Reduction

In Definition 6.2-1 we capture the primitive semantics of objects given in Section 6.1.2. This definition sets out three reduction relations: top-level one-step reduction (\rightarrow_\top), one-step reduction (\rightarrow), and general many-step reduction (\rightarrow_\gg). As is customary, we do not make error conditions explicit. We simply assume that objects and methods are

used consistently, and that otherwise computations stop without producing a result (so, for example, $[] . l$ does not reduce to a result).

Definition 6.2-1 (Reduction relations)

- (1) We write $a \rightarrow b$ if for some $o \equiv [l_i = \zeta(x_i)b_i | x_i \in \{1..n\}]$ and $j \in 1..n$, either:
 $a \equiv o . l_j$ and $b \equiv b_j[o]$, or
 $a \equiv o . l_j \not= \zeta(x)c$ and $b \equiv [l_i = \zeta(x)c, l_i = \zeta(x_i)b_i | i \in \{1..n\} - \{j\}]$.
- (2) A context $C[-]$ is a term with a single hole, and $C[d]$ represents the result of filling the hole with the term d (possibly capturing free variables of d).
We write $a \rightarrow b$ if $a \equiv C[a']$, $b \equiv C[b']$, and $a' \rightarrow b'$, where $C[-]$ is any context.
- (3) We write $\rightarrow\rightarrow$ for the reflexive and transitive closure of \rightarrow .

□

For example,

$$\begin{aligned} \text{let } & \quad o \equiv [l = \zeta(x)[[]].l \\ \text{and } & \quad C[-] \equiv [k = \zeta(x)[-] \\ \text{hence } & \quad C[o] \equiv [k = \zeta(x)o] \end{aligned}$$

then:

$$\begin{aligned} o & \rightarrow [] \\ C[o] & \rightarrow [k = \zeta(x)[[]] \\ C[o].k & \rightarrow [] \end{aligned}$$

The reduction relation satisfies a Church-Rosser property:

Theorem 6.2-2 (Church-Rosser)

If $a \rightarrow b$ and $a \rightarrow c$, then there exists d such that $b \rightarrow d$ and $c \rightarrow d$.

□

The proof of this result follows the method of Tait and Martin-Löf. The sequence of definitions and lemmas is standard (see [24], pp. 59–62).

6.2.3 Equations

We can derive an untyped equational theory from the untyped reduction rules. Our purpose in defining this theory is capturing a notion of equality for ζ -terms. We may use this notion of equality when we want to say that two objects behave in the same way. To this end, we reformulate the reduction rules as equational rules, and we add rules for symmetry, transitivity, and congruence; the congruence rules guarantee that equals can be substituted for equals.

In the following table we give the rules of the equational theory. Each rule has a number of premise judgments above a horizontal line and a single conclusion judgment below the line. Each judgment has the form $\vdash \mathcal{S}$. A premise of the form " $\vdash \mathcal{S}_i \forall i \in 1..n$ " is an abbreviation for n premises " $\vdash \mathcal{S}_1 \dots \vdash \mathcal{S}_n$ ". Instead, " $j \in 1..n$ " indicates that

there are n separate rules, one for each $j \in 1..n$. Abbreviations used within a rule are sometimes given next to the rule name.

A derivation is a tree of judgments, with leaves at the top and root at the bottom, where each judgment is obtained from the ones immediately above it by a rule. A valid judgment is one that can be obtained as the root of a derivation tree. When stating a judgment, we normally mean that it is valid.

Equational theory

(Eq Symm)	(Eq Trans)
$\vdash b \leftrightarrow a$	$\vdash a \leftrightarrow b \quad \vdash b \leftrightarrow c$
$\vdash a \leftrightarrow b$	$\vdash a \leftrightarrow c$
<hr/>	<hr/>
(Eq x)	(Eq Object) (l_i distinct)
	$\vdash b_i \leftrightarrow b'_i \quad \forall i \in 1..n$
$\vdash a \leftrightarrow x$	$\vdash [l_i = \zeta(x_i)b_i \mid_{i \in 1..n}] \leftrightarrow [l_i = \zeta(x_i)b'_i \mid_{i \in 1..n}]$
<hr/>	<hr/>
(Eq Select)	(Eq Update)
$\vdash a \leftrightarrow a'$	$\vdash a \leftrightarrow a' \quad \vdash b \leftrightarrow b'$
$\vdash a.l \leftrightarrow a'.l$	$\vdash a.l \not\models \zeta(x)b \leftrightarrow a'.l \not\models \zeta(x)b'$
<hr/>	<hr/>
(Eval Select) (where $a \equiv [l_i = \zeta(x_i)b_i \mid_{i \in 1..n}]$)	
$j \in 1..n$	
$\vdash a.l_j \leftrightarrow b_j[a]$	
<hr/>	<hr/>
(Eval Update) (where $a \equiv [l_i = \zeta(x_i)b_i \mid_{i \in 1..n}]$)	
$j \in 1..n$	
$\vdash a.l_j \not\models \zeta(x)b \leftrightarrow [l_j = \zeta(x)b, l_i = \zeta(x_i)b_i \mid_{i \in (1..n) - \{j\}}]$	

Equality (\leftrightarrow), as we have defined it, is the equivalence relation generated by one-step reduction (\rightarrow). Therefore the Church-Rosser theorem implies that if $\vdash b \leftrightarrow c$ then there exists d such that $b \rightarrow d$ and $c \rightarrow d$.

6.2.4 Operational Semantics

The reductions and equations studied so far do not impose any specific evaluation order. We now define a reduction system for the closed terms of the ζ -calculus; this reduction system is deterministic.

Our intent is to describe an evaluation strategy of the sort commonly used in programming languages. A characteristic of such evaluation strategies is that they are

weak in the sense that they do not work under binders. In our setting this means that when given an object $[l_i = \zeta(x_i) b_i]_{i \in 1..n}$ we defer reducing the body b_i until l_i is invoked.

The purpose of the reduction system is to reduce every closed expression to a *result*. A result is itself an expression; for the pure ζ -calculus, we define a result to be a term of the form $[l_i = \zeta(x_i) b_i]_{i \in 1..n}$. For example, both $[l_1 = \zeta(x)]$ and $[l_2 = \zeta(y)[l_1 = \zeta(x)]].l_1$ are results. (If we had constants such as natural numbers we would naturally include them among the results.)

Our weak reduction relation is denoted \rightsquigarrow . We write $\vdash a \rightsquigarrow v$ to mean that a reduces to a result v , or that v is the result of a . This relation is axiomatized with three rules:

Operational semantics

(Red Object) (where $v \equiv [l_i = \zeta(x_i) b_i]_{i \in 1..n}$)

$$\overline{\vdash v \rightsquigarrow v}$$

(Red Select) (where $v' \equiv [l_i = \zeta(x_i) b_i[x_i]]_{i \in 1..n}$)

$$\frac{\vdash a \rightsquigarrow v' \quad \vdash b_j[v'] \rightsquigarrow v \quad j \in 1..n}{\vdash a.l_j \rightsquigarrow v}$$

(Red Update)

$$\frac{\vdash a \rightsquigarrow [l_i = \zeta(x_i) b_i]_{i \in 1..n} \quad j \in 1..n}{\vdash a.l_j \neq \zeta(x)b \rightsquigarrow [l_j = \zeta(x)b, l_i = \zeta(x_i) b_i]_{i \in (1..n)-\{j\}}}$$

According to the first rule, results are not reduced further. According to the second rule, in order to evaluate an expression $a.l_j$ we should first calculate the result of a , check that it is in the form $[l_i = \zeta(x_i) b_i[x_i]]_{i \in 1..n}$ with $j \in 1..n$, and then evaluate $b_j[[l_i = \zeta(x_i) b_i]_{i \in 1..n}]$. According to the third rule, in order to evaluate an expression $a.l_j \neq \zeta(x)b$ we should first calculate the result of a , check that it is in the form $[l_i = \zeta(x_i) b_i]_{i \in 1..n}$ with $j \in 1..n$, and return $[l_j = \zeta(x)b, l_i = \zeta(x_i) b_i]_{i \in (1..n)-\{j\}}$. Note that we do not compute inside b or inside the b_i .

We observe that the reduction system is deterministic: if $\vdash a \rightsquigarrow v$ and $\vdash a \rightsquigarrow v'$, then $v \equiv v'$. The next proposition says that \rightsquigarrow is sound with respect to $\rightarrow\!\!\rightarrow$.

Proposition 6.2-3 (Soundness of weak reduction)

If $\vdash a \rightsquigarrow v$, then $a \rightarrow\!\!\rightarrow v$ and hence $\vdash a \leftrightarrow v$.

□

This proposition can be checked with a trivial induction on the structure of the proof that $\vdash a \rightsquigarrow v$.

Further, \rightsquigarrow is complete with respect to $\rightarrow\!\!\rightarrow$, in the following sense:

Theorem 6.2-4 (Completeness of weak reduction)

Let a be a closed term and v be a result.

If $a \rightarrow v$, then there exists v' such that $\vdash a \rightsquigarrow v'$.

□

This theorem was proved by Melliès [87] via a standardization lemma [64].

Weak reduction in the ζ -calculus is analogous to weak reduction in the λ -calculus. In the λ -calculus, weak reduction proceeds by reducing the function part of an application until it becomes an abstraction; then the argument is substituted into the abstraction either without evaluation, for call-by-name, or after evaluation, for call-by-value. In the ζ -calculus, the distinction between call-by-name and call-by-value is not so crisp. We may say that we have adopted a call-by-name strategy in light of Theorem 6.2-4, which holds for call-by-name but not call-by-value in the λ -calculus.

In later chapters we consider \rightsquigarrow again and study its properties related to typing. We also describe a relation analogous to \rightsquigarrow for an imperative calculus.

6.2.5 An Interpreter

The rules for \rightsquigarrow immediately suggest an algorithm for reduction, which constitutes an interpreter for ζ -terms. The algorithm takes a closed term and, if it converges, produces a result or the token *wrong*, which represents a computation error. We write $Outcome(c)$ for the outcome of running the algorithm on input c , assuming the algorithm terminates. The algorithm can be defined recursively as follows:

$$\begin{aligned} Outcome([l_i = \zeta(x_i)b_i]^{i \in 1..n}) &\triangleq \\ &[l_i = \zeta(x_i)b_i]^{i \in 1..n} \\ Outcome(a.l_j) &\triangleq \\ &\text{let } o = Outcome(a) \\ &\text{in} \quad \text{if } o \text{ is of the form } [l_i = \zeta(x_i)b_i\{x_i\}]^{i \in 1..n} \text{ with } j \in 1..n \\ &\quad \text{then } Outcome(b_j[o]) \\ &\quad \text{else } wrong \\ Outcome(a.l_j \notin \zeta(x)b) &\triangleq \\ &\text{let } o = Outcome(a) \\ &\text{in} \quad \text{if } o \text{ is of the form } [l_i = \zeta(x_i)b_i]^{i \in 1..n} \text{ with } j \in 1..n \\ &\quad \text{then } [l_j = \zeta(x)b, l_i = \zeta(x_i)b_i]^{i \in 1..n - \{j\}} \\ &\quad \text{else } wrong \end{aligned}$$

Clearly, $\vdash c \rightsquigarrow v$ if and only if $Outcome(c) \equiv v$ and v is not *wrong*.

6.2.6 Review of Notation

We summarize the relations between terms introduced so far. In addition, we use $=$ to denote an equality relation between terms that we do not fully formalize; the meaning of $=$ should always be clear from context.

Relations between terms

$a \equiv b$	syntactic identity (up to renaming of bound variables and reordering of object attributes)
$a \gg b$	top-level one-step reduction
$a \rightarrow b$	one-step reduction
$a \rightarrow\rightarrow b$	many-step reduction
$\vdash a \leftrightarrow b$	equality (convertibility)
$\vdash c \rightsquigarrow v$	operational reduction (with c closed and v a result)
$a = b$	equality, in an informal sense

6.3 Functions as Objects

It would be possible to incorporate the terms of the λ -calculus in our object calculus. However, having two similar variables binders (λ and ς) in a small calculus seems excessive. We could replace ς with λ , identifying methods with functions. We feel, instead, that the ς -binders have a special status. First, they can be given a simple and direct reduction semantics, instead of an indirect semantics involving both λ -abstraction and application. Second, ς -binders by themselves have a surprising expressive power; we show next that they are at least as expressive as λ -binders.

6.3.1 λ -terms as Objects

We define a translation $\langle\!\langle \cdot \rangle\!\rangle$ from λ -terms to pure objects. The λ -terms consist of variables, applications, and λ -abstractions.

Translation of the untyped λ -calculus

$$\begin{aligned} \langle\!\langle x \rangle\!\rangle &\triangleq x \\ \langle\!\langle b(a) \rangle\!\rangle &\triangleq \langle\!\langle b \rangle\!\rangle \bullet \langle\!\langle a \rangle\!\rangle \quad \text{where } p \bullet q \triangleq (p.\text{arg} := q).\text{val} \\ \langle\!\langle \lambda(x)b\{x\} \rangle\!\rangle &\triangleq \\ &[\text{arg} = \varsigma(x)x.\text{arg}, \\ &\quad \text{val} = \varsigma(x)\langle\!\langle b\{x\} \rangle\!\rangle \{x \leftarrow x.\text{arg}\}] \end{aligned}$$

Recall that $p.\text{arg} := q$ stands for $p.\text{arg} \Leftarrow \varsigma(y)q$, for an unused y .

The idea of the translation is that an application $\langle\!\langle b(a) \rangle\!\rangle$ first stores the argument $\langle\!\langle a \rangle\!\rangle$ into a known place (the field arg) inside of $\langle\!\langle b \rangle\!\rangle$, and then invokes a method of $\langle\!\langle b \rangle\!\rangle$ that can access the argument through self. For example:

$$\begin{aligned} \langle\!\langle (\lambda(x)x)(y) \rangle\!\rangle &\equiv ([\text{arg} = \varsigma(x)x.\text{arg}, \text{val} = \varsigma(x)x.\text{arg}].\text{arg} := y).\text{val} \\ &= y \end{aligned}$$

The initial value of the field *arg* is unimportant. Here, as in later examples, we use non-terminating methods to fill in a component (*arg*) that is conceptually uninitialized; other fillers could be used in an untyped calculus, but this one applies to typed calculi as well. Note that the translation maps nested λ 's to nested ζ 's: although every method has a single self parameter, we can emulate functions with multiple parameters.

Renaming (α -conversion) of λ -bound variables is valid under the translation (up to α -conversion) because of the renaming properties of ζ -binders. We can verify that β -conversion (that is, $(\lambda(x)b\{x\})(a) = b\{a\}$) is valid under the translation as well:

$$\begin{aligned} \text{let } o &\equiv [\text{arg}=\langle a \rangle, \text{val}=\zeta(x)\langle b\{x\} \rangle \{x \leftarrow x.\text{arg}\}] \\ &\equiv \langle (\lambda(x)b\{x\})(a) \rangle \\ &\equiv \langle [\text{arg}=\zeta(x)x.\text{arg}, \text{val}=\zeta(x)\langle b\{x\} \rangle \{x \leftarrow x.\text{arg}\}] . \text{arg} := \langle a \rangle \rangle . \text{val} \\ &= o.\text{val} \\ &= \langle \langle b\{x\} \rangle \{x \leftarrow x.\text{arg}\} \rangle \{x \leftarrow o\} \rangle \\ &= \langle b\{x\} \rangle \{x \leftarrow o.\text{arg}\} \\ &= \langle b\{x\} \rangle \{x \leftarrow \langle a \rangle\} \\ &= \langle b\{a\} \rangle \end{aligned}$$

However, η -conversion is not valid under the translation, basically because not every object is the translation of a λ -term. For example, the λ -terms x and $\lambda(y)x(y)$ are η -convertible, but their translations are quite different:

$$\begin{aligned} \langle x \rangle &\equiv x \\ \langle \lambda(y)x(y) \rangle &\equiv [\text{arg}=\zeta(y)y.\text{arg}, \text{val}=\zeta(y)\langle x(y) \rangle \{y \leftarrow y.\text{arg}\}] \\ &\equiv [\text{arg}=\zeta(y)y.\text{arg}, \text{val}=\zeta(y)(x.\text{arg} := y).\text{arg}].\text{val} \end{aligned}$$

The properties of the translation have been studied further by Gordon and Rees [65].

6.3.2 Default Arguments and Call-by-Keyword

The translation of the λ -calculus can be extended to provide a natural interpretation of λ -terms with default parameters and with call-by-keyword. These extensions demonstrate that the translation technique is robust.

We write $\lambda(x=c)b\{x\}$ for a function with a single parameter x with default c . We write $f(a)$ for a normal application of f to a , and $f()$ for an application of f to its default. For example, $(\lambda(x=c)x)() = c$ and $(\lambda(x=c)x)(a) = a$. The interpretation of λ -terms with a single default parameter is as follows.

Translation of default parameters

$\langle \lambda(x=c)b\{x\} \rangle \triangleq [\text{arg}=\langle c \rangle, \text{val}=\zeta(x)\langle b\{x\} \rangle \{x \leftarrow x.\text{arg}\}]$	
$\langle b(a) \rangle \triangleq \langle b \rangle \bullet \langle a \rangle$	where $p \bullet q \triangleq (p.\text{arg} := q).\text{val}$
$\langle b() \rangle \triangleq \langle b \rangle.\text{val}$	

We adapt this idea to account for call-by-keyword. We write $\lambda(x_i=c_i^{i \in 1..n})b$ for a function with multiple parameters x_i with defaults c_i . The application $f(y_i=a_i^{i \in 1..m})$ can provide fewer parameters than f expects, and in any order. The association of the provided actuals to the formals is made by the names x_i , with the defaults being used for the missing actuals. The interpretation of λ -terms with call-by-keyword and default parameters is as follows.

Translation of call-by-keyword

$\langle\langle \lambda(x_i=c_i^{i \in 1..n})b \{x_i^{i \in 1..n}\} \rangle\rangle \triangleq$	$x_i \neq val, z \notin FV(b)$
$[x_i = \langle\langle c_i \rangle\rangle_{i \in 1..n}, val = \zeta(z)(\langle\langle b \{x_i^{i \in 1..n}\} \rangle\rangle \{x_i \leftarrow z. x_i^{i \in 1..n}\})]$	
$\langle\langle b(y_i=a_i^{i \in 1..m}) \rangle\rangle \triangleq (\langle\langle b \rangle\rangle.y_1 := \langle\langle a_1 \rangle\rangle \dots y_m := \langle\langle a_m \rangle\rangle).val$	$y_i \neq val$

6.4 Fixpoints

As a consequence of the translation of λ -terms into pure objects, we obtain object-oriented versions of all the encodings that are possible within the λ -calculus. None of these encodings seem, however, particularly inspiring; more direct object-oriented encodings can usually be found. Such is the case, for example, for fixpoint operators. The following encoding is much simpler than others that can be obtained by translation of λ -terms:

$$\begin{aligned} fix &\triangleq \\ &[arg = \zeta(x)x.arg, \\ &\quad val = \zeta(x)((x.arg).arg := x.val).val] \end{aligned}$$

We can verify the fixpoint property as follows, recalling that $p \bullet q \equiv (p.arg := q).val$ is the encoding of function application:

$$\begin{aligned} fix_f &\triangleq fix.arg := f \\ &= [arg = f, val = \zeta(x)((x.arg).arg := x.val).val] \\ fix \bullet f &\equiv fix_f.val \\ &= ((fix_f.arg).arg := fix_f.val).val \\ &= (f.arg := fix \bullet f).val \\ &\equiv f \bullet (fix \bullet f) \end{aligned}$$

In particular, if we add a constant **fix** to the λ -calculus, and set $\langle\langle \text{fix} \rangle\rangle \triangleq fix$, then $\langle\langle \text{fix}(f) \rangle\rangle = \langle\langle f(\text{fix}(f)) \rangle\rangle$.

As a curiosity, observe the resemblance of **fix** to the translation of $\lambda(x)x(x)$:

$$\begin{aligned} \langle\langle \lambda(x)x(x) \rangle\rangle &\equiv \\ &[arg = \zeta(x)x.arg, \\ &\quad val = \zeta(x)((x.arg).arg := x.arg).val] \end{aligned}$$

We can also provide a translation for recursive terms, $\mu(x)b$, that is more compact than their natural definition as $\langle\!\langle \text{fix}(\lambda(x)b) \rangle\!\rangle$. We set:

$$\begin{aligned}\langle\!\langle \mu(x)b\{x\} \rangle\!\rangle &\triangleq \\ &[rec=\zeta(x)\langle\!\langle b\{x\} \rangle\!\rangle\{x\leftarrow x.\text{rec}\}].\text{rec}\end{aligned}$$

and obtain the unfolding property $\mu(x)b\{x\} = b\langle\!\langle \mu(x)b\{x\} \rangle\!\rangle$:

$$\begin{aligned}\langle\!\langle \mu(x)b\{x\} \rangle\!\rangle &\equiv [rec=\zeta(x)\langle\!\langle b\{x\} \rangle\!\rangle\{x\leftarrow x.\text{rec}\}].\text{rec} \\ &= \langle\!\langle b\{x\} \rangle\!\rangle\{x\leftarrow [rec=\zeta(x)\langle\!\langle b\{x\} \rangle\!\rangle\{x\leftarrow x.\text{rec}\}]\} \\ &\equiv \langle\!\langle b\{x\} \rangle\!\rangle\{x\leftarrow [rec=\zeta(x)\langle\!\langle b\{x\} \rangle\!\rangle\{x\leftarrow x.\text{rec}\}].\text{rec}\} \\ &\equiv \langle\!\langle b\{x\} \rangle\!\rangle\{x\leftarrow \langle\!\langle \mu(x)b\{x\} \rangle\!\rangle\} \\ &\equiv \langle\!\langle b\langle\!\langle \mu(x)b\{x\} \rangle\!\rangle \rangle\!\rangle\end{aligned}$$

The recursion operators described use only object primitives. With λ -abstraction and application, we can write the fixpoint operator of Fisher, Honsell, and Mitchell [61]:

$$\text{fix}' \triangleq \lambda(f)[rec=\zeta(s)f(s.\text{rec})].\text{rec}$$

whose translation is similar, but not identical, to our *fix*:

$$\langle\!\langle \text{fix}' \rangle\!\rangle \equiv [\text{arg}=\zeta(x)x.\text{arg}, \text{val}=\zeta(x)[rec=\zeta(s)((x.\text{arg}).\text{arg}:=s.\text{rec}).\text{val}].\text{rec}]$$

6.5 Examples

In later chapters we confront the problem of finding useful type systems for the untyped ζ -calculus. We now examine some examples that can be easily written in the untyped calculus, but pose interesting typing difficulties. Our examples involve updating through self, and some are based on updating proper methods. The typing requirements range from very basic ones, such as typing geometric points, to very sophisticated ones, such as typing an object-oriented version of the numerals inspired by Scott numerals [124]. For the impatient, typed versions of these examples can be found in Sections 9.4, 15.2, and 16.5, and imperative variants in Sections 10.4 and 17.4.

In these examples we use boolean, integer, and real constants; one of the examples illustrates how to embed basic data types within our calculus. The notation of the untyped λ -calculus is also used freely, since it can be encoded.

6.5.1 Movable Points

A classic object-oriented example involves geometric points with a move method that updates the point coordinates.

$$\begin{aligned}\text{origin}_1 &\triangleq \\ &[x = 0, \\ &\quad mv_x = \zeta(s)\lambda(dx)s.x := s.x+dx]\end{aligned}$$

$$\begin{aligned} \text{origin}_2 &\triangleq \\ &[x = 0, y = 0, \\ &\quad \text{mv_x} = \zeta(s) \lambda(dx) s.x := s.x + dx, \\ &\quad \text{mv_y} = \zeta(s) \lambda(dy) s.y := s.y + dy] \end{aligned}$$

For example, we can define $\text{unit}_2 \triangleq \text{origin}_2.\text{mv_x}(1).\text{mv_y}(1)$, and then we can compute $\text{unit}_2.x = 1$.

Intuitively, all the operations possible on origin_1 are also possible on origin_2 . Hence we would like to obtain a type system where a movable two-dimensional point, such as origin_2 , can be accepted in any context expecting a movable one-dimensional point, such as origin_1 .

6.5.2 Backup Methods

Our second example is a simple illustration of the technique of storing self. We define an object that is able to keep backup copies of itself, for example as an audit trail. This object has a *backup* method, and a *retrieve* method that returns the last backup:

$$\begin{aligned} o &\triangleq \\ &[\text{retrieve} = \zeta(s_1) s_1, \\ &\quad \text{backup} = \zeta(s_2) s_2.\text{retrieve} \neq \zeta(s_1) s_2, \\ &\quad (\text{additional attributes})] \end{aligned}$$

The initial *retrieve* method is set to return the initial object. Whenever the *backup* method is invoked it stores a copy of self into *retrieve*. Note that *backup* stores the self s_2 that is current at *backup*-invocation time, not the self s_1 that will be current at *retrieve*-invocation time. For example:

$$\begin{aligned} o' &\triangleq o.\text{backup} \\ &= [\text{retrieve} = \zeta(s_1) o, \dots] \end{aligned}$$

Later, possibly after modifying the additional attributes, we can extract the object that was most recently backed up. The *retrieve* method returns the self that was current when *backup* was last invoked, as desired:

$$o'.\text{retrieve} = o$$

We can cascade invocations of the *retrieve* method to recover older and older backups, eventually converging to the initial object.

6.5.3 Object-Oriented Natural Numbers

The technique of storing self, shown in the previous example, comes up in another interesting situation. We define the object-oriented natural numbers, that is, objects that respond to the methods *iszero* (test for zero), *pred* (predecessor), and *succ* (successor), and behave like the natural numbers.

We need to define only the numeral *zero*, because its *succ* method will generate all the other numerals. Obviously, the numeral *zero* should answer *true* to the *iszero* question, and *zero* to the *pred* question.

The *succ* method is not so easy to manufacture. When *succ* is invoked on *zero*, it should modify *zero* so that it becomes *one*. That is, *succ* should modify self so that it answers *false* to the *iszero* question, and *zero* to the *pred* question. Moreover, *succ* should be such that when invoked again on numeral *one*, it produces an appropriate numeral *two*, and so on. Hence, for any numeral, *succ* should update *iszero* to answer *false* and should update *pred* to return the self that is current when *succ* is invoked.

$$\begin{aligned} \text{zero} &\triangleq \\ &[\text{iszero} = \text{true}, \\ &\quad \text{pred} = \zeta(x) x \\ &\quad \text{succ} = \zeta(x) (x.\text{iszero} := \text{false}).\text{pred} := x] \end{aligned}$$

Here the body of *succ* consists of two cascaded updates to self. We can verify, with a few tests, that the operational semantics of natural numbers is well represented.

Instead of defining numbers with *iszero*, *pred*, and *succ*, we can define numbers with only two methods: *succ* and *case*. This gives a shorter, although more opaque, encoding of the natural numbers that does not depend on booleans:

$$\begin{aligned} \text{zero} &\triangleq \\ &[\text{case} = \lambda(z) \lambda(s) z, \\ &\quad \text{succ} = \zeta(x) x.\text{case} := \lambda(z) \lambda(s) s(x)] \end{aligned}$$

The *case* method is in fact a field containing a function of two arguments. For a numeral *n*, the first argument is returned if *n* is zero, otherwise the second argument is applied to the predecessor of *n*; that is, *n.case(a)(f)* equals *a* if *n* is zero, and equals *f(x)* if *n* is non-zero and *x* is its predecessor. The predecessor of *n* is obtained by the now familiar technique of capturing a previous self. We can compute:

$$\begin{aligned} \text{one} &\triangleq \text{zero}. \text{succ} \\ &= [\text{case} = \lambda(z) \lambda(s) s(\text{zero}), \text{succ} = (\text{unchanged})] \\ \text{two} &\triangleq \text{one}. \text{succ} \\ &= [\text{case} = \lambda(z) \lambda(s) s(\text{one}), \text{succ} = (\text{unchanged})] \end{aligned}$$

Moreover, we can recover the *iszero* and *pred* methods as functions:

$$\begin{aligned} \text{iszero} &\triangleq \lambda(n) n.\text{case}(\text{true})(\lambda(p) \text{false}) \\ \text{pred} &\triangleq \lambda(n) n.\text{case}(\text{zero})(\lambda(p) p) \end{aligned}$$

6.5.4 A Calculator

Our next example is that of a calculator object. We exploit the ability to update methods in order to record the pending arithmetic operation. When an operation *add* or *sub* is entered, the *equals* method is updated with code for addition or subtraction; this is an

example of mode-switching (see Section 4.7). Two components (*arg*, *acc*) are needed for the internal operation of the calculator, and the other four (*enter*, *add*, *sub*, *equals*) provide the user interface. A *reset* operation could be added to clear the state and restore the initial *equals* method.

```
calculator  $\triangleq$ 
  [arg = 0.0,
   acc = 0.0,
   enter =  $\zeta(s) \lambda(n) s.\text{arg} := n$ ,
   add =  $\zeta(s) (s.\text{acc} := s.\text{equals}).\text{equals} \neq \zeta(s') s'.\text{acc} + s'.\text{arg}$ ,
   sub =  $\zeta(s) (s.\text{acc} := s.\text{equals}).\text{equals} \neq \zeta(s') s'.\text{acc} - s'.\text{arg}$ ,
   equals =  $\zeta(s) s.\text{arg}$ ]
```

This definition is slightly subtle; it is meant to provide the following calculator-style behavior:

```
calculator.enter(5.0).equals = 5.0
calculator.enter(5.0).sub.enter(3.5).equals = 1.5
calculator.enter(5.0).add.add.equals = 15.0
```

6.5.5 Storage Cells

Our final example is that of storage cells, like those described in Section 2.1. Our first cell has a *contents* field, initialized to 0, and methods to get and to set the contents. We assume that clients do not update these methods.

```
myCell  $\triangleq$ 
  [contents = 0,
   get =  $\zeta(s) s.\text{contents}$ ,
   set =  $\zeta(s) \lambda(n) s.\text{contents} := n$ ]
```

A restorable cell, in addition, has a *backup* field for remembering the next-to-last value to which the cell was set, and a method to restore the contents. The *set* method is responsible for keeping the *backup* field up to date; the *restore* method simply copies the *backup* field to the *contents* field.

```
myReCell  $\triangleq$ 
  [contents = 0,
   get =  $\zeta(s) s.\text{contents}$ ,
   set =  $\zeta(s) \lambda(n) (s.\text{backup} := s.\text{contents}).\text{contents} := n$ ,
   backup = 0,
   restore =  $\zeta(s) s.\text{contents} := s.\text{backup}$ ]
```

Next we define a restorable cell that does without a *backup* field, by taking advantage of method update. The *set* method becomes responsible for keeping the *restore*

method up to date. Every time the *set* method is invoked, it updates *restore* to $\zeta(z) z.\text{contents} := m$, where m is the contents of the cell before the invocation.

```
myOtherReCell  $\triangleq$ 
  [contents = 0,
   get =  $\zeta(s) s.\text{contents}$ ,
   set =  $\zeta(s) \lambda(n) (s.\text{restore} \Leftarrow \zeta(z) z.\text{contents} := s.\text{contents}).\text{contents} := n$ ,
   restore =  $\zeta(s) s.\text{contents} := 0$ ]
```

In this example current self (s) and future self (z) are statically nested and used in the same context. Because of this, it is critical that self be a named parameter. Providing a keyword **self**, as done in many object-oriented languages, would not be sufficient.

6.6 Traits, Classes, and Inheritance

The object-oriented notions of traits, classes, and inheritance, discussed in the Review, are not explicit in our calculi. Here we discuss how these notions can be represented in terms of pure objects.

To avoid ambiguity, we call pre-methods those functions that become methods once embedded into objects. We take the point of view that inheritance means pre-method reuse, and that traits and classes are collections of interdependent reusable pre-methods. We make methods $\zeta(x)b$ reusable by writing them first as functions $\lambda(x)b$ (that is, as pre-methods), and then by repeatedly embedding these functions into objects.

6.6.1 Representing Traits and Classes

A trait is intended as a modular fragment of object behavior, with the understanding that multiple traits may be needed in generating individual objects. We represent a trait as a collection of pre-methods. A class is intended as an object generator that is self-contained, although it may have been assembled with contributions from other classes. We represent a class as a collection of pre-methods together with a method called *new* for generating new objects.

Our idea for representing traits is that if $o \equiv [l_i=\zeta(x_i)b_i \ i \in 1..n]$ is an object, then:

$$t \triangleq [l_i=\lambda(x_i)b_i \ i \in 1..n] \quad (\text{that is, } [l_i=\zeta(z)\lambda(x_i)b_i \ i \in 1..n] \text{ where } z \text{ is not used})$$

can be seen as a trait for o . The pre-methods $\lambda(x_i)b_i$ in the trait are fields that can be extracted.

The trait t is complete in the sense that we can construct o from it. We could also choose to collect the pre-methods of t into two or more smaller traits:

$$t_1 \triangleq [l_i=\lambda(x_i)b_i \ i \in 1..p] \quad t_2 \triangleq [l_i=\lambda(x_i)b_i \ i \in p+1..n]$$

From a collection of partial traits like t_1 and t_2 we can easily reconstruct t .

Given a collection of traits, it is easy to write a function that generates an object from those traits. This function applies the pre-methods of the traits to the self of the object, thereby converting the pre-methods into methods. In the case of the single, complete trait t we would write:

$$\begin{aligned} \text{new} &\triangleq \lambda(z)[l_i=\zeta(s)z.l_i(s) \ i \in 1..n] \\ o &\triangleq \text{new}(t) \\ &= [l_i=\zeta(s)(\lambda(x_i)b_i)(s) \ i \in 1..n] \\ &= [l_i=\zeta(x_i)b_i \ i \in 1..n] \end{aligned}$$

This particular implementation of new works correctly only for traits with labels $l_i \ i \in 1..n$. Therefore in order to create an object from a given trait, or collection of traits, we must pick the appropriate version of new .

We can avoid the annoyance of selecting the appropriate version of new for a trait by associating a new function with the trait. A convenient association is obtained by adding new as a proper method of the trait. This gives us a class; a class is a complete trait, plus the method new :

$$\begin{aligned} c &\triangleq \\ &[\text{new}=\zeta(z)[l_i=\zeta(s)z.l_i(s) \ i \in 1..n], \\ &\quad l_i=\lambda(s)b_i \ i \in 1..n] \\ o &\triangleq c.\text{new} \\ &= [l_i=\zeta(x_i)b_i \ i \in 1..n] \end{aligned}$$

Given any class c , the invocation $c.\text{new}$ produces an object of the right kind. The code for new is tedious to write, but can be uniformly generated.

6.6.2 Representing Inheritance

Inheritance consists of reusing pre-methods across traits and classes. For example, c' below is defined by reusing all the pre-methods of c (namely $c.l_j \ j \in 1..n$) and by adding some more pre-methods ($\lambda(s)b_k \ k \in n+1..n+m$). Informally, we say that c' is a subclass of c :

$$\begin{aligned} c' &\triangleq \\ &[\text{new}=\zeta(z)[l_i=\zeta(s)z.l_i(s) \ i \in 1..n+m], \\ &\quad l_j=c.l_j \ j \in 1..n, \\ &\quad l_k=\lambda(s)b_k \ k \in n+1..n+m] \end{aligned}$$

Inheritance for traits works similarly; we simply need not deal with new .

A subclass may override pre-methods instead of inheriting them. For example, c'' below is a subclass of c defined by reusing the first $n-1$ pre-methods of c , and overriding the last one. An overriding pre-method for l_n may still refer to the original pre-method from c as $c.l_n$, or even to some other pre-method of the superclass as $c.l_p$, for $p \in 1..n$. We can thus model the behavior of `super`, described in Section 2.3.

$$\begin{aligned} c'' &\triangleq \\ &[new = \zeta(z)[l_i = \zeta(s) z.l_i(s) \ i \in 1..n], \\ &\quad l_j = c.l_j \ j \in 1..n-1, \\ &\quad l_n = \lambda(s) \dots c.l_n(s) \dots c.l_p(s) \dots] \end{aligned}$$

Multiple inheritance is obtained by reusing pre-methods from multiple traits or classes.

6.6.3 An Example

As an illustration of the techniques introduced in this section, we revisit the example of storage cells.

In the pseudo-code of the Review, we wrote the class of cells as follows:

```
class cell is
    var contents: Integer = 0;
    method get(): Integer is return self.contents end;
    method set(n: Integer) is self.contents := n end;
end;
```

In the untyped ζ -calculus we can reformulate it as:

```
classCell  $\triangleq$ 
[new =
     $\zeta(z)[contents = \zeta(s) z.contents(s), \ get = \zeta(s) z.get(s), \ set = \zeta(s) z.set(s)],$ 
    contents =  $\lambda(s) 0,$ 
    get =  $\lambda(s) s.contents,$ 
    set =  $\lambda(s) \lambda(n) s.contents := n]$ 
```

This definition resembles the informal one. The main differences are that we have ignored the type information, that we work in a functional calculus, and that we have added the method *new*.

Similarly, we can consider the subclass *reCell*:

```
subclass reCell of cell is
    var backup: Integer = 0;
    override set(n: Integer) is
        self.backup := self.contents;
        super.set(n);
    end;
    method restore() is self.contents := self.backup end;
end;
```

We can reformulate it as:

```

classReCell  $\triangleq$ 
[new =
 $\zeta(z)[\text{contents} = \zeta(s) z.\text{contents}(s), \text{get} = \zeta(s) z.\text{get}(s), \text{set} = \zeta(s) z.\text{set}(s),$ 
 $\quad \text{backup} = \zeta(s) z.\text{backup}(s), \text{restore} = \zeta(s) z.\text{restore}(s)],$ 
 $\text{contents} = \text{classCell.contents},$ 
 $\text{get} = \text{classCell.get},$ 
 $\text{set} = \lambda(s) \lambda(n) \text{classCell.set}(s.\text{backup} := s.\text{contents})(n),$ 
 $\text{backup} = \lambda(s) 0,$ 
 $\text{restore} = \lambda(s) s.\text{contents} := s.\text{backup}]$ 

```

The attributes *contents* and *set* are explicitly inherited from *classCell*. The attribute *set* is overridden; the invocation of *classCell.set* in the new code corresponds to the use of *super* in the pseudo-notation. The attributes *backup* and *restore* are added.

6.7 Interpretations of Objects

In order to avoid premature commitments, so far we have avoided any explicit encoding of objects in terms of other notions. However, an inspection of the primitive semantics of Section 6.1 reveals close similarities between objects and records of functions. It is natural then to try to define objects in terms of records and functions. The correct definition is, however, not evident. We describe two options that have been studied in the literature, recasting them in our notation. Further options appear in Chapter 18.

6.7.1 The Self-Application Semantics

All implementations of standard (single-dispatch) object-oriented languages are based on self-application. In the self-application semantics [73, 74], methods are functions, objects are records, object selection is record selection plus self-application, and update is simply update. We write $(l_i=a_i)_{i \in 1..n}$ for the record with labels l_i and fields a_i ; we write $a \cdot l_j$ for record selection (extracting the l_j component of a), and $a \cdot l_j := b$ for functional update (producing a copy of a with the l_j component replaced by b).

Self-application semantics

Let $o \equiv [l_i=\zeta(x_i)b_i x_i]_{i \in 1..n}$	$(l_i \text{ distinct})$
o	$\triangleq (l_i=\lambda(x_i)b_i)_{i \in 1..n}$
$o \cdot l_j$	$\triangleq o \cdot l_j(o)$
$o \cdot l_j = \zeta(y)b$	$= b_j[o]$ $(j \in 1..n)$
	$= [l_j=\zeta(y)b, l_i=\zeta(x_i)b_i]_{i \in 1..n - \{j\}}$ $(j \in 1..n)$

The simple equalities shown, based on the standard β -reduction of λ -terms, reveal that the self-application semantics matches the primitive semantics. Hence untyped objects can be faithfully interpreted using λ -abstraction, application, and record constructions. Records themselves can be seen as functions over labels and thus reduced

to pure λ -terms. If we are concerned only with untyped object calculi, little else needs to be said.

Unfortunately, the match between primitive and self-application semantics does not extend directly to typed calculi. By interpreting ς -binders directly as λ -binders, the self-application semantics causes the type of each method of an object to have the form $A \rightarrow B_i$, where A is the type of the object (the self parameter), and B_i is the result type of the method. In the terminology of Section 2.6, the type A occurs in contravariant position in $A \rightarrow B_i$; because of contravariance, natural subtype relations between object types fail. We return to this point in Chapter 18, after developing the necessary formal background.

6.7.2 The Recursive-Record Semantics

In view of these difficulties with typing, alternatives to the self-application semantics have been investigated. One immediate idea is to try to “hide” the self parameters so that their types do not appear in contravariant position. This can be achieved by the use of recursive definitions, binding all the self parameters recursively to the object itself [39, 74, 106]. Note that ς is not interpreted as λ in this semantics.

Recursive-record semantics

Let $o \equiv [l_i = \varsigma(x_i) b_i[x_i] \text{ } i \in 1..n]$	$(l_i \text{ distinct})$
o	$\triangleq \mu(x)(l_i = b_i[x] \text{ } i \in 1..n)$
$o.l_j$	$\triangleq o.l_j = b_j[o] \text{ } (j \in 1..n)$

So far, the semantics of method invocation matches the primitive semantics. Moreover, the expected subtype relations are validated, because the troublesome contravariant parameters are “hidden”. If we require only an object calculus without update, then the recursive-record semantics will do. Unfortunately, neither field update nor method update work as expected in this semantics; the most plausible definition appears to be the following:

Recursive-record semantics (update)

$o.l_j = \varsigma(y) b[y]$	$\triangleq \mu(x)o.l_j := (\lambda(y)b[y])(x)$	$(j \in 1..n)$
	$= \langle l_j = b[(o.l_j = \varsigma(y) b[y]))], l_i = b_i[o] \text{ } i \in (1..n) - \{j\} \rangle$	
	$\not\equiv \mu(x)(l_j = b[x], l_i = b_i[x] \text{ } i \in (1..n) - \{j\})$	

The result of updating a method is no longer in the form of an object: record update fails to update “inside the μ ” so that all the other methods can become aware of the update. We have, for example:

$$\begin{aligned} o &\triangleq [l_1 = \varsigma(x) 3, l_2 = \varsigma(x) x.l_1] \\ p &\triangleq o.l_1 = \varsigma(x) 5 \end{aligned}$$

with the primitive semantics: $p.l_1 = 5 \quad p.l_2 = 5$

with the recursive-record semantics: $p.l_1 = 5 \quad p.l_2 = 3$

Therefore the recursive-record semantics is not adequate.

We can reason that the fixpoint operator has been used too soon, and that update has no chance of working once recursion is frozen. In contrast, recursion in the self-application semantics is open-ended: self-application occurs only at method invocation time, not at object-construction time. More sophisticated variants of the recursive-record semantics are discussed in Chapter 18.

6.7.3 Back to the Primitive Semantics

In summary, there exist some more or less faithful interpretations of objects. The best available interpretations require sophisticated typing techniques that we have not yet introduced; we resume our discussion of interpretations in Chapter 18.

In the core of this book, we take the primitive semantics as our “official” semantics of objects. As we shall see, this approach has advantages: primitive objects provide a useful abstraction. We will gain much insight by asking first what abstract properties objects should exhibit, and by investigating later how these properties can be realized.

Many programming languages take objects as primitive too, but often accompany them with additional constructs. In contrast, our calculi include a minimal number of simple primitives from which more complex constructs can be derived.

7 FIRST-ORDER CALCULI

We now begin to investigate the type theory of the ζ -calculus. We start with a simple first-order type system with object types. We prove two simple properties of this system: a unique-types theorem and a subject reduction theorem. We also define an equational theory, and extend the encodings of functions and fixpoints to typed encodings.

7.1 Formal Systems

We compose many of our typed systems from *formal system fragments*. Each fragment is named Δ_s for an appropriate subscript s . These fragments, listed in Appendix A, can be assembled to form various typed calculi, some of which are listed in Appendix B.

Each fragment consists of a set of related rules. Each rule has a number of premise judgments above a horizontal line and a single conclusion judgment below the line. Generalizing the conventions of Section 6.2.3, each judgment has the form $E \vdash \mathfrak{S}$, for a typing environment E and an assertion \mathfrak{S} whose shape depends on the judgment. For now, we can assume that an environment E consists of a list of assumptions for variables, each of the form $x:A$. The empty environment is written \emptyset . A rule has the form:

$$\frac{\begin{array}{c} (\text{Rule name}) \quad (\text{Annotations}) \\ E_1 \vdash \mathfrak{S}_1 \quad \dots \quad E_n \vdash \mathfrak{S}_n \end{array}}{E \vdash \mathfrak{S}}$$

In writing rules we use some abbreviations. A premise of the form " $E, E_i \vdash \mathfrak{S}_i \forall i \in 1..n$ " is an abbreviation for n premises " $E, E_1 \vdash \mathfrak{S}_1 \dots E, E_n \vdash \mathfrak{S}_n$ " if $n > 0$, and if $n = 0$ for " $E \vdash \diamond$ ", which means that E is well-formed. Instead, " $j \in 1..n$ " in the premise indicates that there are n separate rules, one for each j .

Each rule has a name whose first word is determined by the conclusion judgment; for example, rule names of the form "(Type ...)" are for rules whose conclusion is a type judgment. When needed, conditions restricting the applicability of a rule, as well as abbreviations used within the rule, are annotated next to the rule name.

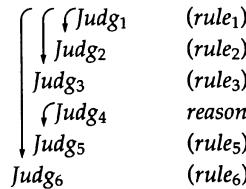
A formal system is a set of rules. A derivation in a given formal system is a tree of judgments. We use the notation:

$$\begin{array}{c} T_1 \\ \dots \\ T_n \\ \downarrow \\ \text{Judg} \end{array} \quad (rule)$$

for a derivation tree having the derivation subtrees T_1, \dots, T_n for $n \geq 0$, and having a conclusion judgment $Judg$ obtained by *(rule)* from the conclusions of T_1, \dots, T_n . To make the presentation more compact, we also use the following notation for derivation subtrees:

$Judg$	<i>reason</i>
--------	---------------

to indicate that the derivation tree leading to the conclusion $Judg$ is omitted. The reason cited for the derivation may be “by *(rule)*” indicating the last rule leading to $Judg$, or a comment like “routine” or “by previous derivation”. A derivation tree could have the shape:



where $Judg_6$ is obtained by $(rule_6)$ from $Judg_3$ and $Judg_5$, and where $Judg_1$ is obtained by $(rule_1)$ with no premises.

A valid judgment is one that can be obtained as the root of a derivation tree in a given formal system. When stating a judgment, we normally mean that it is valid. Informally, we may write \mathfrak{J} for $E \vdash \mathfrak{J}$, when the environment E is clear from context or unimportant.

7.2 The Object Fragment

We start with the formal system fragment, Δ_{Ob} , corresponding to object types. An object type $[l; B_i \ i \in 1..n]$ lists the names and types of the methods of an object. It exhibits only the result types B_i of methods: it does not explicitly list the types of ζ -bound variables. The types of all these variables are equal to the object type itself, so no information is missing. The terms of this fragment are similar to the corresponding terms of the untyped ζ -calculus, except for the addition of typing annotations for ζ -bound variables.

As guide to our first fragment, the following table lists the relevant syntax; additional constructs are necessary to obtain a full calculus (e.g., variables). This syntax is in fact implicit in the rules of Δ_{Ob} ; for later fragments we do not display the syntax explicitly.

Syntax fragment for Δ_{Ob}

$A, B ::=$		types
$[l; B_i \ i \in 1..n]$		object type (l , distinct)
...		

$a, b ::=$	terms
$[l_i = \zeta(x_i; A_i) b_i]^{i \in 1..n}$	object (l_i distinct)
$a.l$	method invocation
$a.l \Leftarrow \zeta(x; A) b$	method update
...	

Two kinds of judgments are used in the fragment Δ_{Ob} : a type judgment $E \vdash B$, stating that B is a well-formed type in the environment E , and a value typing judgment $E \vdash b : B$, stating that b has type B in E . The form of our judgments is chosen in anticipation of future uses of our fragments. For example, at the moment the well-formedness of a type B is independent of any environment E , but this will not always be true; so we prefer to write $E \vdash B$ rather than to omit E .

Δ_{Ob}

(Type Object) (l_i distinct)

$$E \vdash B_i \quad \forall i \in 1..n$$

$$\frac{}{E \vdash [l_i; B_i]^{i \in 1..n}}$$

(Val Object) (where $A \equiv [l_i; B_i]^{i \in 1..n}$)

$$E, x_i; A \vdash b_i : B_i \quad \forall i \in 1..n$$

$$\frac{}{E \vdash [l_i = \zeta(x_i; A) b_i]^{i \in 1..n} : A}$$

(Val Select)

$$E \vdash a : [l_j; B_i]^{i \in 1..n} \quad j \in 1..n$$

$$\frac{}{E \vdash a.l_j : B_j}$$

(Val Update) (where $A \equiv [l_i; B_i]^{i \in 1..n}$)

$$E \vdash a : A \quad E, x; A \vdash b : B_j \quad j \in 1..n$$

$$\frac{}{E \vdash a.l_j \Leftarrow \zeta(x; A) b : A}$$

The first rule, (Type Object) states that the object type $[l_i; B_i]^{i \in 1..n}$ is well-formed in the environment E , provided that each B_i is well-formed in E . We always assume, when writing $[l_i; B_i]^{i \in 1..n}$, that the labels l_i are distinct. We identify object types $[l_i; B_i]^{i \in 1..n}$ up to reordering of their components $l_i; B_i$.

According to the rule (Val Object), an object of type $[l_i; B_i]^{i \in 1..n}$ can be formed from a collection of n methods whose self parameters have type $[l_i; B_i]^{i \in 1..n}$ and whose bodies have types B_1, \dots, B_n . Note the circularity introduced by the self parameter: in order to construct a value of type $[l_i; B_i]^{i \in 1..n}$ we assume the existence of values of type $[l_i; B_i]^{i \in 1..n}$.

The rule (Val Select) describes how to give a type to a method invocation. When a method l_i of an object of type $[l_i; B_i]^{i \in 1..n}$ is invoked, it produces a result of type B_i .

Finally, the rule (Val Update) deals with method update. Method update preserves the type of the object that is updated. The type of the object cannot be allowed to change, because other methods assume it. The method used for the update is checked just as in the rule (Val Object).

Notation

- $[..., l=b, ...]$ stands for $[..., l=\zeta(y:A)b, ...]$, for an appropriate A and a $y \notin FV(b)$.
- $[..., l,m:B, ...]$ stands for $[..., l:B, m:B, ...]$, in examples.
- $a.l:=b$ stands for $a.l=\zeta(y:A)b$, for an appropriate A and some $y \notin FV(b)$.
- We identify $\zeta(x:A)b$ with $\zeta(y:A)(b[y \leftarrow y])$, for any $y \notin FV(b)$.
- We identify any two objects or object types that differ only in the order of their components.

The definitions of free variables and substitution are similar to the ones in Section 6.2.1. From now on we often omit these routine definitions.

7.3 Standard First-Order Fragments

We can obtain a well-rounded typed version of the ζ -calculus by adding to Δ_{Ob} two standard fragments for variables and for a constant type. We also add a fragment for typed functions, leaving their definability from typed objects for later discussions.

The fragment Δ_x describes how to build environments, and how to extract the type of a variable from an environment. The judgment $E \vdash \diamond$ asserts that E is a well-formed environment. By $x \in \text{dom}(E)$ we indicate that x is defined in E .

Δ_x

(Env \emptyset)	(Env x)	(Val x)
$\emptyset \vdash \diamond$	$E \vdash A \quad x \notin \text{dom}(E)$	$E', x:A, E'' \vdash \diamond$
	$E, x:A \vdash \diamond$	$E', x:A, E'' \vdash x : A$

The first rule, (Env \emptyset), states that the empty environment is a well-formed environment. In all our type systems, this is the only rule with no assumptions. Therefore every derivation tree must use (Env \emptyset) at its leaves.

The rule (Env x) describes how to add an assumption $x:A$ to an environment: we must check that the variable x is fresh and that A is a well-formed type.

The rule (Val x) is used to extract an assumption from an environment. The notation $E', x:A, E''$ means that $x:A$ occurs somewhere in the environment (and we know that there is a unique occurrence of x , by the assumptions of (Env x)). The conclusion is that x has type A .

The fragment Δ_K introduces a ground type K .

Δ_K

(Type Const)
$E \vdash \diamond$
$E \vdash K$

It may seem puzzling that no elements are given for the ground type K . The standard use for K is as a starting point for building function types, such as a type of Church numerals $(K \rightarrow K) \rightarrow (K \rightarrow K)$ [24]. A ground type is not strictly necessary as a starting point for building object types, because the type $[]$ is available; however, we include K in some of our object calculi in order to simplify comparisons with λ -calculi. In examples, we informally use other specific ground types, such as *Bool*, *Nat*, *Int*, and *Real*, with associated constants and operations. These could be added to our calculi with standard fragments, or through encodings.

The fragment Δ_{\rightarrow} describes functions and function types.

Δ_{\rightarrow}			
(Type Arrow)	(Val Fun)	(Val Appl)	
$E \vdash A \quad E \vdash B$	$E, x:A \vdash b : B$	$E \vdash b : A \rightarrow B \quad E \vdash a : A$	
$E \vdash A \rightarrow B$	$E \vdash \lambda(x:A)b : A \rightarrow B$	$E \vdash b(a) : B$	

By the rule (Type Arrow), a well-formed function type $A \rightarrow B$ can be obtained from any two well-formed types A and B . The rule (Val Fun) describes how to prove that a function $\lambda(x:A)b$, with parameter x of declared type A and body b , has type $A \rightarrow B$; it is sufficient to check that the body b has type B under the assumption that x has type A . The rule (Val Appl) states that a term b of function type $A \rightarrow B$ can be applied to an argument a of type A ; the result has type B .

Collecting the fragments given above, we now define three typed calculi:

$$\begin{array}{lll} \mathbf{Ob}_1 & \triangleq & \Delta_K \cup \Delta_x \cup \Delta_{\mathbf{Ob}} \\ \mathbf{F}_1 & \triangleq & \Delta_K \cup \Delta_x \cup \Delta_{\rightarrow} \\ \mathbf{FOb}_1 & \triangleq & \Delta_K \cup \Delta_x \cup \Delta_{\rightarrow} \cup \Delta_{\mathbf{Ob}} \end{array} \begin{array}{l} \text{the first-order typed } \varsigma\text{-calculus} \\ \text{the first-order typed } \lambda\text{-calculus} \\ \text{the first-order typed } \lambda\varsigma\text{-calculus} \end{array}$$

We summarize the definition of \mathbf{FOb}_1 , the largest of these calculi, by giving its syntax.

Syntax of the \mathbf{FOb}_1 calculus

$A, B ::=$	types
K	ground type
$[l_i; B_i]^{i \in 1..n}$	object type (l_i distinct)
$A \rightarrow B$	function type
$a, b ::=$	terms
x	variable
$[l_i = \varsigma(x_i; A_i); b_i]^{i \in 1..n}$	object (l_i distinct)
$a.l$	method invocation
$a.l = \varsigma(x; A)b$	method update
$\lambda(x; A)b$	function
$b(a)$	application

Unless noted otherwise, each of our calculi comes with an equational theory made up from equational fragments. For simplicity, when assembling a calculus we list only its typing fragments; the equational theory is uniquely determined, at least in this book. The equational theories for **Ob**₁, **F**₁, and **FOb**₁ are defined later in this chapter. In general, the equational fragments of each theory guarantee at least that equality is a congruence. Moreover, we adopt the convention that a calculus that includes a typing fragment Δ_s also includes the corresponding equational fragment $\Delta_{=s}$, if it exists.

7.4 Examples

The first-order calculi introduced in the previous section are too weak for most interesting examples. Here we give two simple typed examples as a drill in applying the typing rules for objects.

In these examples, and in the rest of the book, we often use the following convention for typed definitions.

Notation

- $x : A \triangleq a$ stands for $x \triangleq a$ and $E \vdash a : A$
where E is determined from the preceding context.

7.4.1 A Divergent Term

Ob₁, unlike **F**₁, is not normalizing: there exist typable terms that diverge (do not terminate). For example, the untyped term $[l = \zeta(x)x.l].l$ of Section 6.1.3 can be annotated to obtain the typed term $[l = \zeta(x:[l:[]])x.l].l$, which is typable in **Ob**₁ as follows.



Note how the rule (Val Object) enables us to assume that the self variable x has the type $[l:[]]$ when checking that the body $x.l$ of the method l has the type $[]$.

7.4.2 Booleans

Booleans and conditionals can be encoded with objects. In **Ob**₁, we do not have a single type of booleans; instead, we have a type $Bool_A$ for every type A . We define:

$$Bool_A \triangleq [if:A, then:A, else:A]$$

The terms of type $Bool_A$ can be used in conditional expressions whose result type is A . The booleans $true_A$ and $false_A$ are objects with three methods: *if*, *then*, and *else*. By indirection through *self*, $true_A.\text{if}$ returns $true_A.\text{then}$, and $false_A.\text{if}$ returns $false_A.\text{else}$. More precisely, we have:

$$\begin{aligned} true_A : Bool_A &\triangleq \\ &[if = \zeta(x:Bool_A) x.\text{then}, \text{then} = \zeta(x:Bool_A) x.\text{then}, \text{else} = \zeta(x:Bool_A) x.\text{else}] \\ false_A : Bool_A &\triangleq \\ &[if = \zeta(x:Bool_A) x.\text{else}, \text{then} = \zeta(x:Bool_A) x.\text{then}, \text{else} = \zeta(x:Bool_A) x.\text{else}] \end{aligned}$$

This typing is derivable for any given type A . For c and d of type A , we define:

$$\begin{aligned} if_A b \text{ then } c \text{ else } d : A &\triangleq \\ ((b.\text{then} \Leftarrow \zeta(x:Bool_A)c).\text{else} \Leftarrow \zeta(x:Bool_A)d).\text{if} & \quad x \notin FV(c) \cup FV(d) \end{aligned}$$

This conditional construct updates the attributes *then* and *else* of the boolean b , and invokes the *if* method. These definitions give the expected behavior of booleans:

$$\begin{aligned} if_A true_A \text{ then } c \text{ else } d &= c \\ if_A false_A \text{ then } c \text{ else } d &= d \end{aligned}$$

7.5 Some Properties of \mathbf{Ob}_1

In this section, we concentrate on \mathbf{Ob}_1 , studying its basic typing properties. Although these properties are of some interest, the main purpose of this section is to introduce ideas and techniques that are important in later chapters.

7.5.1 Unique Types

In \mathbf{Ob}_1 , every term that has a type has a unique type.

Proposition 7.5-1 (\mathbf{Ob}_1 has unique types)

If $E \vdash a : A$ and $E \vdash a : A'$ are derivable in \mathbf{Ob}_1 , then $A \equiv A'$.

□

The proof is a trivial induction on the derivation of $E \vdash a : A$, and extends to \mathbf{FOb}_1 .

Unique typing is an obvious, fundamental property for \mathbf{Ob}_1 , but small perturbations of the rules do not always preserve it. The property remains true if we omit the type annotation for update, by changing $a.l_i \Leftarrow \zeta(x:A)b$ to $a.l_i \Leftarrow \zeta(x)b$, and by adopting the rule:

$$\begin{array}{c} (\text{Val Update}') \quad (\text{where } A \equiv [l_i; B_i]^{i \in 1..n}) \\ E \vdash a : A \quad E, x:A \vdash b : B_j \quad j \in 1..n \\ \hline E \vdash a.l_i \Leftarrow \zeta(x)b : A \end{array}$$

However, the uniqueness property fails if we omit type annotations for object construction, with the rule:

(Val Object') (where $A \equiv [l_i : B_i]^{i \in 1..n}]$)

$$\frac{E, x_i : A \vdash b_i : B_i \quad \forall i \in 1..n}{E \vdash [l_i = \zeta(x_i) b_i]^{i \in 1..n} : A}$$

For example, in the modified system, $[l = \zeta(x) x.l]$ has type $[l : A]$ for any A . Still, the convention of omitting ζ -binders entirely for methods that do not depend on self is innocuous. For example, $[l=3]$ has unique type $[l : Int]$ if Int is the type of 3.

7.5.2 Subject Reduction

All the reduction relations of Section 6.2 can be extended to typed terms. Here we extend the weak reduction relation \rightsquigarrow of Section 6.2.4 to \mathbf{Ob}_1 terms. For this purpose we simply ignore and carry along any type information. Much as in Section 6.2.4, a *result* is a term of the form $[l_i = \zeta(x_i : A_i) b_i]^{i \in 1..n}]$.

Operational semantics

(Red Object) (where $v \equiv [l_i = \zeta(x_i : A_i) b_i]^{i \in 1..n}]$)

$$\overline{\vdash v \rightsquigarrow v}$$

(Red Select) (where $v' \equiv [l_i = \zeta(x_i : A_i) b_i]_{\{x_i\}}^{i \in 1..n}]$)

$$\frac{\vdash a \rightsquigarrow v' \quad \vdash b_j \{v'\} \rightsquigarrow v \quad j \in 1..n}{\vdash a.l_j \rightsquigarrow v}$$

(Red Update)

$$\frac{\vdash a \rightsquigarrow [l_i = \zeta(x_i : A_i) b_i]^{i \in 1..n} \quad j \in 1..n}{\vdash a.l_j \neq \zeta(x : A)b \rightsquigarrow [l_i = \zeta(x : A_j) b, l_i = \zeta(x_i : A_i) b_i]^{i \in (1..n) - \{j\}}}$$

In \mathbf{Ob}_1 , reduction preserves types; technically this is called a *subject reduction* property. We first state a standard substitution lemma, which can be checked by an easy induction, and then prove subject reduction; the proof is simple and serves as an introduction to similar arguments for more complex calculi.

Lemma 7.5-2 (Substitution)

If $E, x : D, E' \vdash \mathfrak{S}\{x\}$ and $E \vdash d : D$, then $E, E' \vdash \mathfrak{S}\{d\}$.

□

Theorem 7.5-3 (Subject reduction for \mathbf{Ob}_1)

Let c be a closed term and v be a result, and assume $\vdash c \rightsquigarrow v$.

If $\emptyset \vdash c : C$, then $\emptyset \vdash v : C$.

Proof

The proof is by induction on the derivation of $\vdash c \rightsquigarrow v$. There is a case for each of the rules of the operational semantics:

Case (Red Object)

This case is trivial, since $c = v$.

Case (Red Select)

Suppose $\vdash a, l_j \rightsquigarrow v$ because $\vdash a \rightsquigarrow [l_i = \zeta(x_i; A_i) b_i | x_i \in 1..n]$ and $\vdash b_j [[l_i = \zeta(x_i; A_i) b_i | x_i \in 1..n]] \rightsquigarrow v$. Assume that $\emptyset \vdash a, l_j : C$. The last step in the derivation of this judgment must be an application of (Val Select). The premise of (Val Select) must be $\emptyset \vdash a : A$ for some A of the form $[l_j; C, \dots]$. By induction hypothesis, we have $\emptyset \vdash [l_i = \zeta(x_i; A_i) b_i | x_i \in 1..n] : A$. The last step in the derivation of this judgment must be an application of (Val Object). This implies that all A_i equal A and one of the premises of (Val Object) must be $\emptyset, x_j : A \vdash b_j : C$. By Lemma 7.5-2, it follows that $\emptyset \vdash b_j [[l_i = \zeta(x_i; A_i) b_i | x_i \in 1..n]] : C$. By induction hypothesis, we obtain $\emptyset \vdash v : C$.

Case (Red Update)

Suppose $\vdash a, l_j \not\in \zeta(x; A) b \rightsquigarrow [l_j = \zeta(x; A_j) b, l_i = \zeta(x_i; A_i) b_i | i \in 1..n - \{j\}]$ because $\vdash a \rightsquigarrow [l_i = \zeta(x_i; A_i) b_i | i \in 1..n]$. Assume that $\emptyset \vdash a, l_j \not\in \zeta(x; A) b : C$. The last step in the derivation of this judgment must be an application of (Val Update). Then C equals A and the premises of (Val Update) must be $\emptyset \vdash a : A$ and $\emptyset, x: A \vdash b : B$, with A of the form $[l_j; B, \dots]$. By induction hypothesis, we have $\emptyset \vdash [l_i = \zeta(x_i; A_i) b_i | i \in 1..n] : A$. The last step in the derivation of this judgment must be an application of (Val Object). This implies that A must have the form $[l_j; B, l_i; B_i | i \in 1..n - \{j\}]$. Moreover all A_i equal A , and the premises of (Val Object) must be $\emptyset, x_i : A \vdash b_i : B_i$ for $i \in 1..n$. Therefore by an application of (Val Object) we obtain $\emptyset \vdash [l_j = \zeta(x; A) b, l_i = \zeta(x_i; A) b_i | i \in 1..n - \{j\}] : A$, that is, $\emptyset \vdash [l_j = \zeta(x; A) b, l_i = \zeta(x_i; A) b_i | i \in 1..n - \{j\}] : C$.

□

Therefore if a term has a type, and the term reduces to a result, then the result has that type too. This statement is vacuous if the term does not reduce to a result. This can happen either because reduction diverges (the rules are applicable ad infinitum), or because it gets stuck (no rule is applicable at a certain stage). We would like to prove that the latter is in fact not possible. In other words, we would like a stronger result: if the reduction does not diverge, then it produces a result of the correct type without getting stuck. The absence of stuck states is the essence of what is often called *type soundness*: well-typed programs that do not diverge produce a result of the expected type.

In the rest of this section we show that reductions never get stuck by considering an algorithm for reduction; in the algorithm, the result *wrong* represents stuck states. The algorithm is that of Section 6.2.5, extended to deal with type annotations:

$$\begin{aligned} \text{Outcome}([l_i = \zeta(x_i; A_i) b_i | i \in 1..n]) &\triangleq \\ &[l_i = \zeta(x_i; A_i) b_i | i \in 1..n] \end{aligned}$$

$\text{Outcome}(a.l_j) \triangleq$
let $o = \text{Outcome}(a)$
in if o is of the form $[l_i = \zeta(x_i:A_i)b_i \{x_i\}^{i \in 1..n}]$ with $j \in 1..n$
then $\text{Outcome}(b_j \{o\})$
else *wrong*

$\text{Outcome}(a.l_j \neq \zeta(x:A)b) \triangleq$
let $o = \text{Outcome}(a)$
in if o is of the form $[l_i = \zeta(x_i:A_i)b_i \{x_i\}^{i \in 1..n}]$ with $j \in 1..n$
then $[l_j = \zeta(x:A)b, l_i = \zeta(x_i:A_i)b_i \{x_i\}^{i \in (1..n)-\{j\}}]$
else *wrong*

If $\text{Outcome}(a)$ is defined, then it is either *wrong* or a result. We obtain:

Theorem 7.5-4 (Ob₁ reductions cannot go wrong)

If $\emptyset \vdash c : C$ and $\text{Outcome}(c)$ is defined, then $\emptyset \vdash \text{Outcome}(c) : C$, hence $\text{Outcome}(c) \neq \text{wrong}$.

Proof

The proof is by induction on the execution of $\text{Outcome}(c)$, and is very similar to the proof of Theorem 7.5-3. Since the proof is by induction on the execution of an algorithm written in an informal notation, the proof itself is somewhat informal.

Case $c \equiv [l_i = \zeta(x_i:A_i)b_i \{x_i\}^{i \in 1..n}]$

This case is trivial, since $c = \text{Outcome}(c)$.

Case $c \equiv a.l_j$

Since $\text{Outcome}(c)$ is defined, so is $\text{Outcome}(a)$. Let $o = \text{Outcome}(a)$. By hypothesis $\emptyset \vdash a.l_j : C$. Then $\emptyset \vdash a : A$ for some A of the form $[l_j : C, \dots]$. By induction hypothesis, $\emptyset \vdash o : A$. Therefore o has the form $[l_i = \zeta(x_i:A_i)b_i \{x_i\}^{i \in 1..n}]$ with $j \in 1..n$. This implies that A_j equals A and that $\emptyset, x_j:A \vdash b_j : C$. By Lemma 7.5-2, it follows that $\emptyset \vdash b_j \{o\} : C$. Since $\text{Outcome}(c)$ is defined, so is $\text{Outcome}(b_j \{o\})$. By induction hypothesis, $\emptyset \vdash \text{Outcome}(b_j \{o\}) : C$, that is, $\emptyset \vdash \text{Outcome}(c) : C$.

Case $c \equiv a.l_j \neq \zeta(x:A)b$

Since $\text{Outcome}(c)$ is defined, so is $\text{Outcome}(a)$. Let $o = \text{Outcome}(a)$. By hypothesis $\emptyset \vdash a.l_j \neq \zeta(x:A)b : C$. Then $\emptyset \vdash a : A$, and A equals C and has the form $[l_j : B, \dots]$. By induction hypothesis, $\emptyset \vdash o : A$. Therefore o has the form $[l_i = \zeta(x_i:A_i)b_i \{x_i\}^{i \in 1..n}]$ with $j \in 1..n$. This implies that A must have the form $[l_j : B, l_i : B_i \{x_i\}^{i \in (1..n)-\{j\}}]$. It follows that A_i equals A and $\emptyset, x_i:A \vdash b_i : B_i$ for all i . In addition, since $\emptyset \vdash a.l_j \neq \zeta(x:A)b : A$, we obtain also $\emptyset, x:A \vdash b : B$. Thus by (Val Object), $\emptyset \vdash [l_j = \zeta(x:A)b, l_i = \zeta(x_i:A_i)b_i \{x_i\}^{i \in (1..n)-\{j\}}] : A$, that is, $\emptyset \vdash \text{Outcome}(a.l_j \neq \zeta(x:A)b) : C$.

□

Theorem 7.5-3 can be derived from Theorem 7.5-4 using the fact that if $\vdash c \rightsquigarrow v$ then $v = \text{Outcome}(c)$. On the other hand, as the proof of Theorem 7.5-4 illustrates, it is generally routine but cumbersome to adapt a subject reduction result in order to prove the absence of stuck states. Therefore in the sequel we prove subject reduction properties but not the absence of stuck states.

7.6 First-Order Equational Theories

The equational theory of \mathbf{Ob}_1 was implicitly assumed in some of the previous discussion; we now define it. The proof of soundness for this equational theory is postponed until Chapter 14.

7.6.1 Equational Rules

We use a new judgment $E \vdash b \leftrightarrow c : A$ to assert that b and c are equivalent when considered as elements of type A . The first two rules for this new judgment express symmetry and transitivity; reflexivity will be obtained as a derived rule.

$\Delta_{=}$

$$\frac{\begin{array}{c} (\text{Eq Symm}) \\ E \vdash a \leftrightarrow b : A \end{array} \quad \begin{array}{c} (\text{Eq Trans}) \\ E \vdash a \leftrightarrow b : A \quad E \vdash b \leftrightarrow c : A \end{array}}{E \vdash b \leftrightarrow a : A \quad E \vdash a \leftrightarrow c : A}$$

There is an obvious rule for variables: a limited form of reflexivity.

$\Delta_{=x}$

$$\frac{\begin{array}{c} (\text{Eq } x) \\ E', x:A, E'' \vdash \diamond \end{array}}{E', x:A, E'' \vdash x \leftrightarrow x : A}$$

The first three rules for objects are congruence rules, establishing that two expressions are equal when all their corresponding subexpressions are equal; the next two rules are evaluation rules for selection and update, corresponding to the operational semantics of the untyped calculus of Section 6.2.4.

$\Delta_{=\mathbf{Ob}}$

$$\frac{\begin{array}{c} (\text{Eq Object}) \quad (\text{where } A \equiv [l_i; B_i]_{i \in 1..n}) \\ E, x_i:A \vdash b_i \leftrightarrow b'_i : B_i \quad \forall i \in 1..n \end{array}}{E \vdash [l_i = \zeta(x_i; A) b_i]^{i \in 1..n} \leftrightarrow [l_i = \zeta(x_i; A) b'_i]^{i \in 1..n} : A}$$

(Eq Select)

$$\frac{E \vdash a \leftrightarrow a' : [l_i : B_i]^{i \in 1..n} \quad j \in 1..n}{E \vdash a.l_j \leftrightarrow a'.l_j : B_j}$$

(Eq Update) (where $A \equiv [l_i : B_i]^{i \in 1..n}$)

$$\frac{E \vdash a \leftrightarrow a' : A \quad E, x:A \vdash b \leftrightarrow b' : B_j \quad j \in 1..n}{E \vdash a.l_j =_{\zeta(x:A)} b \leftrightarrow a'.l_j =_{\zeta(x:A)} b' : A}$$

(Eval Select) (where $A \equiv [l_i : B_i]^{i \in 1..n}$, $a \equiv [l_i =_{\zeta(x_i:A)} b_i]^{i \in 1..n}$)

$$\frac{E \vdash a : A \quad j \in 1..n}{E \vdash a.l_j \leftrightarrow b_j[a] : B_j}$$

(Eval Update) (where $A \equiv [l_i : B_i]^{i \in 1..n}$, $a \equiv [l_i =_{\zeta(x_i:A)} b_i]^{i \in 1..n}$)

$$\frac{E \vdash a : A \quad E, x:A \vdash b : B_j \quad j \in 1..n}{E \vdash a.l_j =_{\zeta(x:A)} b \leftrightarrow [l_j =_{\zeta(x:A)} b, l_i =_{\zeta(x_i:A)} b_i]^{i \in (1..n)-\{j\}} : A}$$

For functions, we have the standard theory for the first-order λ -calculus: $\Delta_{= \rightarrow}$

(Eq Fun)

$$\frac{E, x:A \vdash b \leftrightarrow b' : B}{E \vdash \lambda(x:A)b \leftrightarrow \lambda(x:A)b' : A \rightarrow B}$$

(Eq Appl)

$$\frac{E \vdash b \leftrightarrow b' : A \rightarrow B \quad E \vdash a \leftrightarrow a' : A}{E \vdash b(a) \leftrightarrow b'(a') : B}$$

(Eval Beta)

$$\frac{E \vdash \lambda(x:A)b(x) : A \rightarrow B \quad E \vdash a : A}{E \vdash (\lambda(x:A)b(x))(a) \leftrightarrow b[a] : B}$$

(Eval Eta)

$$\frac{E \vdash b : A \rightarrow B \quad x \notin \text{dom}(E)}{E \vdash \lambda(x:A)b(x) \leftrightarrow b : A \rightarrow B}$$

The fragments $\Delta_=_$ and $\Delta_{=x}$ are used in the equational theories of all our calculi. In addition, $\Delta_{=\text{Ob}}$ is used for Ob_1 , Δ_{\rightarrow} is used for F_1 , and both are used for FOb_1 .

7.6.2 Equality and self

The simple equational theory of Ob_1 is already quite interesting. In record calculi we can safely assume that two records r and r' are equal if they have the same labels l_i and if the values $r \cdot l_i$ and $r' \cdot l_i$ are equal for all $i \in 1..n$. This property does not hold for objects: two objects may give equal result for all their methods, and still be distinguishable. Consider the following definitions, where Nat is the type of natural numbers:

$$A \triangleq [x:\text{Nat}, f:\text{Nat}]$$

$$\begin{aligned} b : A &\triangleq [x=1, f=\zeta(s:A)1] \\ c : A &\triangleq [x=1, f=\zeta(s:A)s.x] \end{aligned}$$

We might think that b and c are equal at type A because $\emptyset \vdash b.x \leftrightarrow c.x : Nat$ and $\emptyset \vdash b.f \leftrightarrow c.f : Nat$. But to prove their equality, the (Eq Object) rule requires showing $\emptyset, s:A \vdash 1 \leftrightarrow s.x : Nat$. This cannot be obtained because we have no assumptions about the value of self, in particular that $s.x$ is currently 1. In fact, self may change and invalidate this assumption about its value. For example, it is possible to distinguish b from c after updating them both with equal values:

$$\begin{aligned} b' : A &\triangleq b.x:=2 \\ c' : A &\triangleq c.x:=2 \end{aligned}$$

Now we have $\emptyset \vdash b'.f \leftrightarrow 1 : Nat$ and $\emptyset \vdash c'.f \leftrightarrow 2 : Nat$, so asserting $\emptyset \vdash b \leftrightarrow c : A$ would lead to a contradiction.

This example illustrates a fundamental difference between the equational theories of object calculi and record calculi, as well as a fundamental difficulty in reasoning about objects. Still, we would like to say more about object equivalence than Δ_{Ob} allows. For example, we may wish to determine that b and c are interchangeable in contexts that read or modify only their x components; that is, in contexts where b and c are considered as having type $[x:Nat]$. This equation involves an implicit or explicit assumption that a longer object belongs to a shorter type. We examine this idea in Chapter 8.

7.7 Functions and Fixpoints

The \mathbf{Ob}_1 calculus is sufficient to encode \mathbf{F}_1 along the lines sketched in Section 6.3. Hence \mathbf{FOb}_1 has a built-in redundancy, although a very convenient one. The translation from \mathbf{F}_1 to \mathbf{Ob}_1 is shown below. Strictly speaking, this translation is defined on typing derivations, but for simplicity we write it as a translation of type-annotated λ -terms (that is, of λ -terms subscripted with their types). We write $\langle\!\langle E \rangle\!\rangle$ for the translation of an environment, $\langle\!\langle A \rangle\!\rangle$ for the translation of a type, and $\langle\!\langle a \rangle\!\rangle_\rho$ for the translation of a term. Here ρ is a function from variables to terms of \mathbf{Ob}_1 ; \emptyset is the identity function, and $\rho\{y \leftarrow a\}$ is the modification of ρ that maps y to a .

Translation of the first-order λ -calculus

$$\begin{aligned} \rho &\in Var \rightarrow \mathbf{Ob}_1\text{-term} \\ \emptyset(x) &\triangleq x \\ (\rho\{y \leftarrow a\})(x) &\triangleq \text{if } x = y \text{ then } a \text{ else } \rho(x) \\ \langle\!\langle \emptyset \rangle\!\rangle &\triangleq \emptyset \\ \langle\!\langle E, x:A \rangle\!\rangle &\triangleq \langle\!\langle E \rangle\!\rangle, x:\langle\!\langle A \rangle\!\rangle \\ \langle\!\langle K \rangle\!\rangle &\triangleq K \\ \langle\!\langle A \rightarrow B \rangle\!\rangle &\triangleq [\text{arg}:\langle\!\langle A \rangle\!\rangle, \text{val}:\langle\!\langle B \rangle\!\rangle] \end{aligned}$$

$$\begin{aligned}
 \langle\langle x_A \rangle\rangle_p &\triangleq p(x) \\
 \langle\langle b_{A \rightarrow B}(a_A) \rangle\rangle_p &\triangleq \\
 &(\langle\langle b \rangle\rangle_p, \text{arg} = \zeta(x: \langle\langle A \rightarrow B \rangle\rangle) \langle\langle a \rangle\rangle_p).val \quad \text{for } x \notin FV(\langle\langle a \rangle\rangle_p) \\
 \langle\langle \lambda(x:A)b_B \rangle\rangle_p &\triangleq \\
 &[\text{arg} = \zeta(x: \langle\langle A \rightarrow B \rangle\rangle)x.\text{arg}, \\
 &\text{val} = \zeta(x: \langle\langle A \rightarrow B \rangle\rangle)\langle\langle b \rangle\rangle_{p[x \leftarrow x.\text{arg}]}]
 \end{aligned}$$

It is not difficult to verify that the translation maps derivations in F_1 to derivations in \mathbf{Ob}_1 . Much as in Section 6.3, typed β -reduction is satisfied by this encoding, but typed η -reduction is not.

In the following chapters, we describe several extensions of F_1 and \mathbf{Ob}_1 . Consider a pair of extensions, F_e and \mathbf{Ob}_e . We say that \mathbf{Ob}_e can encode F_e when there exists a translation mapping derivations of F_e that do not use the first-order η rule into derivations of \mathbf{Ob}_e . (This η rule is (Eval Eta) of Section 7.6.1.) In this sense, \mathbf{Ob}_1 can encode F_1 .

In addition, \mathbf{Ob}_1 can encode an extension of F_1 with a fixpoint operator $\text{fix}_A: (A \rightarrow A) \rightarrow A$ for each A , in such a way that $\langle\langle \text{fix}_A(f) \rangle\rangle = \langle\langle f(\text{fix}_A(f)) \rangle\rangle$. All we need is the fixpoint operator of Section 6.4, which is typable within \mathbf{Ob}_1 :

$$\begin{aligned}
 \langle\langle \text{fix}_A \rangle\rangle_p &\triangleq \\
 &[\text{arg} = \zeta(x: \langle\langle (A \rightarrow A) \rightarrow A \rangle\rangle)x.\text{arg}, \\
 &\text{val} = \zeta(x: \langle\langle (A \rightarrow A) \rightarrow A \rangle\rangle)((x.\text{arg}).\text{arg} := x.\text{val}).\text{val}]
 \end{aligned}$$

Both $\langle\langle \text{fix}_A \rangle\rangle_p$ and the ζ -bound variables of this term have type:

$$\langle\langle (A \rightarrow A) \rightarrow A \rangle\rangle = [\text{arg}: [\text{arg}: \langle\langle A \rangle\rangle, \text{val}: \langle\langle A \rangle\rangle], \text{val}: \langle\langle A \rangle\rangle]$$

Similarly, a typed version $\mu(x:A)b$ of the recursive terms $\mu(x)b$ of Section 6.4 can be expressed in \mathbf{Ob}_1 :

$$\begin{aligned}
 \langle\langle \mu(x:A)b \rangle\rangle_p &\triangleq \\
 &[\text{rec} = \zeta(x: [\text{rec}: \langle\langle A \rangle\rangle]) \langle\langle b \rangle\rangle_{p[x \leftarrow x.\text{rec}]}.rec]
 \end{aligned}$$

The standard rules for $\mu(x:A)b$, namely:

$$\frac{\begin{array}{c} E, x:A \vdash a : A \\ \hline E \vdash \mu(x:A)a : A \end{array}}{} \quad \frac{\begin{array}{c} E, x:A \vdash a[x] : A \\ \hline E \vdash \mu(x:A)a[x] \leftrightarrow a[\mu(x:A)a] : A \end{array}}{}$$

are validated by this translation.

8 SUBTYPING

A characteristic of object-oriented languages is that an object can emulate another object that has fewer methods, since the former supports the entire protocol of the latter. Conversely, a context that expects an object with a given method protocol can be filled with an object that has an extended protocol. We call this notion *subsumption*: an object can subsume another object that has a more limited protocol.

No object calculus can fully justify its existence without some notion of subsumption. This criticism should first be directed to \mathbf{FOb}_1 , studied in Chapter 7. We should notice that, in accordance with the self-application semantics, there is no difficulty in adding to \mathbf{FOb}_1 a new operation that extracts a method from an object and returns it as a function with parameter self. Without a good reason for ruling out this extraction operation, an object calculus would be just an oddly restricted record calculus. As it happens, the idea of extracting a method from an object is uncharacteristic of object-oriented languages and, as we shall see shortly, this is precisely because this operation is fundamentally incompatible with the typing of subsumption.

In this chapter we define a particular form of subsumption that is induced by a subtype relation between object types. According to the rules for this relation, an object that belongs to a given object type also belongs to any supertype of that type, and can subsume objects in the supertype.

After extending our first-order calculi with subsumption, we prove a minimum-types theorem and a subject reduction theorem. We consider classes again in the context of the subtype relation. We also illustrate the formal differences between objects and records, and introduce variance annotations that unify objects and records.

8.1 Subtyping

We begin with the basic rules of subtyping: reflexivity, transitivity, and subsumption. It is also convenient to add a type constant, *Top*, that is a supertype of every type. The judgment $E \vdash A <: B$ asserts that *A* is a subtype of *B* in environment *E*.

$\Delta_{<:}$

(Sub Refl)	(Sub Trans)	(Val Subsumption)
$E \vdash A$	$E \vdash A <: B \quad E \vdash B <: C$	$E \vdash a : A \quad E \vdash A <: B$
$E \vdash A <: A$	$E \vdash A <: C$	$E \vdash a : B$

$$\frac{\begin{array}{c} (\text{Type Top}) \\ E \vdash \diamond \end{array}}{E \vdash \text{Top}} \quad \frac{\begin{array}{c} (\text{Sub Top}) \\ E \vdash A \end{array}}{E \vdash A <: \text{Top}}$$

After these general preliminaries, we write the subtyping rules for function and object types:

$\Delta_{<:\rightarrow}$

$$\frac{\begin{array}{c} (\text{Sub Arrow}) \\ E \vdash A' <: A \quad E \vdash B <: B' \end{array}}{E \vdash A \rightarrow B <: A' \rightarrow B'}$$

$\Delta_{<:\text{Ob}}$

$$\frac{\begin{array}{c} (\text{Sub Object}) \quad (l_i \text{ distinct}) \\ E \vdash B_i \quad \forall i \in 1..n+m \end{array}}{E \vdash [l_i:B_i]^{i \in 1..n+m} <: [l_i:B_i]^{i \in 1..n}}$$

Let us write $A\{\bullet\}$ for an incomplete type where zero or more subexpressions are missing, and represented as holes (\bullet), such as $((\bullet \rightarrow \text{Top}) \rightarrow \bullet)$. Then $A\{B\}$ is the type obtained from $A\{\bullet\}$ by filling all the holes with B . We say that $A\{\bullet\}$ is *covariant* if $B <: B'$ implies $A\{B\} <: A\{B'\}$ for all B, B' . We say that $A\{\bullet\}$ is *contravariant* if $B <: B'$ implies $A\{B'\} <: A\{B\}$ for all B, B' . Otherwise, we say that $A\{\bullet\}$ is *invariant*.

The subtyping rule (Sub Arrow) for function types is the standard one. According to this rule, a function type is contravariant in its domain and covariant in its codomain. More precisely, $\bullet \rightarrow B$ is contravariant for any type B , and $A \rightarrow \bullet$ is covariant for any type A (and $\bullet \rightarrow \bullet$ is invariant). As a consequence, $A \rightarrow B <: A' \rightarrow B$ if $A' <: A$, and $A \rightarrow B <: A \rightarrow B'$ if $B <: B'$.

The subtyping rule for object types (Sub Object) allows a longer object type $[l_i:B_i]_{i \in 1..n+m}$ to be a subtype of a shorter object type $[l_i:B_i]_{i \in 1..n}$. In general an object type has multiple incomparable supertypes, for example $[l_1:B_1, l_2:B_2]$ is a subtype of both $[l_1:B_1]$ and $[l_2:B_2]$.

Object types are invariant in their component types: the subtyping $[l_i:B_i]^{i \in 1..n+m} <: [l_i:B'_i]^{i \in 1..n}$ requires $B_i \equiv B'_i$ for all $i \in 1..n$. That is, object types are neither covariant nor contravariant; in particular, $[l:A \rightarrow \bullet]$ is not covariant and $[l:\bullet \rightarrow B]$ is not contravariant. The necessity of this invariance condition is explained in Section 8.6.

We define the calculi:

$$\begin{aligned} \mathbf{Ob}_{1<} &\triangleq \mathbf{Ob}_1 \cup \Delta_{<} \cup \Delta_{<:\text{Ob}} \\ \mathbf{F}_{1<} &\triangleq \mathbf{F}_1 \cup \Delta_{<} \cup \Delta_{<:\rightarrow} \\ \mathbf{FOb}_{1<} &\triangleq \mathbf{FOb}_1 \cup \Delta_{<} \cup \Delta_{<:\rightarrow} \cup \Delta_{<:\text{Ob}} \end{aligned}$$

The syntax of the new calculi with subtyping differs from the old calculi only by the addition of the type *Top*.

The translation of \mathbf{F}_1 into \mathbf{Ob}_1 of Section 7.7 does not extend to a corresponding translation of $\mathbf{F}_{1<}$ into $\mathbf{Ob}_{1<}$, because $\langle\!\langle A \rightarrow B \rangle\!\rangle = [\text{arg}:\langle\!\langle A \rangle\!\rangle, \text{val}:\langle\!\langle B \rangle\!\rangle]$ is invariant in $\langle\!\langle A \rangle\!\rangle$ and $\langle\!\langle B \rangle\!\rangle$. Hence $\mathbf{Ob}_{1<}$ is essentially a restricted version of $\mathbf{FOb}_{1<}$ with invariant function types. We recover the covariant/contravariant subtyping properties of function types in Section 13.2 by an encoding based on bounded universal and existential types, and in Section 8.7 by a simpler encoding based on variance annotations.

8.2 Examples

As an exercise in the use of subtyping, we treat a variant of the cell example of Section 6.5.5. (We cannot treat the cell example itself because we do not yet have recursive types.)

A *RomCell* is a storage cell that can only be read. A *PromCell* can be written once, and then becomes a *RomCell*. For the purpose of information hiding, neither *RomCell* nor *PromCell* exposes the *contents* field. A third type, *PrivateCell*, is used for operating on that field.

$$\begin{aligned}\mathit{RomCell} &\triangleq [\mathit{get}:\mathit{Nat}] \\ \mathit{PromCell} &\triangleq [\mathit{get}:\mathit{Nat}, \mathit{set}:\mathit{Nat} \rightarrow \mathit{RomCell}] \\ \mathit{PrivateCell} &\triangleq [\mathit{contents}:\mathit{Nat}, \mathit{get}:\mathit{Nat}, \mathit{set}:\mathit{Nat} \rightarrow \mathit{RomCell}]\end{aligned}$$

We obtain the subtypings $\mathit{PrivateCell} <: \mathit{PromCell} <: \mathit{RomCell}$. Thus a *PrivateCell* is a *PromCell* and a *RomCell* simply by subsumption.

We define a cell of type *PrivateCell*, and immediately subsume it into *PromCell*:

$$\begin{aligned}\mathit{myCell} : \mathit{PromCell} &\triangleq \\ &[\mathit{contents} = 0, \\ &\quad \mathit{get} = \zeta(s:\mathit{PrivateCell}) s.\mathit{contents}, \\ &\quad \mathit{set} = \zeta(s:\mathit{PrivateCell}) \lambda(n:\mathit{Nat}) s.\mathit{contents} := n]\end{aligned}$$

The subtyping $\mathit{PrivateCell} <: \mathit{RomCell}$ is needed for typing the body of the *set* method, because the body of *set* has type *PrivateCell* whereas the expected return type is *RomCell*. We can now write, for example, *myCell.set(3).get*.

8.3 Some Properties of $\mathbf{Ob}_{1<}$

In this section, we adapt the basic results about \mathbf{Ob}_1 to $\mathbf{Ob}_{1<}$.

8.3.1 Minimum Types

With the addition of subsumption we have obviously lost the unique-types property of \mathbf{Ob}_1 (see Section 7.5.1). However, a weaker property holds: every term of $\mathbf{Ob}_{1<}$ has a minimum type (if it has a type at all). This property also holds for $\mathbf{F}_{1<}$, and we believe

that it holds for $\mathbf{FOb}_{1<}$. The minimum-types property is potentially useful for developing typechecking algorithms, since it guarantees the existence of a canonical type for each typable term.

In order to prove the minimum-types property for $\mathbf{Ob}_{1<}$, we consider a system $\mathbf{MinOb}_{1<}$ obtained from $\mathbf{Ob}_{1<}$ by removing (Val Subsumption), and by modifying the (Val Object) and (Val Update) rules as follows:

Modified rules for $\mathbf{MinOb}_{1<}$:

(Val Min Object) (where $A \equiv [l_i : B_i]_{i \in 1..n}]$)
$E, x_i : A \vdash b_i : B_i' \quad \emptyset \vdash B_i' <: B_i \quad \forall i \in 1..n$
$E \vdash [l_i = \zeta(x_i : A) b_i]_{i \in 1..n} : A$
(Val Min Update) (where $A \equiv [l_i : B_i]_{i \in 1..n}]$)
$E \vdash a : A' \quad \emptyset \vdash A' <: A \quad E, x : A \vdash b : B_j' \quad \emptyset \vdash B_j' <: B_j \quad j \in 1..n$
$E \vdash a.l_j = \zeta(x : A) b : A$

Typing in $\mathbf{MinOb}_{1<}$ is unique, as we show below. We can easily extract from $\mathbf{MinOb}_{1<}$ a typechecking algorithm that, given any environment E and term a , computes the type A such that $E \vdash a : A$ if one exists.

The next three propositions are proved by easy inductions on the derivations of $E \vdash a : A$ in $\mathbf{MinOb}_{1<}$.

Proposition 8.3-1 ($\mathbf{MinOb}_{1<}$ typings are $\mathbf{Ob}_{1<}$ typings)

If $E \vdash a : A$ is derivable in $\mathbf{MinOb}_{1<}$, then it is also derivable in $\mathbf{Ob}_{1<}$.

□

Proposition 8.3-2 ($\mathbf{MinOb}_{1<}$ has unique types)

If $E \vdash a : A$ and $E \vdash a : A'$ are derivable in $\mathbf{MinOb}_{1<}$, then $A \equiv A'$.

□

Proposition 8.3-3 ($\mathbf{MinOb}_{1<}$ has smaller types than $\mathbf{Ob}_{1<}$)

If $E \vdash a : A$ is derivable in $\mathbf{Ob}_{1<}$, then $E \vdash a : A'$ is derivable in $\mathbf{MinOb}_{1<}$ for some A' such that $E \vdash A' <: A$ is derivable (in either system).

□

We obtain:

Proposition 8.3-4 ($\mathbf{Ob}_{1<}$ has minimum types)

In $\mathbf{Ob}_{1<}$, if $E \vdash a : A$ then there exists B such that $E \vdash a : B$ and, for any A' , if $E \vdash a : A'$ then $E \vdash B <: A'$.

Proof

Assume $E \vdash a : A$. By Proposition 8.3-3, $E \vdash a : B$ is derivable in $\text{MinOb}_{1<}$ for some B such that $E \vdash B <: A$. By Proposition 8.3-1, $E \vdash a : B$ is also derivable in $\text{Ob}_{1<}$. By Proposition 8.3-3, if $E \vdash a : A'$, then $E \vdash a : B'$ is also derivable in $\text{MinOb}_{1<}$ for some B' such that $E \vdash B' <: A'$. By Proposition 8.3-2, $B \equiv B'$, so $E \vdash B <: A'$.

□

Just as lack of annotations for ζ -binders destroys the unique-types property for Ob_1 , it destroys the minimum-types property for $\text{Ob}_{1<}$. For example, let:

$$\begin{aligned} A &\equiv [l:[]] \\ A' &\equiv [l:A] \\ a &\equiv [l=\zeta(x)[l=\zeta(x)[]]] \end{aligned}$$

then:

$$\emptyset \vdash a : A \quad \text{and} \quad \emptyset \vdash a : A'$$

but A and A' have no common subtype. This example also shows that minimum typing is lost for objects with fields (where the ζ -binders are omitted entirely) because a can be written as $[l=[l=[]]]$ with our conventions.

The term $a.l:=[]$ typechecks using $\emptyset \vdash a : A$ but not using $\emptyset \vdash a : A'$. Naive type inference algorithms might find the type A' for a , and fail to find any type for $a.l:=[]$. Thus the absence of minimum typings poses practical problems for type inference. Palsberg has described an ingenious algorithm for type inference that surmounts these problems [98].

In contrast, in $\text{Ob}_{1<}$ (with annotations), both $\emptyset \vdash [l=\zeta(x:A)[l=\zeta(x:A)[]]] : A$ and $\emptyset \vdash [l=\zeta(x:A')[l=\zeta(x:A)[]]] : A'$ are minimum typings. The former typing can be used to construct a typing for $a.l:=[]$, by deriving $\emptyset \vdash [l=\zeta(x:A)[l=\zeta(x:A)[]]]=\zeta(x:A)[] : A$.

The term $a.l:=[]$ is also an example of a term typable in $\text{Ob}_{1<}$ but not in Ob_1 .

8.3.2 Subject Reduction

As in Ob_1 (see Section 7.5.2), typing is consistent with reduction in $\text{Ob}_{1<}$. We start the proof of consistency by stating two standard lemmas.

Lemma 8.3-5 (Bound weakening)

If $E, x:D, E' \vdash \mathfrak{S}$ and $E \vdash D' <: D$, then $E, x:D', E' \vdash \mathfrak{S}$.

□

Lemma 8.3-6 (Substitution)

If $E, x:D, E' \vdash \mathfrak{S}\{x\}$ and $E \vdash d : D$, then $E, E' \vdash \mathfrak{S}\{d\}$.

□

Using these lemmas, we obtain a subject reduction theorem:

Theorem 8.3-7 (Subject reduction for $\text{Ob}_{1<:}$)

Let c be a closed term and v be a result, and assume $\vdash c \rightsquigarrow v$.
If $\emptyset \vdash c : C$, then $\emptyset \vdash v : C$.

Proof

The proof is by induction on the derivation of $\vdash c \rightsquigarrow v$.

Case (Red Object)

This case is trivial, since $c = v$.

Case (Red Select)

Suppose $\vdash a.l_j \rightsquigarrow v$ because $\vdash a \rightsquigarrow [l_i = \zeta(x_i:A_i)b_i\{x_i\}^{i \in 1..n}]$ and $\vdash b_j \{ [l_i = \zeta(x_i:A_i)b_i\{x_i\}^{i \in 1..n}] \} \rightsquigarrow v$. Assume that $\emptyset \vdash a.l_j : C$. This must have come from an application of (Val Select) with assumption $\emptyset \vdash a : A$ where A has the form $[l_j : B_j, \dots]$, and with conclusion $\emptyset \vdash a.l_j : B_j$, followed by a number of subsumption steps implying $\emptyset \vdash B_j <: C$ by transitivity. By induction hypothesis, we have $\emptyset \vdash [l_i = \zeta(x_i:A_i)b_i\{x_i\}^{i \in 1..n}] : A$. This implies that there exists A' such that $\emptyset \vdash A' <: A$, that all A_i equal A' , that $\emptyset \vdash [l_i = \zeta(x_i:A')b_i\{x_i\}^{i \in 1..n}] : A'$, and that $\emptyset, x_j:A' \vdash b_j : B_j$. By Lemma 8.3-6, it follows that $\emptyset \vdash b_j \{ [l_i = \zeta(x_i:A')b_i\{x_i\}^{i \in 1..n}] \} : B_j$. By induction hypothesis, we obtain $\emptyset \vdash v : B_j$ and, by subsumption, $\emptyset \vdash v : C$.

Case (Red Update)

Suppose $\vdash a.l_j \not\in \zeta(x:A)b \rightsquigarrow [l_j = \zeta(x:A)b, l_i = \zeta(x_i:A_i)b_i^{i \in 1..n - |j|}]$ because $\vdash a \rightsquigarrow [l_i = \zeta(x_i:A_i)b_i^{i \in 1..n}]$. Assume that $\emptyset \vdash a.l_j \not\in \zeta(x:A)b : C$. This must have come from an application of (Val Update) with assumptions $\emptyset \vdash a : A$ and $\emptyset, x:A \vdash b : B$ where A has the form $[l_j : B_j, \dots]$, and with conclusion $\emptyset \vdash a.l_j \not\in \zeta(x:A)b : A$, followed by a number of subsumption steps implying $\emptyset \vdash A <: C$ by transitivity. By induction hypothesis, we have $\emptyset \vdash [l_i = \zeta(x_i:A_i)b_i^{i \in 1..n}] : A$. This implies that A_j has the form $[l_j : B, l_i : B_i^{i \in (1..n)-|j|}]$, that $\emptyset \vdash A_j <: A$, that A_i equals A_j , and that $\emptyset, x_i:A_j \vdash b_i : B_i$ for all i . By Lemma 8.3-5, it follows that $\emptyset, x:A_j \vdash b : B$. Therefore by (Val Object), $\emptyset \vdash [l_j = \zeta(x:A)b, l_i = \zeta(x_i:A_j)b_i^{i \in (1..n)-|j|}] : A_j$. We obtain $\emptyset \vdash [l_j = \zeta(x:A)b, l_i = \zeta(x_i:A_j)b_i^{i \in (1..n)-|j|}] : C$ by subsumption.

□

As in Ob_1 , the proof of subject reduction is simply a sanity check. It remains an easy proof, with just one subtle point: notice that the proof would have failed if we had defined (Red Update) so that $\vdash a.l_j \not\in \zeta(x:A)b \rightsquigarrow [l_j = \zeta(x:A)b, l_i = \zeta(x_i:A_i)b_i^{i \in (1..n)-|j|}]$ with an A instead of an A_j in the bound for x .

8.4 First-Order Equational Theories with Subtyping

We extend the equational theories of Section 7.6.1 to take subsumption into account. Although subsumption is easy to incorporate, and the equational theory becomes stronger, we still find some interesting difficulties in reasoning about self.

8.4.1 Equational Rules with Subtyping

With subtyping, two terms may be equal at some types but not at others. Therefore the type information in the equality judgment $E \vdash a \leftrightarrow a' : A$ is essential.

The following equalities are associated with the Δ_{\leq} fragment:

Δ_{\leq}

$$\frac{\begin{array}{c} (\text{Eq Subsumption}) \\ E \vdash a \leftrightarrow a' : A \quad E \vdash A \leq B \end{array}}{E \vdash a \leftrightarrow a' : B} \quad \frac{\begin{array}{c} (\text{Eq Top}) \\ E \vdash a : A \quad E \vdash b : B \end{array}}{E \vdash a \leftrightarrow b : \text{Top}}$$

Still, this does not give us enough power to compare objects of different lengths (except as members of Top). We remedy this by augmenting the $\Delta_{\leq \text{Ob}}$ fragment from Section 7.6.1 with the rule (Eq Sub Object), which allows us to ignore methods that do not appear in the type of an object. At the same time we generalize (Eval Select) and (Eval Update) to deal with objects and types of different lengths.

$\Delta_{\leq \text{Ob}}$

$$\frac{\begin{array}{c} (\text{Eq Sub Object}) \quad (\text{where } A \equiv [l_i : B_i]_{i \in 1..n}, A' \equiv [l_i : B_i]_{i \in 1..n+m}) \\ E, x_i : A \vdash b_i : B_i \quad \forall i \in 1..n \quad E, x_j : A' \vdash b_j : B_j \quad \forall j \in n+1..n+m \end{array}}{E \vdash [l_i = \zeta(x_i : A) b_i]_{i \in 1..n} \leftrightarrow [l_i = \zeta(x_i : A') b_i]_{i \in 1..n+m} : A}$$

$$\frac{\begin{array}{c} (\text{Eval Select}) \quad (\text{where } A \equiv [l_i : B_i]_{i \in 1..n}, a \equiv [l_i = \zeta(x_i : A') b_i]_{i \in 1..n+m}) \\ E \vdash a : A \quad j \in 1..n \end{array}}{E \vdash a.l_j \leftrightarrow b_j \llbracket a \rrbracket : B_j}$$

$$\frac{\begin{array}{c} (\text{Eval Update}) \quad (\text{where } A \equiv [l_i : B_i]_{i \in 1..n}, a \equiv [l_i = \zeta(x_i : A') b_i]_{i \in 1..n+m}) \\ E \vdash a : A \quad E, x : A \vdash b : B_j \quad j \in 1..n \end{array}}{E \vdash a.l_j \llbracket \zeta(x : A) b \rrbracket \leftrightarrow [l_j = \zeta(x : A') b, l_i = \zeta(x_i : A') b_i]_{i \in (1..n+m) - \{j\}} : A}$$

According to (Eq Sub Object) an object can be truncated to its externally visible collection of methods, but only if those methods do not depend on the hidden ones. (The truncated object would not work otherwise.) It is hard to obtain more than this, as we discuss in the next section.

The fragment Δ_{\leq} is used in the equational theories of all our calculi with subtyping. In addition, $\Delta_{\leq \text{Ob}}$ is used for Ob_{\leq} and FOb_{\leq} .

8.4.2 Equality, Subtyping, and self

In Section 7.6.2 we saw that the following terms b and c cannot be equal at type A :

$$A \triangleq [x : \text{Nat}, f : \text{Nat}]$$

$$\begin{aligned} b : A &\triangleq [x=1, f=\zeta(s:A)1] \\ c : A &\triangleq [x=1, f=\zeta(s:A)s.x] \end{aligned}$$

Using (Eq Sub Object) it is at least possible to show that b and c are equal at type $[x:Nat]$, by showing $\emptyset \vdash b \leftrightarrow [x=1] : [x:Nat]$ and $\emptyset \vdash c \leftrightarrow [x=1] : [x:Nat]$.

We can ask next whether b and c are equal at type $[f:Nat]$. This seems reasonable because the only way to distinguish b and c is to update their x components, which are not exposed in $[f:Nat]$. However, we cannot apply (Eq Object) and (Eq Sub Object) because we would need to show $\emptyset, s:[f:Nat] \vdash 1 \leftrightarrow s.x : Nat$.

A stronger rule would be required to show $\emptyset \vdash b \leftrightarrow c : [f:Nat]$, but it is not clear how to write such a rule satisfactorily. Fortunately, equations such as $\emptyset \vdash b \leftrightarrow c : [f:Nat]$ can be handled by more sophisticated proof techniques based on bisimilarity [65].

8.5 Classes and Inheritance

We extend the untyped representation of classes discussed in Section 6.6 to account for typing and subtyping. As in Section 6.6 we represent classes as collections of pre-methods with a *new* method. In $\text{FOb}_{1<}$, these classes acquire natural types; moreover, inheritance is in harmony with subtyping. After discussing the typing of classes, we give an extensive example.

8.5.1 Representing Class Types and Inheritance

If $A \equiv [l_i:B_i]_{i \in 1..n}$ is an object type, then:

$$\text{Class}(A) \triangleq [\text{new}:A, l_i:A \rightarrow B_i]_{i \in 1..n}$$

is the type of classes generating objects of type A . These classes have the form:

$$\begin{aligned} &[\text{new}=\zeta(z:\text{Class}(A))[l_i=\zeta(s:A)z.l_i(s)]_{i \in 1..n}], \\ &l_i=\lambda(s:A)b_i]_{i \in 1..n} \end{aligned}$$

We would like to say that a class of type $\text{Class}(A')$ with more pre-methods inherits (or may inherit) from a class of type $\text{Class}(A)$ with fewer pre-methods. It can be seen easily that inheritance is not simply related to subtyping of class types: $A' <: A$ implies neither $\text{Class}(A') <: \text{Class}(A)$ nor $\text{Class}(A) <: \text{Class}(A')$, since A and A' occur invariantly in $\text{Class}(A)$ and $\text{Class}(A')$. Therefore we define an ad hoc inheritance relation on class types that captures the intuition of method reuse. When A and A' are object types, we set:

$$\text{Class}(A') \text{ may inherit from } \text{Class}(A) \text{ iff } A' <: A$$

If $\text{Class}(A')$ may inherit from $\text{Class}(A)$, then A' and A have the forms $A' \equiv [l_i:B_i]_{i \in 1..n+m}$ and $A \equiv [l_i:B_i]_{i \in 1..n}$. Thus $A \rightarrow B_i <: A' \rightarrow B_i$ for $i \in 1..n$, because of the contravariance of function types. Hence, pre-methods of a class $c : \text{Class}(A)$, having type $A \rightarrow B_i$, may be reused in assembling another class $c' : \text{Class}(A')$, by subsumption.

Whenever c' is defined by reassembling, extending, and modifying c we may informally say that c' inherits from c . For example, c' may replace the pre-method l_1 of c , inherit all the other pre-methods, and add some new ones:

$$\begin{aligned} c : \text{Class}(A) &\triangleq \\ &[new = \zeta(z:\text{Class}(A))[l_i = \zeta(s:A)z.l_i(s)^{i \in 1..n}], \\ &l_i = \lambda(s:A)b_i^{i \in 1..n}] \\ c' : \text{Class}(A') &\triangleq \\ &[new = \zeta(z:\text{Class}(A'))[l_i = \zeta(s:A')z.l_i(s)^{i \in 1..n+m}], \\ &l_1 = \lambda(s:A')b_1, \\ &l_j = c.l_j^{j \in 2..n}, \\ &l_k = \lambda(s:A')b_k^{k \in n+1..n+m}] \end{aligned}$$

The subtyping $A' \ll A$ is needed for typing the components l_j of c' , which contain inherited pre-methods.

This technique generalizes straightforwardly to multiple inheritance. A class c' may reuse pre-methods from several other classes, say c_1 and c_2 of types $\text{Class}(A_1)$ and $\text{Class}(A_2)$, respectively. These classes c_1 and c_2 need not even have the same type: if $A' \ll A_1$ and $A' \ll A_2$, then we have that $\text{Class}(A')$ may inherit from both $\text{Class}(A_1)$ and $\text{Class}(A_2)$.

In Section 6.6 we described an interpretation of **super**; that interpretation is well-typed according to the class types just described. For example, the overriding pre-method l_1 of the subclass c' may contain an invocation to the original pre-method l_1 of the superclass c :

$$l_1 = \lambda(s:A') \dots c.l_1(s) \dots$$

The application $c.l_1(s)$ is our interpretation for **super**. l_1 . Since $c.l_1$ has type $A \rightarrow B_1$, and $A' \ll A$, the application $c.l_1(s)$ is well-typed.

8.5.2 Variations on Class Types

In Section 6.6 we introduced traits as possibly incomplete collections of pre-methods. If $A \equiv [l_i:B_i^{i \in 1..n}]$ is an object type, then we say that:

$$[l_i:A \rightarrow B_i^{i \in 1..m}] \quad \text{for } m \leq n$$

is a trait type for it. An element of such a type is a trait.

Even when $m < n$, the pre-methods for $l_i^{i \in 1..m}$ may still refer in a well-typed way to the pre-methods for $l_i^{i \in m+1..n}$ that are not provided. For example, in the trait $[l_1 = \lambda(s:A) \dots s.l_n \dots]$, the invocation $s.l_n$ is well-typed. Thus a trait may implicitly refer to its completion, and may be combined with other traits to form a complete collection of pre-methods.

A trait, especially when incomplete, is also called an *abstract class* [115]. The term “abstract” is used in the sense that concrete objects cannot be constructed from an abstract class, because not all the necessary pre-methods may be available. Abstract

classes, therefore, can be used for inheritance but cannot be instantiated. In the literature one can also find *concrete classes* that can be both instantiated and used for inheritance (like our normal classes), and *leaf classes* that can be instantiated but not used for inheritance [116]. An element of type $[new:A]$ can be seen as a leaf class: it can be instantiated, via *new*, but cannot be used for inheritance because its pre-methods have been forgotten.

Generalizing from these considerations, we may associate two separate interfaces with a class type. The first interface is the collection of methods that are available on instances. The second interface is the collection of methods that can be inherited in subclasses. We generalize our definition of class types to account for these additional parameters.

For an object type $A \equiv [l_i:B_i]_{i \in I}$ with methods $l_i : B_i$ (where I is an index set) we consider a restricted *instance interface*, determined by a set $Ins \subseteq I$, and a restricted *subclass interface*, determined by a set $Sub \subseteq I$. Alternatively, we may think that each method in a class definition has two annotations: whether the method can be inherited in subclasses, and whether the method can be used on instances. These annotations determine the sets Ins and Sub .

Therefore if $A \equiv [l_i:B_i]_{i \in I}$ is an object type, we define:

$$\text{Class}(A)_{Ins,Sub} \triangleq [new:[l_i:B_i]_{i \in Ins}, l_i:A \rightarrow B_i]_{i \in Sub} \quad \text{for } Ins, Sub \subseteq I$$

$$\text{Class}(A) \triangleq \text{Class}(A)_{I,I}$$

At the term level we define:

$$\text{class}(A, a_i)_{Ins} \triangleq [new = \zeta(z:\text{Class}(A)_{Ins,I})[l_i = \zeta(s:A)z.l_i(s)]_{i \in I}, l_i = a_i]_{i \in I}$$

$$\text{class}(A, a_i)_{I,I} \triangleq \text{class}(A, a_i)_{i \in I}$$

where $a_i : A \rightarrow B_i$.

Note that the inclusion $\text{Class}(A)_{I,I} <: \text{Class}(A)_{Ins,Sub}$ does not hold in general, because of invariance in the type of *new*. However, for any $Ins, Sub \subseteq I$ we do obtain:

$$\text{class}(A, a_i)_{Ins} : \text{Class}(A)_{Ins,Sub}$$

because $\text{class}(A, a_i)_{Ins}$ has type $\text{Class}(A)_{Ins,I}$ by construction, and because $\text{Class}(A)_{Ins,I} <: \text{Class}(A)_{Ins,Sub}$. In essence, any class built according to our standard pattern can be given restricted instance and subclass interfaces.

Particular values of *Ins* and *Sub* correspond to common situations. Among others, the common notions of abstract, concrete, and leaf classes, as well as public, protected, and private methods [97, 109, 115], can be characterized as follows:

$c : \text{Class}(A)_{\emptyset,Sub}$	is an abstract class based on A
$c : \text{Class}(A)_{Ins,\emptyset}$	is a leaf class based on A
$c : \text{Class}(A)_{I,I}$	is a concrete class based on A
$c : \text{Class}(A)_{Pub,Pub}$	has public methods $l_i^{i \in Pub}$ and private methods $l_i^{i \in I-Pub}$

$c : Class(A)_{Pub, Pub \cup Pro}$ has public methods $l_i^{i \in I^{Pub}}$,
protected methods $l_i^{i \in I^{Pro}}$,
and private methods $l_i^{i \in I - Pub \cup Pro}$

Some subtleties of private method mechanisms are modeled by these definitions. In a class of the form $class(A, a_i^{i \in I})_{Pub} : Class(A)_{Pub, Pub}$, the pre-methods a_i have types $A \rightarrow B_i$. Therefore any pre-method may refer through self (of type A) to all other methods, including the private ones. However, a private method can be accessed only from the class in which it is declared, and not from objects generated from that class or from other classes. Similar considerations apply to protected methods.

A further variation on class definitions consists in changing the type of some pre-method l_j of $Class(A)$ from $A \rightarrow B_j$ to $C \rightarrow B_j$ for some C such that $A <: C$. While pre-methods of $Class(C)$ can normally be inherited into $Class(A)$, this change allows a pre-method l_j of the modified $Class(A)$ to be inherited “backwards” into $Class(C)$. This form of inheritance can sometimes be useful, but does not seem to lend itself to a systematic treatment.

8.5.3 An Example

As an exercise in the use of classes, we continue the cell example of Section 8.2; we illustrate the use of abstract classes, leaf classes, and their hybrids.

Corresponding to the type *PrivateCell*, we have the following class type:

```
PrivateCellClass  $\triangleq$ 
[new:PrivateCell,
 contents:PrivateCell  $\rightarrow$  Nat,
 get:PrivateCell  $\rightarrow$  Nat,
 set:PrivateCell  $\rightarrow$  Nat  $\rightarrow$  RomCell]
```

In our notation, *PrivateCellClass* is simply $Class(PrivateCell)$. A typical element of this class type is:

```
privateCellClass : PrivateCellClass  $\triangleq$ 
[new =
 $\zeta(z:PrivateCellClass)$ 
[contents =  $\zeta(s:PrivateCell)$  z.contents(s),
 get =  $\zeta(s:PrivateCell)$  z.get(s),
 set =  $\zeta(s:PrivateCell)$  z.set(s)],
contents =  $\lambda(s:PrivateCell)$  0,
get =  $\lambda(s:PrivateCell)$  s.contents,
set =  $\lambda(s:PrivateCell)$   $\lambda(n:Nat)$  s.contents := n]
```

By subsumption, *privateCellClass* can be seen as a trait, from which other classes can inherit pre-methods:

```

PrivateCellTrait  $\triangleq$ 
  [contents:PrivateCell $\rightarrow$ Nat,
   get:PrivateCell $\rightarrow$ Nat,
   set:PrivateCell $\rightarrow$ Nat $\rightarrow$ RomCell]

privateCellClass : PrivateCellTrait

```

Also by subsumption, *privateCellClass* can be seen as a leaf class, that can only be instantiated:

```

PrivateCellLeafClass  $\triangleq$ 
  [new:PrivateCell]

privateCellClass : PrivateCellLeafClass

privateCellClass.new : PrivateCell

```

The *contents* attribute is exposed in any object generated from *privateCellClass*. Since a user should interact with a cell only through the methods *get* and *set*, it is sensible to hide *contents*; for this purpose, we define:

```

RestrictedCellClass  $\triangleq$ 
  [new:PromCell,
   contents:PrivateCell $\rightarrow$ Nat,
   get:PrivateCell $\rightarrow$ Nat,
   set:PrivateCell $\rightarrow$ Nat $\rightarrow$ RomCell]

restrictedCellClass : RestrictedCellClass  $\triangleq$ 
  [new =
     $\varsigma(z:\textit{RestrictedCellClass})$ 
      [contents =  $\varsigma(s:\textit{PrivateCell}) z.\textit{contents}(s)$ ,
       get =  $\varsigma(s:\textit{PrivateCell}) z.\textit{get}(s)$ ,
       set =  $\varsigma(s:\textit{PrivateCell}) z.\textit{set}(s)$ ],
    contents =  $\lambda(s:\textit{PrivateCell}) 0$ ,
    get =  $\lambda(s:\textit{PrivateCell}) s.\textit{contents}$ ,
    set =  $\lambda(s:\textit{PrivateCell}) \lambda(n:\textit{Nat}) s.\textit{contents} := n$ ]
  ]

```

In our notation, *RestrictedCellClass* is $\text{Class}(\textit{PrivateCell})_{\{\textit{get}, \textit{set}\}, \{\textit{contents}, \textit{get}, \textit{set}\}}$. The code for *restrictedCellClass* is essentially identical to the code for *privateCellClass*. The only difference is in the type annotation for the methods of the class (e.g., for *new*), which has changed from *PrivateCellClass* to *RestrictedCellClass*. As a result of this change, we have *restrictedCellClass.new*: *PromCell*; thus *contents* is not visible in objects generated from *restrictedCellClass*.

Still, other classes can inherit the (trivial) implementation for *contents* because *restrictedCellClass.contents*: *PrivateCell* \rightarrow *Nat*. Applying subsumption once more, *contents* can be made invisible to other classes; we define:

$$\begin{aligned}
 NoContentsCellClass &\triangleq \\
 &[new:PromCell, \\
 &\quad get:PrivateCell \rightarrow Nat, \\
 &\quad set:PrivateCell \rightarrow Nat \rightarrow RomCell] \\
 restrictedCellClass : NoContentsCellClass
 \end{aligned}$$

In our notation, the type *NoContentsCellClass* is $Class(PrivateCell)_{\{get, set\}, \{get, set\}}$.

Note that the domain of the pre-methods *restrictedCellClass.get* and *restrictedCellClass.set* remains *PrivateCell*, which has a *contents* attribute. A class that inherits one of these pre-methods would have to provide its own implementation for *contents*. This implementation can be different from the original one.

For example, a cell may store a number in compressed form in a field *rep*. The attribute *contents* becomes a method that decompresses the number. We assume two primitives, *compress* and *uncompress*, for compressing and decompressing numbers. The method *set* is overridden so as to store inputs in *rep*. The method *get* is inherited from *restrictedCellClass*.

$$\begin{aligned}
 CompressedPrivateCell &\triangleq \\
 &[contents:Nat, rep:CompressedNat, get:Nat, set:Nat \rightarrow RomCell] \\
 CompressedPrivateCellClass &\triangleq \\
 &Class(CompressedPrivateCell) \\
 compressedPrivateCellClass : CompressedPrivateCellClass &\triangleq \\
 &[new = \\
 &\quad \zeta(z:CompressedPrivateCellClass) \\
 &\quad [contents = \zeta(s:CompressedPrivateCell) z.contents(s), \\
 &\quad \quad rep = \zeta(s:CompressedPrivateCell) z.rep(s), \\
 &\quad \quad get = \zeta(s:CompressedPrivateCell) z.get(s), \\
 &\quad \quad set = \zeta(s:CompressedPrivateCell) z.set(s)], \\
 &\quad contents = \lambda(s:CompressedPrivateCell) uncompress(s.rep), \\
 &\quad rep = \lambda(s:CompressedPrivateCell) compress(0), \\
 &\quad get = restrictedCellClass.get, \\
 &\quad set = \lambda(s:PrivateCell) \lambda(n:Nat) s.rep := compress(n)]]
 \end{aligned}$$

Finally, we define a leaf class:

$$\begin{aligned}
 PromCellLeafClass &\triangleq \\
 &[new:PromCell] \\
 restrictedCellClass : PromCellLeafClass
 \end{aligned}$$

Given a class of type *PromCellLeafClass*, we cannot tell whether the class or its instances have a *contents* attribute; we can only create an object of type *PromCell*.

8.6 Objects versus Records

In this section we compare object types with record types. We consider two natural but incorrect extensions of $\text{Ob}_{1\leq}$: method extraction and covariant object types. Taken together, these two failed extensions show that, in a calculus with subsumption, object typing is fundamentally different from record typing because both extensions are sound for records. Another extension, elder, allows us to define a new version of a method in terms of the previous one.

8.6.1 Records

To permit a precise comparison between objects and records, we introduce the basic rules for record types with subtyping.

As in Section 6.7.1 we write $\langle l_i = a_i \mid i \in 1..n \rangle$ for the record with labels l_i and fields a_i ; we write $r \cdot l_j$ for record selection, and $r \cdot l_j := b$ for functional record update. Functional record update can be defined from the other operations.

A record has type $\langle l_i; B_i \mid i \in 1..n \rangle$ if it has components l_i with types B_i . Unlike our object types, record types have a covariant subtyping rule.

Δ_{Rcd}

(Type Record) $(l_i \text{ distinct})$

$$E \vdash B_i \quad \forall i \in 1..n$$

$$\frac{}{E \vdash \langle l_i; B_i \mid i \in 1..n \rangle}$$

(Sub Record) $(l_i \text{ distinct})$

$$E \vdash B_i <: B'_i \quad \forall i \in 1..n \quad E \vdash B_i \quad \forall i \in n+1..n+m$$

$$\frac{}{E \vdash \langle l_i; B_i \mid i \in 1..n+m \rangle <: \langle l_i; B'_i \mid i \in 1..n \rangle}$$

(Val Record)

$$E \vdash b_i : B_i \quad \forall i \in 1..n$$

$$\frac{}{E \vdash \langle l_i = b_i \mid i \in 1..n \rangle : \langle l_i; B_i \mid i \in 1..n \rangle}$$

(Val Record Select)

$$E \vdash a : \langle l_i; B_i \mid i \in 1..n \rangle \quad j \in 1..n$$

$$\frac{}{E \vdash a \cdot l_j : B_j}$$

The basic equational rules for records are cut-down versions of the rules for objects, for the special case where all attributes are fields.

$\Delta_{=\text{Rcd}}$

(Eq Record)

$$E \vdash b_i \leftrightarrow b'_i : B_i \quad \forall i \in 1..n$$

$$\frac{}{E \vdash \langle l_i = b_i \mid i \in 1..n \rangle \leftrightarrow \langle l_i = b'_i \mid i \in 1..n \rangle : \langle l_i; B_i \mid i \in 1..n \rangle}$$

(Eq Sub Record)

$E \vdash b_i : B_i \quad \forall i \in 1..n+m$	$E \vdash \langle l_i = b_i^{i \in 1..n} \rangle \leftrightarrow \langle l_i = b_i^{i \in 1..n+m} \rangle : \langle l_i : B_i^{i \in 1..n} \rangle$
(Eq Record Select) $E \vdash a \leftrightarrow a' : \langle l_i : B_i^{i \in 1..n} \rangle \quad j \in 1..n$	(Eval Record Select) (where $a \equiv \langle l_i = b_i^{i \in 1..n} \rangle$) $E \vdash a : \langle l_i : B_i^{i \in 1..n} \rangle \quad j \in 1..n$

$$E \vdash a \cdot l_j \leftrightarrow a' \cdot l_j : B_j$$

$$E \vdash a \cdot l_j \leftrightarrow b_j : B_j$$

Functional record update is definable as follows:

$$a \cdot l_j := b \triangleq \langle l_j = b, l_i = a \cdot l_i^{i \in (1..n) - \{j\}} \rangle \quad \text{for } a : \langle l_i : B_i^{i \in 1..n} \rangle$$

Derived rules for this operation are:

(Val Record Update)

$$\frac{E \vdash a : \langle l_i : B_i^{i \in 1..n} \rangle \quad E \vdash b : B \quad j \in 1..n}{E \vdash a \cdot l_j := b : \langle l_j : B, l_i : B_i^{i \in (1..n) - \{j\}} \rangle}$$

(Eq Record Update)

$$\frac{E \vdash a \leftrightarrow a' : \langle l_i : B_i^{i \in 1..n} \rangle \quad E \vdash b \leftrightarrow b' : B \quad j \in 1..n}{E \vdash a \cdot l_j := b \leftrightarrow a' \cdot l_j := b' : \langle l_j : B, l_i : B_i^{i \in (1..n) - \{j\}} \rangle}$$

(Eval Record Update) (where $a \equiv \langle l_i = b_i^{i \in 1..n} \rangle$)

$$\frac{E \vdash a : \langle l_i : B_i^{i \in 1..n} \rangle \quad E \vdash b : B \quad j \in 1..n}{E \vdash a \cdot l_j := b \leftrightarrow \langle l_j = b, l_i = b_i^{i \in (1..n) - \{j\}} \rangle : \langle l_j : B, l_i : B_i^{i \in (1..n) - \{j\}} \rangle}$$

8.6.2 Method Extraction

Let us assume we have an operation for extracting a method from an object, as discussed at the beginning of Chapter 8. Considering the analogous rules for records, it would seem natural to give the following rules:

(Val Extract) (where $A \equiv [l_i : B_i^{i \in 1..n}]$)

$$\frac{E \vdash a : A \quad j \in 1..n}{E \vdash a \cdot l_j : A \rightarrow B_j}$$

(Eval Extract) (where $A \equiv [l_i : B_i^{i \in 1..n}]$, $a \equiv [l_i = \zeta(x_i; A') b_i^{i \in 1..n+m}]$)

$$\frac{E \vdash a : A \quad j \in 1..n}{E \vdash a \cdot l_j \leftrightarrow \lambda(x_j; A) b_j : A \rightarrow B_j}$$

These rules amount to interpreting an object type $A \equiv [l_i : B_i^{i \in 1..n}]$ as a recursively defined record-of-functions type $A \equiv \langle l_i : A \rightarrow B_i^{i \in 1..n} \rangle$.

The extraction operation is unsound, as the following example shows:

$$\begin{array}{ll}
 P \triangleq [x:Int, f:Int] & \\
 p \triangleq [x=1, f=1] & p : P \quad \text{by (Val Object)} \\
 Q \triangleq [x,y:Int, f:Int] & Q <: P \quad \text{by (Sub Object)} \\
 a \triangleq [x=1, y=1, f=\zeta(s:Q)s.x+s.y] & a : Q \quad \text{by (Val Object), hence also} \\
 b \triangleq a.f & a : P \quad \text{by (Val Subsumption)} \\
 & b : P \rightarrow Int \text{ by (Val Extract),} \\
 & \quad \text{with } b \leftrightarrow \lambda(s:P)s.x+s.y : P \rightarrow Int \\
 & \quad \text{by (Eval Extract)}
 \end{array}$$

Now we have $\emptyset \vdash b(p) \leftrightarrow (\lambda(s:P)s.x+s.y)(p) \leftrightarrow p.x+p.y : Int$, via (Eval Beta), but p does not have a y component. If we change an A to A' in the conclusion of (Eval Extract), which becomes $E \vdash a.l_j \leftrightarrow \lambda(x:A')b_j : A' \rightarrow B_j$, then the (Eval Beta) step is not even possible.

Hence it is unsound to add the method extraction operation to $\mathbf{FOb}_{1\leftarrow}$ (or to $\mathbf{Ob}_{1\leftarrow}$ with encoded function types), although it is sound to add it to \mathbf{FOb}_1 (or to \mathbf{Ob}_1).

8.6.3 Elder

Although in general it is unsound to extract a method, it is sound to refer to the previous value of a method in the course of an update. Just as the new value of a record field can be defined from its old value, the new code for a method can reuse the updated code. We write $a.l_j \Leftarrow (y:B_j)\zeta(x:A)b$ for the result of updating the l_j method of a . When l_j is subsequently invoked, x is self and y (which we call *elder*) is bound to the body of the old method l_j .

The type and evaluation rules for update with elder are:

Rules for elder

(Val Update with elder) (where $A \equiv [l_i:B_i \mid i \in 1..n]$)

$$\frac{E \vdash a : A \quad E, x:A, y:B_j \vdash b : B_j \quad j \in 1..n}{E \vdash a.l_j \Leftarrow (y:B_j)\zeta(x:A)b : A}$$

(Eval Update with elder) (where $A \equiv [l_i:B_i \mid i \in 1..n]$, $a \equiv [l_i=\zeta(x_i:A')b_i \mid i \in 1..n+m]$, $x_j \notin \text{dom}(E)$)

$$\frac{E \vdash a : A \quad E, x:A, y:B_j \vdash b[x,y] : B_j \quad j \in 1..n}{E \vdash a.l_j \Leftarrow (y:B_j)\zeta(x:A)b[x,y] \leftrightarrow [l_j=\zeta(x_j:A')b[l_j,b_j], l_i=\zeta(x_i:A')b_i \mid i \in (1..n+m)-\{j\}] : A}$$

Update with elder could easily be implemented using method extraction. But, unlike method extraction, update with elder is sound because we never apply the old method to an arbitrary element of type A . The old method's self remains bound to the object's self, of type A' .

In a calculus like ours, with primitive objects but without primitive classes, this mechanism provides a form of inheritance; it allows the reuse of methods of existing objects in new objects. For example, consider a two-dimensional point p with a *draw*

method that produces a picture with the point in it. This point would have the type $P_{xyd} \triangleq [x,y:\text{Real}, draw:\text{Bitmap}]$. We may change the *draw* method in order to invert the color of the picture, reusing the existing drawing code: $p.draw = (e:\text{Bitmap}) \zeta(s:P_{xyd}) \text{invert}(e)$.

For the sake of simplicity, we do not add elder to our core calculi. It is possible to achieve roughly the effect of elder by using second-order techniques [3].

8.6.4 Covariant Object Types

The subtyping rule for object types should be compared with the analogous rule for product types. Object types are invariant in their components, whereas product types are covariant in both their components. Similarly, record types $\langle l_i : A_i \mid i \in 1..n \rangle$, which generalize product types, are covariant in all their components.

It is natural to ask what happens if we allow object types to be covariant in their components. Suppose we adopt the following more liberal subtyping rule for objects:

$$\frac{(\text{Sub Object/covariant}) \quad (l_i \text{ distinct})}{\begin{array}{c} E \vdash B_i <: B'_i \quad E \vdash B_j \quad \forall i \in 1..n, j \in n+1..m \\ \hline E \vdash \langle l_i : B_i \mid i \in 1..n+m \rangle <: \langle l_i : B'_i \mid i \in 1..n \rangle \end{array}}$$

Then we can derive a contradiction. Let *PosReal* be the type of positive reals. Assume $\text{PosReal} <: \text{Real}$ and $\ln : \text{PosReal} \rightarrow \text{Real}$ (the natural logarithm). Define:

$$\begin{array}{ll} P \triangleq [x:\text{Real}, f:\text{Real}] & [x=1.0, f=\zeta(s:P)\ln(s.x)] : P \text{ is not derivable} \\ Q \triangleq [x:\text{PosReal}, f:\text{Real}] & \text{we have } Q <: P \text{ by (Sub Object/covariant)} \\ a \triangleq [x=1.0, f=\zeta(s:Q)\ln(s.x)] & \text{we have } a : Q \text{ by (Val Object),} \\ b \triangleq a.x := -1.0 & \text{hence also } a : P \text{ by (Val Subsumption)} \\ & \text{from } a : P \text{ we have } b : P \text{ by (Val Update),} \\ & \text{therefore } b.f : \text{Real} \end{array}$$

Now we have $\emptyset \vdash b.f \leftrightarrow \ln(-1.0) : \text{Real}$; we have derived a typing for a program that applies a function to an argument outside its domain. Hence the type system is unsound as a direct consequence of adding (Sub Object/covariant). The preceding example can be recast with other nontrivial subtypings in place of $\text{PosReal} <: \text{Real}$, such as a subtyping between object types. Note that the example involves simple field update, and does not require proper method update.

The basic reason for this problem is that each method relies on the types of the other methods, through the type of self. This dependence is essentially contravariant and hence is incompatible with covariance. The problem disappears if we disallow certain updates; this solution is explored in the next section.

8.7 Variance Annotations

Our basic object types are invariant in their components. In this section we study the possibility of building variance into object types. The techniques developed in this sec-

tion are not formally incorporated in our first-order calculi. We include variance in the formal treatment of some of our calculi in Parts II and III.

8.7.1 Object Types with Variance

Variance properties are expressed by annotating each component of an object type as invariant, covariant, or contravariant. Invariant components are the familiar ones. Covariant components allow covariant subtyping, but prevent updating. Contravariant components allow contravariant subtyping, but prevent invocation. Invariant components can be regarded, by subtyping, as either covariant or contravariant.

Variance annotations are attractive, from the points of view of both programming and theory. From a programming perspective, variance annotations support flexible subtyping, along with protection from unwanted reading or writing. From a theoretical perspective, they model a range of realistic features in an orthogonal and relatively simple way. Variance annotations can be encoded using quantifiers, but the encoding is not simple (see Chapter 18).

In this section we describe an extension of $\text{Ob}_{1\leq}$ with variance annotations. Object types now have the form $[l; v_i : B_i]^{i \in 1..n}$ where each v_i (a variance annotation) is one of the symbols $^-, ^0$, and $^+$, for contravariance, invariance, and covariance, respectively.

Object types with variance annotations

(Type Object) $(l_i \text{ distinct}, v_i \in \{^0, ^-, ^+\})$

$E \vdash B_i \quad \forall i \in 1..n$

$E \vdash [l; v_i : B_i]^{i \in 1..n}$

By convention, any omitted v 's are taken to be equal to 0 . Therefore we obtain the object types of $\text{Ob}_{1\leq}$ as abbreviations.

The rule for object subtyping is the most affected by the addition of variance annotations. This rule still says, to a first approximation, that a longer object type on the left is a subtype of a shorter one on the right. Because of the variance annotations, however, we use a new, auxiliary judgment $E \vdash v B <: v' B'$ for inclusion of attributes with variance. The rule becomes:

Subtyping with variance annotations

(Sub Object) $(l_i \text{ distinct})$

$E \vdash v_i B_i <: v'_i B'_i \quad \forall i \in 1..n \quad E \vdash B_i \quad \forall i \in n+1..n+m$

$E \vdash [l; v_i : B_i]^{i \in 1..n+m} <: [l; v'_i : B'_i]^{i \in 1..n}$

(Sub Invariant)

$E \vdash B$

$E \vdash {}^0 B <: {}^0 B$

(Sub Covariant)

$E \vdash B <: B' \quad v \in \{^0, ^+\}$

$E \vdash v B <: {}^+ B'$

(Sub Contravariant)

$E \vdash B' <: B \quad v \in \{^0, {}^-\}$

$E \vdash v B <: {}^- B'$

These four rules say:

- (Sub Object) Components that occur both on the left and on the right are handled by the other three rules. For components that occur only on the left, the component types must be well-formed.
- (Sub Invariant) An invariant component on the right requires an identical one on the left.
- (Sub Covariant) A covariant component type on the right can be a supertype of a corresponding component type on the left, either covariant or invariant. Intuitively, an invariant component can be regarded as covariant.
- (Sub Contravariant) A contravariant component type on the right can be a subtype of a corresponding component type on the left, either contravariant or invariant. Intuitively, an invariant component can be regarded as contravariant.

For example, these rules imply that if $B <: B'$ then $[l^0:B'] <: [l^-:B]$ and $[l^0:B] <: [l^+:B']$, but $[l^-:B]$ and $[l^+:B']$ are unrelated.

To preserve soundness in the presence of variant components, the rules for selection and update are restricted: selection cannot operate on contravariant components and update cannot operate on covariant components:

Typing with variance annotations

(Val Object) (where $A \equiv [l_1v_i:B_i]^{i \in 1..n}]$)

$$E, x_i:A \vdash b_i : B_i \quad \forall i \in 1..n$$

$$\underline{E \vdash [l_i = \zeta(x_i:A)b_i]^{i \in 1..n} : A}$$

(Val Select)

$$E \vdash a : [l_1v_i:B_i]^{i \in 1..n} \quad v_j \in \{^0, +\} \quad j \in 1..n$$

$$\underline{E \vdash a.l_j : B_j}$$

(Val Update) (where $A \equiv [l_1v_i:B_i]^{i \in 1..n}]$)

$$E \vdash a : A \quad E, x:A \vdash b : B_j \quad v_j \in \{^0, -\} \quad j \in 1..n$$

$$\underline{E \vdash a.l_j = \zeta(x:A)b : A}$$

The rule (Val Object) is essentially unchanged since we add variance annotations only to object types, not to objects.

8.7.2 Encodings and Examples with Variance

We show the expressiveness of variance annotations by representing variant type operators as object types with variance annotations.

We begin with covariant product types:

$$\begin{aligned} B_1 \times^+ B_2 &\triangleq [lft^+:B_1, rht^+:B_2] \\ (b_1, b_2) &\triangleq [lft=\zeta(x:B_1 \times^+ B_2)b_1, rht=\zeta(x:B_1 \times^+ B_2)b_2] \quad \text{for } x \notin FV(b_1) \cup FV(b_2) \end{aligned}$$

We can show that if $b_1 : B_1$ and $b_2 : B_2$, then $(b_1, b_2) : B_1 \times^+ B_2$ with a dummy assumption $x : B_1 \times^+ B_2$ for the self variables.

The encoding of records works similarly:

$$\begin{aligned} (l_i; B_i^{i \in 1..n})^+ &\triangleq [l_i^+: B_i^{i \in 1..n}] \\ (l_i = b_i^{i \in 1..n}) &\triangleq [l_i = \zeta(x_i; (l_i; B_i^{i \in 1..n})^+)b_i^{i \in 1..n}] \quad \text{for } x_i \notin FV(b_i) \text{ and } b_i : B_i \\ a \cdot l_j &\triangleq a.l_j \quad \text{for } a : (l_j; B_j^{j \in 1..n})^+, j \in 1..n \end{aligned}$$

The rules for records given in Section 8.6.1 are validated by this encoding. Thus covariance annotations unify object types and record types.

As shown in Section 7.7, functions can be expressed in \mathbf{Ob}_1 and hence in $\mathbf{Ob}_{1\leftarrow}$. A function $\lambda(x:A)b$ with argument type A and result type B receives the type $[arg^0:A, val^0:B]$. This typing, however, is invariant in both A and B . With variance annotations, we can define a type construction $A \rightarrow^{-+} B$, contravariant in A and covariant in B :

$$\begin{aligned} A \rightarrow^{-+} B &\triangleq [arg^-:A, val^+:B] \\ \text{obtaining } [arg^0:A, val^0:B] &<: A \rightarrow^{-+} B \end{aligned}$$

The encoding of abstractions and applications is unchanged:

$$\begin{aligned} \lambda(x:A)b\{x\} &\triangleq \\ &[\arg = \zeta(x:[\arg^0:A, \val^0:B])x.\arg, \val = \zeta(x:[\arg^0:A, \val^0:B])b\{x \leftarrow x.\arg\}] \\ b(a) &\triangleq (b.\arg := a).\val \end{aligned}$$

We have that $\lambda(x:A)b\{x\}$ has type $[arg^0:A, val^0:B]$, as before, and hence has also type $A \rightarrow^{-+} B$, by subsumption. Application still typechecks: if $b : A \rightarrow^{-+} B$ and $a : A$, then in $b(a)$ the assignment to \arg is legal because of the contravariance of \arg , and the invocation of \val is legal because of the covariance of \val .

This encoding illustrates the use of an intermediate type with invariant components as a subtype of a desired type with variant components. As in this encoding, we can often begin by defining objects with invariant components that can be internally read and written through self. Then, by subsumption, certain components can be protected against external reads or writes, obtaining covariance and contravariance without preventing internal reads and writes. In the encoding of λ -terms, $x.\arg$ can be read internally, although externally it can only be written.

As a further example, we decorate the cell types of Section 8.2 with variance annotations. We define the types:

$$\begin{aligned} ProtectedRomCell &\triangleq [get^+:Nat] \\ ProtectedPromCell &\triangleq [get^+:Nat, set^+:Nat \rightarrow ProtectedRomCell] \\ ProtectedPrivateCell &\triangleq [contents:Nat, get^+:Nat, set^+:Nat \rightarrow ProtectedRomCell] \end{aligned}$$

The $^+$ annotations protect the methods get and set from updates from the outside. We obtain the sub typings $ProtectedPrivateCell <: ProtectedPromCell <: ProtectedRomCell$.

9 RECURSION

In the typed calculi considered so far, we can find some typings for objects that use or modify self. However, we cannot find satisfactory typings for objects whose methods return either self or an updated self: this feature calls for the use of recursive types. For example, it is natural to give a recursive type to an object with a method that returns self; the result type of the method is the type of the object, so the type of the object must be defined recursively.

In this chapter we discuss recursion in first-order calculi with and without subtyping, and we provide typings for the untyped examples of Section 6.5. The interaction of subtyping and recursion is a delicate one. Considering our examples, we find that we do not obtain all the subtypings we might like for recursive object types. We discuss several remedies, including dynamic typing.

9.1 Recursion

To formulate the rules for recursive types, we need to introduce type variables, which are typically denoted by X and Y . We also need to substitute types for type variables:

Notation

- The substitution of a type C for the free occurrences of X in B is written $B\{X \leftarrow C\}$. Similarly $b\{X \leftarrow C\}$ is the substitution of C for X in b .
- $B\{X\}$ and $b\{X\}$ are used to highlight that X may occur free in B and b , respectively.
- $B\{C\}$ and $b\{C\}$ stand for $B\{X \leftarrow C\}$ and $b\{X \leftarrow C\}$, respectively, when X is clear from context.

A recursive type $\mu(X)B\{X\}$ is the unique solution of the equation $X = B\{X\}$, up to isomorphism. The two types $\mu(X)B\{X\}$ and $B\{\mu(X)B\{X\}\}$ are therefore isomorphic. Their isomorphism is given by two constructs, *fold* and *unfold*, that can be used explicitly to move around the isomorphism. For example, if we want to define a type A isomorphic to $[l:A]$, we let $A \triangleq \mu(X)[l:X]$. If a has type A then $\text{unfold}(a)$ has type $[l:A]$, and $\text{fold}(A, \text{unfold}(a))$ has type A and is equal to a ; note, however, that a itself does not have type $[l:A]$.

In the literature (e.g., [18]), there is an alternative treatment of recursive types, which we do not adopt. In that treatment, the two types $\mu(X)B\{X\}$ and $B\{\mu(X)B\{X\}\}$ are equal. This means, in particular, that the two types have the same sets of elements; it also means that the two constructs *fold* and *unfold* are not necessary. This treatment of

recursive types is more concise than ours, but it has some technical disadvantages. For example, it requires defining a notion of type equality. These technical disadvantages are the main reason why we prefer to distinguish the types $\mu(X)B\{X\}$ and $B\{\mu(X)B\{X\}\}$, and to use the constructs *fold* and *unfold*. In a programming language with recursive types, the isomorphism is usually kept implicit for conciseness. This simplification can be obtained easily even when the language is based on a calculus with explicit isomorphism, as we show in Chapter 12.

Formally, we have the following fragments for type variables and recursive types:

Δ_X

$$\frac{\begin{array}{c} (\text{Env } X) \\ E \vdash \diamond \quad X \notin \text{dom}(E) \end{array}}{E, X \vdash \diamond} \quad \frac{(\text{Type } X)}{E', X, E'' \vdash \diamond}$$

$$\frac{}{E', X, E'' \vdash X}$$

Δ_μ

$$\frac{(\text{Type Rec})}{E, X \vdash A}$$

$$\frac{}{E \vdash \mu(X)A}$$

(Val Fold) (where $A \equiv \mu(X)B\{X\}$)

$$E \vdash b : B\{A\}$$

$$\frac{}{E \vdash \text{fold}(A,b) : A}$$

(Val Unfold) (where $A \equiv \mu(X)B\{X\}$)

$$E \vdash a : A$$

$$\frac{}{E \vdash \text{unfold}(a) : B\{A\}}$$

We obtain the calculi:

$$\begin{array}{lll} \mathbf{Ob}_{1\mu} & \triangleq & \mathbf{Ob}_1 \cup \Delta_X \cup \Delta_\mu \\ \mathbf{F}_{1\mu} & \triangleq & \mathbf{F}_1 \cup \Delta_X \cup \Delta_\mu \\ \mathbf{FOb}_{1\mu} & \triangleq & \mathbf{FOb}_1 \cup \Delta_X \cup \Delta_\mu \end{array} \quad \begin{array}{l} \mathbf{Ob}_1 \text{ with recursion} \\ \mathbf{F}_1 \text{ with recursion} \\ \mathbf{FOb}_1 \text{ with recursion} \end{array}$$

with equations:

$\Delta_{=\mu}$

$$\frac{(\text{Eq Fold}) \quad (\text{where } A \equiv \mu(X)B\{X\})}{E \vdash b \leftrightarrow b' : B\{A\}}$$

$$\frac{}{E \vdash \text{fold}(A,b) \leftrightarrow \text{fold}(A,b') : A}$$

$$\frac{(\text{Eq Unfold}) \quad (\text{where } A \equiv \mu(X)B\{X\})}{E \vdash a \leftrightarrow a' : A}$$

$$\frac{}{E \vdash \text{unfold}(a) \leftrightarrow \text{unfold}(a') : B\{A\}}$$

$$\frac{(\text{Eval Fold}) \quad (\text{where } A \equiv \mu(X)B\{X\})}{E \vdash a : A}$$

$$\frac{}{E \vdash \text{fold}(A,a) \leftrightarrow a : A}$$

$$\frac{(\text{Eval Unfold}) \quad (\text{where } A \equiv \mu(X)B\{X\})}{E \vdash b : B\{A\}}$$

$$\frac{}{E \vdash \text{unfold}(\text{fold}(A,b)) \leftrightarrow b : B\{A\}}$$

Consider, for example, the untyped self-returning object $[l=\zeta(x)x]$. To make this object typable in $\mathbf{Ob}_{1\mu}$, we insert a type annotation for the self variable and two *fold* operations. With $A \triangleq \mu(X)[l:X]$, the rules yield that $a \triangleq \text{fold}(A,[l=\zeta(x:[l:A])\text{fold}(A,x)])$ has type A , and that $\text{unfold}(a).l$ is equal to a .

The translation of \mathbf{F}_1 into \mathbf{Ob}_1 of Section 7.7 extends to a corresponding translation of $\mathbf{F}_{1\mu}$ into $\mathbf{Ob}_{1\mu}$. The untyped λ -calculus can be translated into $\mathbf{F}_{1\mu}$; the encodings of untyped λ -terms receive the recursive type $\mu(X)X \rightarrow X$. Combining these two translations, we obtain a translation of the untyped λ -calculus into $\mathbf{Ob}_{1\mu}$:

Typed translation of the untyped λ -calculus

$\Lambda \triangleq \mu(X)[arg:X, val:X]$	(globally free variables should be assigned type Λ)
$U\Lambda \triangleq [arg:\Lambda, val:\Lambda]$	(the unfolding of Λ)
$\rho \in Var \rightarrow \mathbf{Ob}_{1\mu}\text{-term}$	
$\rho(x) \triangleq x$	
$(\rho\{y \leftarrow a\})(x) \triangleq \text{if } x = y \text{ then } a \text{ else } \rho(x)$	
$\langle x \rangle_\rho \triangleq \rho(x)$	
$\langle b(a) \rangle_\rho \triangleq$	
$(\text{unfold}(\langle b \rangle_\rho).arg = \zeta(x:U\Lambda)\langle a \rangle_\rho).val$	for $x \notin FV(\langle a \rangle_\rho)$
$\langle \lambda(x)b \rangle_\rho \triangleq$	
$\text{fold}(\Lambda,$	
$[arg = \zeta(x:U\Lambda)x.arg,$	
$val = \zeta(x:U\Lambda)(\langle b \rangle_{\rho\{x \leftarrow x.arg\}})]$	

We saw in Section 7.7 that, for each type A , recursively defined values $\mu(x:A)b$ of type A are definable from objects; using the recursive type $\mu(X)X \rightarrow A$, they are definable from functions as well [67]:

$$\begin{aligned} \mu(x:A)b\{x\} &\triangleq \\ &(\lambda(x:\mu(X)X \rightarrow A)b[\text{unfold}(x)(x)]) \\ &(\text{fold}(\mu(X)X \rightarrow A, (\lambda(x:\mu(X)X \rightarrow A)b[\text{unfold}(x)(x)]))) \end{aligned}$$

Therefore, recursively defined values can be encoded in any of $\mathbf{Ob}_{1\mu}$, $\mathbf{F}_{1\mu}$ and $\mathbf{FOb}_{1\mu}$.

9.2 Recursion and Subsumption

To add recursion to a calculus with subtyping we cannot reuse directly the fragments Δ_X and Δ_μ of Section 9.1, because type variables in environments now require subtype bounds. Therefore we introduce new fragments $\Delta_{<:X}$ and $\Delta_{<:\mu}$. In $\Delta_{<:X}$ the components of environments are type variables with bounds, of the form $X <: A$. The fragment $\Delta_{<:\mu}$ includes a rule for subtyping recursive types, (Sub Rec), which uses the bounded type variables. We may still write E, X, E' , but now as an abbreviation for $E, X <: Top, E'$.

$\Delta_{<:X}$

(Env $X <:$)	(Type $X <:$)	(Sub X)
$E \vdash A \quad X \notin \text{dom}(E)$	$E', X <: A, E'' \vdash \diamond$	$E', X <: A, E'' \vdash \diamond$
$E, X <: A \vdash \diamond$	$E', X <: A, E'' \vdash X$	$E', X <: A, E'' \vdash X <: A$

With type variables, we can revise and generalize the definition of variance given in Section 8.1. We say that $A\{X\}$ is *covariant* in X if $B <: B'$ implies $A\{B\} <: A\{B'\}$ for all B and B' . We say that $A\{X\}$ is *contravariant* in X if $B <: B'$ implies $A\{B'\} <: A\{B\}$ for all B and B' . Otherwise, we say that $A\{X\}$ is *invariant* in X .

 $\Delta_{<:\mu}$

(Type Rec $<:$)	(Sub Rec)
$E, X <: \text{Top} \vdash A$	$E \vdash \mu(X)A \quad E \vdash \mu(Y)B \quad E, Y <: \text{Top}, X <: Y \vdash A <: B$
$E \vdash \mu(X)A$	$E \vdash \mu(X)A <: \mu(Y)B$

(Val Fold) (where $A \equiv \mu(X)B\{X\}$)	(Val Unfold) (where $A \equiv \mu(X)B\{X\}$)
$E \vdash b : B\{A\}$	$E \vdash a : A$
$E \vdash \text{fold}(A,b) : A$	$E \vdash \text{unfold}(a) : B\{A\}$

 $\Delta_{= <:\mu}$

same as $\Delta_{=\mu}$

The most interesting new rule in $\Delta_{<:\mu}$ is (Sub Rec). For establishing the subtyping of two recursive types, $\mu(X)A <: \mu(Y)B$, the rule (Sub Rec) requires a subtype relation between their bodies, $A <: B$, under the assumption of a subtype relation between their recursion variables, $X <: Y$.

This rule determines the variance behavior of recursive types. If A is covariant in X , then the variance of Z in $\mu(X)A$ is the same as the variance of Z in A . But if A is contravariant in X , then $\mu(X)A$ may be only invariant in Z , even if A is covariant or contravariant in Z . For example, $\mu(X)X \rightarrow Z$ is invariant in Z .

The rule (Sub Rec) is part of a complete system for first-order recursive types, studied in [18]. In that system the types $\mu(X)B\{X\}$ and $B\{\mu(X)B\{X\}\}$ are equivalent. We do not assume this equivalence; as a consequence, in the context of our calculi, rules stronger than (Sub Rec) can be found, such as the following one:

(Sub Rec')
$E \vdash \mu(X)A\{X\} \quad E \vdash \mu(Y)B\{Y\} \quad E, X <: \mu(Y)B\{Y\} \vdash A\{X\} <: B\{\mu(Y)B\{Y\}\}$
$E \vdash \mu(X)A\{X\} <: \mu(Y)B\{Y\}$

This rule is denotationally and operationally sound, and it implies (Sub Rec) as an admissible rule. The standard (Sub Rec) is sufficient for almost all our purposes so, for simplicity, we do not adopt (Sub Rec'). We leave as open questions all matters of completeness (in the sense of [18]).

The equational rules for recursion are not affected by subtyping. However, in the presence of subtyping, some interesting rules can be derived. The following rule can prove the equality of two *fold* terms even when the types used for folding are different:

$$\frac{(Eq\;Fold<:\;Lemma1)\;(where\;A \equiv \mu(X)B\{X\},\;A' \equiv \mu(Y)B'\{Y\})}{E \vdash A \quad E \vdash A' \quad E,\;Y <: Top,\;X <: Y \vdash B <: B' \quad E \vdash b \leftrightarrow b' : B\{A\}} \\ E \vdash fold(A,b) \leftrightarrow fold(A',b') : A'$$

The hypotheses of this rule imply a subtype relation between the types used for folding: $A <: A'$. For the derivation of this rule, we use transitivity on the following intermediate results: $E \vdash fold(A,b) \leftrightarrow fold(A,b') : A'$; $E \vdash fold(A,b') \leftrightarrow fold(A',unfold(fold(A,b'))) : A'$; and $E \vdash fold(A',unfold(fold(A,b'))) \leftrightarrow fold(A',b') : A'$. A further rule is derivable from (Eq Fold<: Lemma1) and (Eq Fold):

$$\frac{(Eq\;Fold<:\;Lemma2)\;(where\;A \equiv \mu(X)B\{X\},\;A' \equiv \mu(Y)B'\{Y\})}{E \vdash A \quad E \vdash A' \quad E,\;Y <: Top,\;X <: Y \vdash B <: B' \quad E \vdash b \leftrightarrow b' : B'\{A'\} \quad E \vdash b : B\{A\}} \\ E \vdash fold(A,b) \leftrightarrow fold(A',b') : A'$$

For the derivation of this rule, we use (Eq Fold<: Lemma1) with $b \equiv b'$ and (Eq Fold) for $E \vdash b \leftrightarrow b' : B'\{A\}$, and apply transitivity to the two results.

We now obtain the calculi:

$$\begin{aligned} \mathbf{Ob}_{1<:\mu} &\triangleq \mathbf{Ob}_{1<} \cup \Delta_{<:X} \cup \Delta_{<:\mu} \\ \mathbf{F}_{1<:\mu} &\triangleq \mathbf{F}_{1<} \cup \Delta_{<:X} \cup \Delta_{<:\mu} \\ \mathbf{FOb}_{1<:\mu} &\triangleq \mathbf{FOb}_{1<} \cup \Delta_{<:X} \cup \Delta_{<:\mu} \end{aligned}$$

$\mathbf{FOb}_{1<:\mu}$ is the strongest of these first-order systems, and $\mathbf{Ob}_{1<:\mu}$ is a rather mild restriction of it. We summarize the definition of $\mathbf{FOb}_{1<:\mu}$ with the following syntax and scoping rules:

Syntax of the $\mathbf{FOb}_{1<:\mu}$ calculus

$A, B ::=$	types
X	type variable
K	ground type
Top	the biggest type
$[l_i; B_i \ i \in 1..n]$	object type (l_i distinct)
$A \rightarrow B$	function type
$\mu(X)A$	recursive type

$a, b ::=$	terms
x	variable
$[l_i = \zeta(x_i; A_i) b_i]^{i \in 1..n}$	object (l_i ; distinct)
$a.l$	method invocation
$a.l \Leftarrow \zeta(x; A) b$	method update
$\lambda(x; A) b$	function
$b(a)$	application
$fold(A, a)$	recursive fold
$unfold(a)$	recursive unfold

Scoping for the $\text{FOb}_{1<;\mu}$ calculus

$FV(X)$	$\triangleq \{X\}$
$FV(K)$	$\triangleq \{\}$
$FV(Top)$	$\triangleq \{\}$
$FV([l_i; B_i]^{i \in 1..n})$	$\triangleq \bigcup^{i \in 1..n} FV(B_i)$
$FV(A \rightarrow B)$	$\triangleq FV(A) \cup FV(B)$
$FV(\mu(X)A)$	$\triangleq FV(A) - \{X\}$
$FV(\zeta(x; A) b)$	$\triangleq FV(A) \cup (FV(b) - \{x\})$
$FV(x)$	$\triangleq \{x\}$
$FV([l_i = \zeta(x_i; A_i) b_i]^{i \in 1..n})$	$\triangleq \bigcup^{i \in 1..n} FV(\zeta(x_i; A_i) b_i)$
$FV(a.l)$	$\triangleq FV(a)$
$FV(a.l \Leftarrow \zeta(x; A) b)$	$\triangleq FV(a) \cup FV(\zeta(x; A) b)$
$FV(\lambda(x; A) b)$	$\triangleq FV(A) \cup (FV(b) - \{x\})$
$FV(b(a))$	$\triangleq FV(b) \cup FV(a)$
$FV(fold(A, a))$	$\triangleq FV(A) \cup FV(a)$
$FV(unfold(a))$	$\triangleq FV(a)$

9.3 Some Properties of $\text{Ob}_{1<;\mu}$

In this section, we extend the basic results about $\text{Ob}_{1<:}$ to $\text{Ob}_{1<;\mu}$. As usual, we expect analogous properties to hold for $\text{FOb}_{1<;\mu}$.

9.3.1 Minimum Types

We prove the minimum-types property for a restricted version $r\text{Ob}_{1<;\mu}$ of $\text{Ob}_{1<;\mu}$. In this version, the rules (Env x) and (Val Subsumption) are applicable only for closed types, that is, we replace (Env x) and (Val Subsumption) with the weaker rules:

Restricted rules for $\mathbf{rOb}_{1<:\mu}$

(Env x Restricted)

$$\frac{E \vdash \diamond \quad \emptyset \vdash A \quad x \notin \text{dom}(E)}{E, x:A \vdash \diamond}$$

(Val Subsumption Restricted)

$$\frac{E \vdash a : A \quad \emptyset \vdash A <: B}{E \vdash a : B}$$

The restriction implies that if $E \vdash a : A$ is derivable then A is closed. However, it does not affect the typings obtainable in an empty environment. The restriction simplifies our argument; a technique for handling the general case is described in [56] in the context of a different calculus.

In order to prove the minimum-types property for $\mathbf{rOb}_{1<:\mu}$ we consider a system $\mathbf{rMinOb}_{1<:\mu}$ obtained from $\mathbf{rOb}_{1<:\mu}$ by removing (Val Subsumption), by modifying (Val Object) and (Val Update) as in $\mathbf{MinOb}_{1<:}$ from Section 8.3.1, and by modifying (Val Fold) as follows:

Modified rule for fold for $\mathbf{rMinOb}_{1<:\mu}$ (Val Min Fold) (where $A \equiv \mu(X)B(X)$)

$$\frac{E \vdash b : C \quad \emptyset \vdash C <: B\{A\}}{E \vdash \text{fold}(A,b) : A}$$

The next four propositions are the analogues of Propositions 8.3-1 through 8.3-4.

Proposition 9.3-1 ($\mathbf{rMinOb}_{1<:\mu}$ typings are $\mathbf{rOb}_{1<:\mu}$ typings)

If $E \vdash a : A$ is derivable in $\mathbf{rMinOb}_{1<:\mu}$ then it is also derivable in $\mathbf{rOb}_{1<:\mu}$.

□

Proposition 9.3-2 ($\mathbf{rMinOb}_{1<:\mu}$ has unique types)

If $E \vdash a : A$ and $E \vdash a : A'$ are derivable in $\mathbf{rMinOb}_{1<:\mu}$ then $A \equiv A'$.

□

Proposition 9.3-3 ($\mathbf{rMinOb}_{1<:\mu}$ has smaller types than $\mathbf{rOb}_{1<:\mu}$)

If $E \vdash a : A$ is derivable in $\mathbf{rOb}_{1<:\mu}$ then $E \vdash a : A'$ is derivable in $\mathbf{rMinOb}_{1<:\mu}$ for some A' such that $E \vdash A' <: A$ is derivable (in either system).

□

Proposition 9.3-4 ($\mathbf{rOb}_{1<:\mu}$ has minimum types)

In $\mathbf{rOb}_{1<:\mu}$, if $E \vdash a : A$ then there exists B such that $E \vdash a : B$ and, for any A' , if $E \vdash a : A'$ then $E \vdash B <: A'$.

□

From this we obtain a restricted result for the original system $\mathbf{Ob}_{1<:\mu}$.

Corollary 9.3-5 ($\mathbf{Ob}_{1<:\mu}$ has minimum types in an empty environment)

In $\mathbf{Ob}_{1<:\mu}$, if $\emptyset \vdash a : A$ then there exists B such that $\emptyset \vdash a : B$ and, for any A' , if $\emptyset \vdash a : A'$ then $\emptyset \vdash B <: A'$.

□

9.3.2 Subject Reduction

We extend the operational semantics to treat *fold* and *unfold*. The set of results now includes the terms of the form $\text{fold}(A, v)$, where v is a result.

Operational semantics for *fold* and *unfold*

(Red Fold)

$$\vdash a \rightsquigarrow v$$

$$\vdash \text{fold}(A, a) \rightsquigarrow \text{fold}(A, v)$$

(Red Unfold)

$$\vdash a \rightsquigarrow \text{fold}(A, v)$$

$$\vdash \text{unfold}(a) \rightsquigarrow v$$

In order to prove subject reduction, we extend the bound weakening and substitution lemmas of Section 8.3.2. The substitution lemma requires a notion of environment substitution.

Environment substitution

$E\{Y \leftarrow C\}$ for $Y \notin \text{dom}(E)$ is defined by:

$$\begin{array}{lll} \emptyset\{Y \leftarrow C\} & \triangleq & \emptyset \\ (E, x:A)\{Y \leftarrow C\} & \triangleq & E\{Y \leftarrow C\}, x:A\{Y \leftarrow C\} \\ (E, X <: A)\{Y \leftarrow C\} & \triangleq & E\{Y \leftarrow C\}, X <: A\{Y \leftarrow C\} \end{array}$$

Lemma 9.3-6 (Bound weakening)

If $E, X <: D, E' \vdash \mathcal{S}$ and $E \vdash D' <: D$, then $E, X <: D', E' \vdash \mathcal{S}$.

If $E, x:D, E' \vdash \mathcal{S}$ and $E \vdash D' <: D$, then $E, x:D', E' \vdash \mathcal{S}$.

□

Lemma 9.3-7 (Substitution)

If $E, X <: A, E'[X] \vdash \mathfrak{S}\{X\}$ and $E \vdash A' <: A$, then $E, E'[A'] \vdash \mathfrak{S}\{A'\}$.

If $E, x:D, E' \vdash \mathfrak{S}\{x\}$ and $E \vdash d : D$, then $E, E' \vdash \mathfrak{S}\{d\}$.

□

The proof of subject reduction is then routine.

Theorem 9.3-8 (Subject reduction for $\text{Ob}_{1<:\mu}$)

Let c be a closed term and v be a result, and assume $\vdash c \rightsquigarrow v$.

If $\emptyset \vdash c : C$, then $\emptyset \vdash v : C$.

Proof

We add two cases to the proof of Theorem 8.3-7, which is by induction on the derivation of $\vdash c \rightsquigarrow v$.

Case (Red Fold)

Suppose $\vdash \text{fold}(A,a) \rightsquigarrow \text{fold}(A,v)$ because $\vdash a \rightsquigarrow v$. Assume that $\emptyset \vdash \text{fold}(A,a) : C$. Then A has the form $\mu(X)B\{X\}$, $\emptyset \vdash A <: C$, and $\emptyset \vdash a : B\{A\}$. By induction hypothesis, we have $\emptyset \vdash v : B\{A\}$. Therefore we obtain $\emptyset \vdash \text{fold}(A,v) : B\{A\}$ by (Val Fold), and $\emptyset \vdash \text{fold}(A,v) : C$ by subsumption.

Case (Red Unfold)

Suppose $\vdash \text{unfold}(a) \rightsquigarrow v$ because $\vdash a \rightsquigarrow \text{fold}(A,v)$. Assume that $\emptyset \vdash \text{unfold}(a) : C$. Then $\emptyset \vdash a : \mu(X)B\{X\}$ for some type of the form $\mu(X)B\{X\}$ such that $\emptyset \vdash B\{\mu(X)B\} <: C$. By induction hypothesis, we have $\emptyset \vdash \text{fold}(A,v) : \mu(X)B$. This implies that A has the form $\mu(X')B'\{X'\}$, that $\emptyset \vdash \mu(X')B' <: \mu(X)B$, and that $\emptyset \vdash v : B'\{\mu(X')B'\}$. Since $\emptyset \vdash \mu(X')B' <: \mu(X)B$, either $\mu(X')B'$ is $\mu(X)B$ or $\emptyset, X <: \text{Top}, X' <: X \vdash B' <: B$. In the former case, $\emptyset \vdash v : B\{\mu(X)B\}$. In the latter case, by Lemma 9.3-7, $\emptyset \vdash B'\{\mu(X')B'\} <: B\{\mu(X)B\}$, and we obtain $\emptyset \vdash v : B\{\mu(X)B\}$ again, by subsumption. In either case it follows that $\emptyset \vdash v : C$ by subsumption.

□

9.4 Examples

In this section we provide first-order typings for the examples of Section 6.5. All the examples are revisited in Section 15.2.

We start with the example of objects with backup; the type in question is:

$$Bk \triangleq \mu(X)[\text{retrieve}:X, \text{backup}:X, \dots]$$

We obtain the following typed version of the code, where $UBk \triangleq [\text{retrieve}:Bk, \text{backup}:Bk, \dots]$ is the unfolding of Bk :

$$\begin{aligned} &\text{fold}(Bk, \\ &[\text{retrieve} = \varsigma(s_1:UBk) \text{fold}(Bk, s_1), \end{aligned}$$

$$\begin{aligned} \text{backup} &= \zeta(s_2:\text{UBk}) \text{fold}(\text{Bk}, s_2.\text{retrieve} \neq \zeta(s_1:\text{UBk}) \text{fold}(\text{Bk}, s_2)), \\ &\dots \\ \}] &: \text{Bk} \end{aligned}$$

If we now consider the types $\text{Point} \triangleq [x,y:\text{Real}]$ and $\text{PointBk} \triangleq \mu(X)[\text{retrieve}:X, \text{backup}:X, x,y:\text{Real}]$, we obtain $\text{PointBk} <: \text{Point}$ modulo an unfolding. Thus points with backup can subsume points.

We use similar techniques to treat storage cells:

$$\begin{aligned} \text{Cell} &\triangleq \mu(X)[\text{contents}:\text{Nat}, \text{get}:\text{Nat}, \text{set}:\text{Nat} \rightarrow X] \\ \text{UCell} &\triangleq [\text{contents}:\text{Nat}, \text{get}:\text{Nat}, \text{set}:\text{Nat} \rightarrow \text{Cell}] \\ \text{myCell} : \text{Cell} &\triangleq \\ &\text{fold}(\text{Cell}, \\ &\quad [\text{contents} = 0, \\ &\quad \text{get} = \zeta(s:\text{UCell}) s.\text{contents}, \\ &\quad \text{set} = \zeta(s:\text{UCell}) \lambda(n:\text{Nat}) \text{fold}(\text{Cell}, s.\text{contents} := n)]) \end{aligned}$$

We can also write a typed version of the movable points:

$$\begin{aligned} P_1 &\triangleq \mu(X)[x:\text{Int}, mv_x:\text{Int} \rightarrow X] && \text{movable one-dimensional points} \\ P_2 &\triangleq \mu(X)[x,y:\text{Int}, mv_x,mv_y:\text{Int} \rightarrow X] && \text{movable two-dimensional points} \\ \text{origin}_1 &\triangleq \\ &\text{fold}(P_1, \\ &\quad [x = 0, \\ &\quad mv_x = \zeta(s:[x:\text{Int}, mv_x:\text{Int} \rightarrow P_1]) \lambda(dx:\text{Int}) \text{fold}(P_1, s.x := s.x+dx)]) \end{aligned}$$

and of the calculator:

$$\text{Calc} \triangleq \mu(X)[\text{arg}, \text{acc}:\text{Real}, \text{enter}:\text{Real} \rightarrow X, \text{add}, \text{sub}:X, \text{equals}:\text{Real}]$$

The example of the numerals requires some additions to our first-order type system. One possibility is second-order types, to be studied later. A simpler, first-order alternative is sum types, if we are prepared to rephrase the example. We add a type construction $A+B$, with the operations $\text{inl}_{AB} : A \rightarrow (A+B)$, $\text{inr}_{AB} : B \rightarrow (A+B)$, and $\text{if}_{ABC} : (A+B) \rightarrow (A \rightarrow C) \rightarrow (B \rightarrow C) \rightarrow C$. (From now on, we omit the subscripts.) Informally, $A+B$ is the disjoint union of A and B , inl and inr are the obvious injections, and if is used to examine elements of $A+B$ returning elements of C . For convenience, we also add a type Unit with an element unit . The numerals can be expressed using sums as follows:

$$\begin{aligned} \text{Nat} &\triangleq \mu(X)[\text{case}:\text{Unit}+X, \text{succ}:X] \\ \text{zero} &\triangleq \\ &\text{fold}(\text{Nat}, \\ &\quad [\text{case} = \text{inl}(\text{unit}), \\ &\quad \text{succ} = \zeta(x:[\text{case}:\text{Unit}+\text{Nat}, \text{succ}:\text{Nat}]) \\ &\quad \text{fold}(\text{Nat}, x.\text{case} := \text{inr}(\text{fold}(\text{Nat}, x))))]) \end{aligned}$$

$$\begin{aligned} \text{iszero} &\triangleq \lambda(n:\text{Nat}) \text{ if}(\text{unfold}(n).\text{case})(\lambda(u:\text{Unit}) \text{ true})(\lambda(p:\text{Nat}) \text{ false}) \\ \text{pred} &\triangleq \lambda(n:\text{Nat}) \text{ if}(\text{unfold}(n).\text{case})(\lambda(u:\text{Unit}) \text{ zero})(\lambda(p:\text{Nat}) p) \end{aligned}$$

Although this code looks quite different from that of Section 6.5.3, the two versions are related by a type isomorphism (explained in Section 15.2.4).

9.5 The Shortcomings of First-Order Typing

The $\text{FOb}_{1<:\mu}$ calculus looks very promising because it adds subtyping to a rich first-order theory. For example, we can use it to write types of movable points, as we did in Section 9.4:

$$\begin{array}{ll} P_1 \triangleq \mu(X)[x:\text{Int}, mv_x:\text{Int}\rightarrow X] & \text{movable one-dimensional points} \\ P_2 \triangleq \mu(X)[x,y:\text{Int}, mv_x,mv_y:\text{Int}\rightarrow X] & \text{movable two-dimensional points} \end{array}$$

In addition, since we have subtyping, we may expect to obtain that $P_2 <: P_1$, because intuitively P_2 extends P_1 . However, this inclusion is not provable with our rules, because the invariance of object types blocks the application of (Sub Rec) to the result type of mv_x . In fact, the inclusion $P_2 <: P_1$ is unsound, as we show next.

We derive an inconsistency from $P_2 <: P_1$. Briefly, if we assume $P_2 <: P_1$, we can use subsumption from $p : P_2$ to $p : P_1$ and then update the mv_x method of p with one that returns a proper element of P_1 . Then some other method of p may go wrong because it expects mv_x to produce an element of P_2 . More precisely, we can construct the following counterexample:

$$\begin{array}{ll} UP_1 \triangleq [x:\text{Int}, mv_x:\text{Int}\rightarrow P_1] & (\text{the unfolding of } P_1) \\ UP_2 \triangleq [x,y:\text{Int}, mv_x,mv_y:\text{Int}\rightarrow P_2] & (\text{the unfolding of } P_2) \\ p_2 : P_2 \triangleq \\ \quad \text{fold}(P_2, \\ \quad [x = \zeta(s_2:UP_2) \text{ unfold}(s_2.mv_x(1)).y, \\ \quad y = 0, \\ \quad mv_x = \zeta(s_2:UP_2) \lambda(dx:\text{Int}) \text{ fold}(P_2, s_2), \\ \quad mv_y = \zeta(s_2:UP_2) \lambda(dy:\text{Int}) \text{ fold}(P_2, s_2)]) \\ p_1 : P_1 \triangleq \text{fold}(P_1, [x = 0, mv_x = \zeta(s_1:UP_1) \lambda(dx:\text{Int}) \text{ fold}(P_1, s_1)]) \\ p : P_1 \triangleq p_2 & (\text{retyping } p_2 \text{ using the assumption } P_2 <: P_1) \\ q : P_1 \triangleq \text{fold}(P_1, \text{unfold}(p).mv_x := \lambda(dx:\text{Int}) p_1) & \end{array}$$

We have:

$$\begin{aligned} \text{unfold}(q).x &= (\text{unfold}(p_2).mv_x := \lambda(dx:\text{Int}) p_1).x \end{aligned}$$

$$\begin{aligned}
 &= [x = \zeta(s_2: UP_2) \text{ unfold}(s_2.mv_x(1)).y, mv_x = \lambda(dx:Int) p_1, \\
 &\quad y = \dots, mv_y = \dots].x \\
 &= \text{unfold}(p_1).y
 \end{aligned}$$

But $\text{unfold}(p_1)$ does not have a y component.

As we have just seen, the failure of $P_2 <: P_1$ is necessary. At the same time, it is unacceptable: in the common situation where a method returns an updated self, we lose all useful subsumption relations.

In many programming languages, these difficulties are avoided by not allowing a subclass to change the type of a method of a superclass. In our calculus, similarly, these difficulties are avoided if we define mv_x to return P_1 even when embedded in P_2 :

$$\begin{aligned}
 P_1 &\triangleq \mu(X)[x:Int, mv_x:Int \rightarrow X] \\
 P_2^+ &\triangleq \mu(X)[x,y:Int, mv_x:Int \rightarrow P_1, mv_y:Int \rightarrow X] \\
 UP_1 &\triangleq [x:Int, mv_x:Int \rightarrow P_1] && (\text{the unfolding of } P_1) \\
 UP_2^+ &\triangleq [x,y:Int, mv_x:Int \rightarrow P_1, mv_y:Int \rightarrow P_2^+] && (\text{the unfolding of } P_2^+)
 \end{aligned}$$

Then we have $UP_2^+ <: UP_1$, so we can at least convert every point $p : P_2^+$ to a point $\text{fold}(P_1, \text{unfold}(p))$ of type P_1 . It is possible to strengthen the type theory to identify recursive types with their unfoldings, as in [18]; then we can obtain directly $P_2^+ <: P_1$.

After these changes, the pre-method mv_x can be inherited because its result type remains the same. However, whenever we invoke mv_x on an element of P_2^+ , we “forget” its second dimension. For this kind of solution to be useful, Simula and Modula-3, among other languages, provide dynamic testing of membership in a subtype of a given type, so that the forgotten information can be recovered (see Section 2.5). In our example we would test for membership in the subtype P_2^+ of P_1 . We would abandon, though, the truly static typing of subsumption. We explore this approach in Section 9.7.

Another possible solution is based on the variance annotations of Section 8.7. Using variance annotations, we can rewrite:

$$\begin{aligned}
 P_1^v &\triangleq \mu(X)[x:Int, mv_x^+:Int \rightarrow X] \\
 P_2^v &\triangleq \mu(X)[x,y:Int, mv_x^+, mv_y^+:Int \rightarrow X]
 \end{aligned}$$

The subtyping $P_2^v <: P_1^v$ holds because of the covariance of mv_x . The counterexample above is avoided because mv_x can no longer be updated.

This solution might seem sufficient at least for class-based languages, where methods cannot be updated. Unfortunately, subtyping problems resurface in the treatment of classes and inheritance. Assume, for simplicity, the equivalence of P_1^v and P_2^v with their respective unfoldings, and consider the class types associated with P_1^v and P_2^v (see Section 8.5):

$$\begin{aligned}
 \text{Class}(P_1^v) &\triangleq [new:P_1^v, mv_x:P_1^v \rightarrow Int \rightarrow P_1^v, \dots] \\
 \text{Class}(P_2^v) &\triangleq [new:P_2^v, mv_x:P_2^v \rightarrow Int \rightarrow P_2^v, \dots]
 \end{aligned}$$

Now, $P_1^v \rightarrow Int \rightarrow P_1^v <: P_2^v \rightarrow Int \rightarrow P_2^v$ fails because of the result types. Hence the pre-method mv_x cannot be inherited from a class c_1 of type $\text{Class}(P_1^v)$ to a subclass c_2 of

type $\text{Class}(P_2^v)$. Such an inheritance would be unsound in general, because mv_x in c_2 must return a P_2^v , while mv_x in c_1 may return a P_1^v that is not a P_2^v .

This conclusion is unacceptable as well: many methods have the type of self as their result type, and many of these methods can be soundly inherited (for example, if they simply return self). A task of a good type system should be to distinguish the sound instances of inheritance from the unsound ones. At the very least, we need a more sophisticated treatment of classes.

All these problems are less severe in imperative languages, where the mv_x method could be redefined to cause a side-effect on the host point and return nothing. Then the type of the modified method would not depend on the type of self, and P_1 and P_2 would not be recursive (see Section 11.2.1). Even in imperative languages, though, we often find methods that allocate new objects of the type of self and return them; it is not reasonable to forbid recursive types and to require the use of side-effects in situations where side-effects are not natural.

9.6 Towards the Type Self

In Part II, we look for a satisfactory treatment of self-returning methods. A natural idea in this direction is to handle the type of self in some special way. A proposal introduces a distinguished type as the type of self. This type, which we call *Self*, denotes in every subtype of an object type, the subtype in question (see Section 2.8). Intuitively, *Self* is the partially unknown type of the self parameter of each method. Using *Self*, we would wish to write the types:

$$\begin{aligned} P_1' &\triangleq [x:\text{Int}, mv_x:\text{Int}\rightarrow\text{Self}] \\ P_2' &\triangleq [x,y:\text{Int}, mv_x,mv_y:\text{Int}\rightarrow\text{Self}] \end{aligned}$$

with the following desirable properties:

$$\begin{aligned} P_2' &<: P_1' \\ \text{if } p_1' : P_1' \text{ and } p_2' : P_2', \text{ then } p_1'.mv_x : \text{Int}\rightarrow P_1' \text{ and } p_2'.mv_x : \text{Int}\rightarrow P_2' \end{aligned}$$

Several object-oriented languages have included *Self* in their type systems [88, 116]. Therefore it seems important to have a precise understanding of *Self*. The counterexample to $P_2 <: P_1$ given in Section 9.5 still looms. In fact, Eiffel has soundness problems because of such counterexamples [55]; these problems are addressed only outside the typing discipline.

Still, it may be possible to use the type *Self*, instead of recursive types, and to adopt ad hoc typing rules for *Self* that avoid unsoundness. An appealing idea is to require that any term meant to update a method returning *Self* should be *parametric in Self*: that is, it should always return a modification of its self parameter, and never return a fixed object [61]. This requirement blocks the counterexample in Section 9.5, while allowing the definition of useful movable points.

The tentative conclusion we should draw here is that recursive types are the wrong way of modeling the type *Self* (whatever this is). We could now proceed to extend

$\mathbf{Ob}_{1<\mu}$ with some ad hoc notion of Self. However, this approach would involve a fair amount of “guessing” to find the right typing rules, which are not at all obvious. Instead, we turn to second-order systems in Part II. We shall see that a plain second-order extension of $\mathbf{Ob}_{1<\mu}$ can already provide a concept of Self, essentially because there we can express the requirement of being parametric in Self. After studying the rules for Self derivable in second-order systems, in Chapter 16 we return to the problem of axiomatizing Self directly.

9.7 Dynamic Typing

As an alternative to extending our calculi with the type Self, we could add dynamic typing to $\mathbf{FOb}_{1<\mu}$. This addition would preserve the first-order character of the calculus, and enable many realistic examples; it is methodologically problematic, as we discussed in the Review, but it is technically simple.

In this section, we describe the technical aspects of adding a particular *typecase* construct to $\mathbf{FOb}_{1<\mu}$; other forms of *typecase* are possible [8].

9.7.1 Typecase

The *typecase* construct evaluates a term to a result, and discriminates on the type of the result. We write:

$$\text{typecase } a \mid (x:A)d_1 \mid d_2$$

If a yields a result of type A , then $(\text{typecase } a \mid (x:A)d_1 \mid d_2)$ returns d_1 with x replaced by this result. If a yields a result that does not have type A , then $(\text{typecase } a \mid (x:A)d_1 \mid d_2)$ returns d_2 .

The typing rule for the *typecase* construct is:

Typing rule for *typecase*

(Val Typecase)

$$E \vdash a : A' \quad E, x:A \vdash d_1 : D \quad E \vdash d_2 : D$$

$$\frac{}{E \vdash \text{typecase } a \mid (x:A)d_1 \mid d_2 : D}$$

This typing rule has, as first hypothesis, that a is well-typed; the precise type (A') is irrelevant. The body of the first branch, d_1 , is typed under the assumption that x has type A . The two branches have the same type, D , which is also the type of the whole *typecase* expression.

It seems hard to write satisfactory equational rules for *typecase*. Moreover, in the presence of *typecase*, the equational theory that we have developed so far is unsound (in particular because *typecase* can distinguish the two terms that (Eq Sub Object) equates). For these reasons, we consider only reduction rules for *typecase*.

In programming languages that include *typecase*, the type of a value is represented using a tag attached to the value; *typecase* relies on this tag to perform run-time type

discrimination. An operational semantics for an untyped calculus, such as the one of Section 6.2.4, could be augmented with tagged results in order to handle *typecase* [8].

We adopt an operational semantics without such tags; in our semantics, *typecase* performs run-time type discrimination by constructing a typing derivation. Our rules do not immediately suggest an efficient implementation, but have the advantage of being simple and general, applying equally well to calculi with and without subtyping.

Operational semantics for *typecase*

(Red Typecase Match)

$$\frac{\vdash a \rightsquigarrow v' \quad \emptyset \vdash v' : A \quad \vdash d_1[v'] \rightsquigarrow v}{\vdash \text{typecase } a \mid (x:A)d_1[x] \mid d_2 \rightsquigarrow v}$$

(Red Typecase Else)

$$\frac{\vdash a \rightsquigarrow v' \quad \emptyset \not\vdash v' : A \quad \vdash d_2 \rightsquigarrow v}{\vdash \text{typecase } a \mid (x:A)d_1 \mid d_2 \rightsquigarrow v}$$

These rules are unusual in that the result of a reduction depends on whether a typing is derivable. If $\emptyset \vdash v' : A$ is derivable, then the first rule is applicable. Otherwise (when $\emptyset \not\vdash v' : A$), the second rule is applicable; thus the second rule has a negative assumption.

9.7.2 Subject Reduction

Although *typecase* permits dynamic typing, the static typing rules remain consistent. In fact, it is straightforward to extend our subject reduction results to *typecase*.

Lemma 9.7-1 (Bound weakening)

If $E, X <: D, E' \vdash \mathfrak{S}$ and $E \vdash D' <: D$, then $E, X <: D', E' \vdash \mathfrak{S}$.

If $E, x:D, E' \vdash \mathfrak{S}$ and $E \vdash D' <: D$, then $E, x:D', E' \vdash \mathfrak{S}$.

□

Lemma 9.7-2 (Substitution)

If $E, X <: A, E'[X] \vdash \mathfrak{S}[X]$ and $E \vdash A' <: A$, then $E, E'[A'] \vdash \mathfrak{S}[A']$.

If $E, x:D, E' \vdash \mathfrak{S}[x]$ and $E \vdash d : D$, then $E, E' \vdash \mathfrak{S}[d]$.

□

Theorem 9.7-3 (Subject reduction for $\text{Ob}_{1<:\mu}$ with *typecase*)

Let c be a closed term and v be a result, and assume $\vdash c \rightsquigarrow v$.

If $\emptyset \vdash c : C$, then $\emptyset \vdash v : C$.

Proof

We add two cases to the previous proof of subject reduction (Theorem 9.3-8), which is by induction on the derivation of $\vdash c \rightsquigarrow v$.

Case (Red Typecase Match)

Suppose $\vdash \text{typecase } a \mid (x:A)d_1\{x\} \mid d_2 \rightsquigarrow v$ because $\vdash a \rightsquigarrow v'$, $\emptyset \vdash v' : A$, and $\vdash d_1\{v'\} \rightsquigarrow v$. Assume that $\emptyset \vdash \text{typecase } a \mid (x:A)d_1\{x\} \mid d_2 : C$. Then $\emptyset, x:A \vdash d_1\{x\} : D$ for some type D such that $\emptyset \vdash D <: C$. By Lemmas 9.7-1 and 9.7-2 we obtain $\emptyset \vdash d_1\{v'\} : D$. We obtain $\emptyset \vdash v : D$ by induction hypothesis, and $\emptyset \vdash v : C$ by subsumption.

Case (Red Typecase Else)

Suppose $\vdash \text{typecase } a \mid (x:A)d_1 \mid d_2 \rightsquigarrow v$ because $\vdash a \rightsquigarrow v'$, $\emptyset \not\vdash v' : A$, and $\vdash d_2 \rightsquigarrow v$. Assume that $\emptyset \vdash \text{typecase } a \mid (x:A)d_1 \mid d_2 : C$. Then $\emptyset \vdash d_2 : D$ for some type D such that $\emptyset \vdash D <: C$. We obtain $\emptyset \vdash v : D$ by induction hypothesis, and $\emptyset \vdash v : C$ by subsumption. (Note that the premise $\emptyset \not\vdash v' : A$ is not used in this case.)

□

9.7.3 An Example

Using *typecase*, we can recover lost type information. For example, in Section 9.5 we introduced the following types for movable points:

$$\begin{aligned} P_1 &\triangleq \mu(X)[x:Int, mv_x:Int \rightarrow X] \\ P_2^+ &\triangleq \mu(X)[x,y:Int, mv_x:Int \rightarrow P_1, mv_y:Int \rightarrow X] \\ UP_1 &\triangleq [x:Int, mv_x:Int \rightarrow P_1] && \text{(the unfolding of } P_1) \\ UP_2^+ &\triangleq [x,y:Int, mv_x:Int \rightarrow P_1, mv_y:Int \rightarrow P_2^+] && \text{(the unfolding of } P_2^+) \end{aligned}$$

with the inclusion $UP_2^+ <: UP_1$. Consider a two-dimensional point p :

$$\begin{aligned} p : P_2^+ &\triangleq \\ &fold(P_2^+, \\ &\quad [x = 0, y = 0, \\ &\quad mv_x = \zeta(s_2:UP_2^+) \lambda(dx:Int) fold(P_1, s_2.x := s_2.x + dx), \\ &\quad mv_y = \zeta(s_2:UP_2^+) \lambda(dy:Int) fold(P_2^+, s_2.y := s_2.y + dy)]) \end{aligned}$$

We have:

$$\begin{aligned} p' : P_1 &\triangleq unfold(p).mv_x(1) \\ unfold(p') &= \\ &[x = 1, y = 0, \\ &mv_x = \zeta(s_2:UP_2^+) \lambda(dx:Int) fold(P_1, s_2.x := s_2.x + dx), \\ &mv_y = \zeta(s_2:UP_2^+) \lambda(dy:Int) fold(P_2^+, s_2.y := s_2.y + dy)] \end{aligned}$$

Therefore, although p' has static type P_1 , the result of $unfold(p')$ is a two-dimensional point with type UP_2^+ . Using *typecase*, we can recover the forgotten y component of p' :

$$\text{typecase } unfold(p') \mid (up':UP_2^+) up'.y \mid -1 = 0$$

10 UNTYPED IMPERATIVE CALCULI

Object-oriented languages are naturally imperative, with methods performing side-effects on the internal state of objects. So far we have examined only stateless execution models. In the next two chapters we show that our theory of objects is applicable to imperative languages, by adapting our techniques to an operational semantics with side-effects.

In this chapter we develop a tiny but expressive untyped imperative calculus, an imperative variant of the calculus of Chapter 6. This imperative calculus forms the kernel of the *Obliq* programming language. The calculus comprises objects, method invocation, method update, object cloning, and local definitions. Although an object is usually seen as a bundle of mutable fields together with a method suite, in our calculus the method suite is mutable, so we can dispense with the fields.

We start with the syntax of a bare calculus. Then we detail a few encodings, and give some illustrative programming examples. In particular, we show how to express convenient constructs such as fields, procedures, data structures, and control structures. Finally, we formalize the operational semantics of the calculus.

10.1 Syntax

The $\text{imp}\zeta$ -calculus is a pure, imperative calculus of objects:

Syntax of the $\text{imp}\zeta$ -calculus

$a, b ::=$	terms
x	variable
$[l_i = \zeta(x_i)b_i]^{i \in 1..n}$	object (l_i distinct)
$a.l$	method invocation
$a.l \Leftarrow \zeta(x)b$	method update
$\text{clone}(a)$	cloning
$\text{let } x = a \text{ in } b$	let

As in Chapter 6, an object $[l_i = \zeta(x_i)b_i]^{i \in 1..n}$ is a collection of components $l_i = \zeta(x_i)b_i$, where l_i is a method name and $\zeta(x_i)b_i$ is a method. The binders $\zeta(x_i)$ suspend the evaluation of the bodies b_i , hence the order of the components $l_i = \zeta(x_i)b_i$ does not matter. Method invocation is still based on the self-application idea: if the method named l in a is $\zeta(x)b$, then $a.l$ executes the body b with x bound to a . Method update, $a.l \Leftarrow \zeta(y)b$, is

now imperative: it replaces the method named l in a with $\zeta(y)b$, and returns the modified object.

A cloning operation $clone(a)$ produces a new object with the same labels as a , with each component sharing the methods of the corresponding component of a . Cloning may not be strictly necessary as a primitive, but it is convenient in examples, and it is an interesting operation characteristic of prototype-based languages. Moreover, cloning enables us to translate the ζ -calculus into the $\text{imp}\zeta$ -calculus: we can express functional method update as cloning followed by imperative method update.

The *let* construct evaluates a term, binds the result to a variable, then evaluates a second term with that variable in scope. Sequential evaluation can be defined from *let*:

$$a, b \triangleq \text{let } x=a \text{ in } b \quad \text{for } x \notin FV(b)$$

10.2 Fields

In the $\text{imp}\zeta$ -calculus every component of an object contains a method, as in the ζ -calculus; we have avoided introducing a separate notion of field components. Methods are sufficient in functional calculi, where we can regard a field as a method that does not use its self parameter, a field selection as a method invocation on the field, and a field update as a method update on the field. So we could define:

$$\text{field: } [\dots, l=b, \dots] \triangleq [\dots, l=\zeta(x)b, \dots] \quad \text{for } x \notin FV(b)$$

$$\text{field selection: } o.l \triangleq o.l$$

$$\text{field update: } o.l:=b \triangleq o.l \Leftarrow \zeta(x)b \quad \text{for } x \notin FV(b)$$

These definitions have an unfortunate property: in both field definition and field update, the implicit $\zeta(x)$ binder suspends evaluation of the field until selection. This semantics is both inefficient and inadequate for an imperative calculus, because at every access the suspended fields are reevaluated and their side-effects are repeated.

Therefore we consider an alternative definition for fields, based on *let*. The *let* construct gives us a way of controlling execution flow; this feature is used in translating a calculus with eagerly evaluated fields, $\text{imp}\zeta_\ell$, into $\text{imp}\zeta$.

Syntax of the $\text{imp}\zeta$ -calculus

$a, b ::=$	terms
x	variable
$[l_i=b_i]_{i \in 1..n}, [l_j=\zeta(x_j)b_j]_{j \in n+1..n+m}$	object (l_i, l_j distinct)
$a.l$	field selection / method invocation
$a.l:=b$	field update
$a.l \Leftarrow \zeta(x)b$	method update
$clone(a)$	cloning

Here the $l_i=b_i$ are fields and the $l_j=\zeta(x_j)b_j$ are methods. Note that a field can be changed into a method by a method update, and a method into a field by a field update.

In $\text{imp}_{\zeta f}$, the semantics of an object may depend on the order of its components, because of side-effects in computing contents of fields. Therefore we set evaluation to proceed from left to right, and we do not identify objects up to reordering of their components. Usually, we write the field components before the method components.

We need not include sequencing and *let* in $\text{imp}_{\zeta f}$, because they can be defined as abbreviations:

$$\begin{aligned} \text{let } x=a \text{ in } b\{x\} &\triangleq [\text{def}=a, \text{val}=\zeta(x)b[\![x.\text{def}]\!]].\text{val} \\ a; b &\triangleq [\text{fst}=a, \text{snd}=b].\text{snd} \end{aligned}$$

The translation of the calculus with fields is:

Translation of $\text{imp}_{\zeta f}$ into imp_{ζ}

$\langle\!\langle x \rangle\!\rangle \triangleq x$	
$\langle\!\langle [l_i=b_i]_{i \in 1..n}, l_j=\zeta(x_j)b_j]_{j \in n+1..n+m} \rangle\!\rangle \triangleq$	for $y_i \notin FV(b_k)_{k \in 1..n+m}$, y_i distinct, $i \in 0..n$
$\text{let } y_1=\langle\!\langle b_1 \rangle\!\rangle \text{ in } \dots \text{ let } y_n=\langle\!\langle b_n \rangle\!\rangle \text{ in } [l_i=\zeta(y_0)y_i]_{i \in 1..n}, l_j=\zeta(x_j)\langle\!\langle b_j \rangle\!\rangle]_{j \in n+1..n+m}$	
$\langle\!\langle a.l \rangle\!\rangle \triangleq \langle\!\langle a \rangle\!\rangle.l$	
$\langle\!\langle a.l := b \rangle\!\rangle \triangleq$	for $y_i \notin FV(b)$, y_i distinct, $i \in 0..n$
$\text{let } y_1=\langle\!\langle a \rangle\!\rangle \text{ in let } y_2=\langle\!\langle b \rangle\!\rangle \text{ in } y_1.l = \zeta(y_0)y_2$	
$\langle\!\langle a.l \Leftarrow \zeta(x)b \rangle\!\rangle \triangleq \langle\!\langle a \rangle\!\rangle.l \Leftarrow \zeta(x)\langle\!\langle b \rangle\!\rangle$	
$\langle\!\langle \text{clone}(a) \rangle\!\rangle \triangleq \text{clone}(\langle\!\langle a \rangle\!\rangle)$	

Thus we have a choice between an object calculus with fields, field selection, and field update ($\text{imp}_{\zeta f}$), and one with *let* (imp_{ζ}). These calculi are inter-translatable. We adopt imp_{ζ} as primitive because it is more economical and it enables easier comparisons with our other calculi.

10.3 Procedures

Just as the objects of Chapter 6 can express functions, the imperative objects of this chapter can express procedures. In order to demonstrate this, we consider a call-by-value λ -calculus with side-effects, imp_{λ} , that includes abstraction, application, and assignment to λ -bound variables.

Syntax of the imp_{λ} -calculus

$a, b ::=$	terms
x	variable
$x := a$	assignment
$\lambda(x)b$	abstraction
$b(a)$	application

The intended semantics of this notation should be evident. For example, assuming that we have arithmetic primitives, $(\lambda(x) x:=x+1; x)(3)$ is an $\text{imp}\lambda$ -term whose evaluation yields 4.

Within $\text{imp}\lambda$ we can define abbreviations for local assignable variables and for sequencing:

$$\begin{aligned} \text{var } x=a \text{ in } b &\triangleq (\lambda(x)b)(a) \\ a; b &\triangleq (\lambda(x)b)(a) \text{ with } x \notin FV(b) \end{aligned}$$

We translate $\text{imp}\lambda$ into the calculus with fields, $\text{imp}\varsigma_f$. Thus we show that both $\text{imp}\varsigma_f$ and $\text{imp}\varsigma$ can express procedures.

Translation of the $\text{imp}\lambda$ -calculus into the $\text{imp}\varsigma_f$ -calculus

$$\begin{aligned} \rho \in Var \rightarrow \text{imp}\varsigma_f\text{-term} \\ \emptyset(x) &\triangleq x \\ (\rho(y \leftarrow a))(x) &\triangleq \text{if } x = y \text{ then } a \text{ else } \rho(x) \\ \langle x \rangle_\rho &\triangleq \rho(x) \\ \langle x := a \rangle_\rho &\triangleq x.\text{arg} := \langle a \rangle_\rho \\ \langle \lambda(x)b \rangle_\rho &\triangleq [\text{arg} = \varsigma(x)x.\text{arg}, \text{val} = \varsigma(x)\langle b \rangle_{\rho|x \leftarrow x.\text{arg}}] \\ \langle b(a) \rangle_\rho &\triangleq (\text{clone}(\langle b \rangle_\rho).\text{arg} := \langle a \rangle_\rho).\text{val} \end{aligned}$$

In the translation, an environment ρ maps each variable x either to $x.\text{arg}$ if x is λ -bound, or to x if x is a free variable. As in Section 6.3, a λ -abstraction is translated to an object with an arg component, for storing the argument, and a val method, for executing the body. The arg component is initially set to a divergent method, and is filled with an argument upon procedure call. A call activates the val method that can then access the argument through self as $x.\text{arg}$. An assignment $x := a$ updates $x.\text{arg}$, where the argument is stored (assuming that x is λ -bound).

A procedure needs to be cloned when it is called; the clone provides a fresh location in which to store the argument of the call, preventing interference with other calls of the same procedure. Such interference would derail recursive invocations.

This encoding has similarities with the mechanism of method activation in Self and Beta, and with a brief comment of Adams and Rees in [11]. It is unrelated to the encoding of functions with objects in Oaklisp [76].

We can extend the $\text{imp}\lambda$ -calculus into a calculus with procedures and objects, $\text{imp}\lambda\varsigma_f$. This extended calculus has procedures with call-by-value parameters and assignable formals, objects with fields and methods, constant declarations (let), variable declarations (var), and sequencing ($.$). Sequencing and let are defined as abbreviations as in $\text{imp}\varsigma_f$. The syntax of $\text{imp}\lambda\varsigma_f$ is simply the union of the syntaxes of $\text{imp}\lambda$ and $\text{imp}\varsigma_f$.

Syntax of the $\text{imp}\lambda\zeta_f$ -calculus

$a, b ::=$	terms
x	variable
$[l_i=b_i]^{i \in 1..n}, l_j=\zeta(x_j)b_j]^{j \in n+1..n+m}$	object (l_i, l_j distinct)
$a.l$	field selection / method invocation
$a.l := b$	field update
$a.l \Leftarrow \zeta(x)b$	method update
$\text{clone}(a)$	cloning
$x := a$	assignment
$\lambda(x)b$	abstraction
$b(a)$	application
(abbreviations)	
$\text{let } x = a \text{ in } b$	local identifier
$\text{var } x = a \text{ in } b$	local variable
$a; b$	sequencing

The translation of $\text{imp}\lambda\zeta_f$ into $\text{imp}\zeta_f$ is a trivial extension of the translation of $\text{imp}\lambda$ into $\text{imp}\zeta_f$. The $\text{imp}\lambda\zeta_f$ -calculus is fairly complete and convenient, and we use it in the following examples.

10.4 Examples

Our first examples deal with the encoding of basic types, booleans, and natural numbers, from which we derive iteration constructs. The encoding of basic types is adapted from previous chapters. As a programming example, we also present an implementation of the prime number sieve.

We use the notation:

$\text{let } x = a;$

for the top-level definition of an identifier that is used in later discussion and examples. Each occurrence of this notation can be recast as an expression:

$\text{let } x = a \text{ in } b$

where b is a collection of program fragments presented after the top-level definition. We still use \triangleq for equality by definition. The new notation is necessary because of the imperative nature of the examples in this chapter.

10.4.1 Booleans

The booleans $true$ and $false$ are objects with three methods: if , $then$, and $else$. By indirection through self , $true.\text{if}$ returns $true.\text{then}$, and $false.\text{if}$ returns $false.\text{else}$.

$$\begin{aligned}
 \text{let } \text{true} &= [\text{if} = \zeta(x) x.\text{then}, \text{then} = \zeta(x) x.\text{then}, \text{else} = \zeta(x) x.\text{else}]; \\
 \text{let } \text{false} &= [\text{if} = \zeta(x) x.\text{else}, \text{then} = \zeta(x) x.\text{then}, \text{else} = \zeta(x) x.\text{else}]; \\
 \text{if } b \text{ then } c \text{ else } d &\triangleq ((b.\text{then} \neq \zeta(x) c).\text{else} \neq \zeta(x) d).\text{if} \quad x \notin FV(c) \cup FV(d)
 \end{aligned}$$

The branch bodies c and d of the conditional are not immediately evaluated, since they are contained inside ζ -binders.

When a conditional is evaluated, the boolean b is modified in the components *then* and *else*. This side-effect does not matter because the relevant branch is used immediately, and any subsequent use of b will reset it appropriately. If desired, the conditional could be rewritten so as to clone b and to perform the side-effect on the clone of b .

10.4.2 Object-Oriented Natural Numbers

Our imperative numerals have a *case* field and a *succ* method. The *case* field of a numeral takes two procedures (that is, case branches) as parameters. The first branch is invoked in the *zero* case with a dummy argument (to delay evaluation). The second branch is invoked in the non-*zero* case, with the predecessor of the current numeral as argument. The *succ* method of a numeral produces its successor.

Therefore the *case* field for *zero* invokes its first argument. The *succ* method must convert any numeral into a non-*zero* numeral. First the current numeral (available as *self*) is cloned. Then the *case* field of the clone is modified to invoke the second branch with *self* as parameter. Finally the modified clone is returned.

$$\begin{aligned}
 \text{let } \text{zero} &= \\
 &[\text{case} = \lambda(z) \lambda(s) z([]), \\
 &\text{succ} = \zeta(x) \text{clone}(x).\text{case} := \lambda(z) \lambda(s) s(x)];
 \end{aligned}$$

Cloning protects a numeral from being permanently modified when its successor method is invoked. As an exercise, the reader may try to express the natural numbers without using cloning, via recursive definitions.

When *zero.succ* is executed, a clone of the numeral *zero* is transformed into the numeral *one*. In other words, *zero* is used as a prototype for *one*. Of the components of *zero*, the *succ* method is inherited, while the *case* field is updated. This transformation is an example of cloning plus update, typical of prototype-based languages.

A convenient control structure for primitive recursion over the natural numbers can be defined as follows:

$$\begin{aligned}
 \text{case } n \text{ when } 0 \text{ do } c, \text{when } m+1 \text{ do } d &\triangleq \\
 (n.\text{case}) (\lambda(w)c) (\lambda(m)d) &\quad \text{for } w \notin FV(c)
 \end{aligned}$$

10.4.3 Iteration

Booleans and natural numbers can be used to define *while* and *for* iterators, respectively:

$$\begin{aligned}
 \text{while } b \text{ do } c &\triangleq \\
 [l = \zeta(x) \text{ if } b \text{ then } (c; x.l) \text{ else } []].l \\
 \text{for } i \text{ in } 1..n \text{ do } c &\triangleq \\
 [l = \zeta(x) \lambda(i) \text{ case } i \text{ when } 0 \text{ do } [], \text{ when } m+1 \text{ do } (x.l(m); c; []).l(n) \\
 &\quad \text{for } x, m \notin FV(c)
 \end{aligned}$$

In the encoding of *for*, we could clone *n* and *m* to make sure that *c* can do no damage by causing a side-effect on *i*. Both the *while* construct and the *for* construct return the empty object.

10.4.4 A Prime Number Sieve

Our final example is an implementation of the prime number sieve. This example is meant to illustrate advanced usage of object-oriented features, and not necessarily transparent programming style.

```

let sieve =
  [m =  $\zeta(s) \lambda(n)$ 
   let sieve' = clone(s)
   in s.prime := n;
      s.next := sieve';
      s.m =  $\zeta(s') \lambda(n')$ 
            case (n' mod n)
            when 0 do [],
            when p+1 do sieve'.m(n');
   ],
  prime =  $\zeta(x) x.\text{prime}$ ,
  next =  $\zeta(x) x.\text{next}$ ;
  ];

```

The sieve starts as a root object which, whenever it receives a prime *p*, splits itself into a filter for multiples of *p*, and a clone of itself. As filters accumulate in a pipeline, they prevent multiples of known primes from reaching the root object. After the integers from 2 to *n* have been fed to the sieve, there are as many filter objects as there are primes smaller than or equal to *n*, plus a root object. Each prime is stored in its filter; the *n*-th prime can be recovered by scanning the pipeline for the *n*-th filter.

The sieve is used, for example, in the following way:

<i>for i in 1..99 do sieve.m(i.succ);</i>	(accumulate the primes ≤ 100)
<i>sieve.next.next.prime</i>	(returns the third prime)

10.5 Operational Semantics

The semantics of our imperative calculi is based exclusively on reduction relations; we do not define equational theories.

We give an operational semantics that relates terms to results in stores. The operational semantics is based on a global store. We say that a term b reduces to a result v to mean that, operationally, b yields v . Object terms reduce to object results consisting of sequences of store locations, one location for each object component. In order to stay close to standard implementation techniques, we avoid using formal substitutions during reduction. We describe a semantics based on stacks and closures. A stack associates variables with results; a closure is a pair of a method together with a stack that is used for the reduction of the method body. A store maps locations to method closures.

The operational semantics is expressed in terms of a relation that relates a store σ , a stack S , a term b , a result v , and another store σ' . This relation is written:

$$\sigma \cdot S \vdash b \rightsquigarrow v \cdot \sigma'$$

and it means that with the store σ and the stack S , the term b reduces to a result v , yielding an updated store σ' in the process; the stack does not change.

10.5.1 Reduction Rules

We now list the entities used in the operational semantics, the judgments that relate them to terms, and the reduction rules that manipulate them. We represent stacks and stores as finite sequences. In particular, the sequence $l_i \mapsto m_i \ i \in 1..n$ is the store that maps the location l_i to the closure m_i , for $i \in 1..n$. We let $\sigma \cdot l_i \leftarrow m$ denote the result of storing m in the location l_i of σ , so if $\sigma \equiv l_i \mapsto m_i \ i \in 1..n$ and $j \in 1..n$ then $\sigma \cdot l_j \leftarrow m \equiv l_j \mapsto m, l_i \mapsto m_i \ i \in 1..n - \{j\}$.

Operational semantics

l	store location	(e.g., an integer)
$v ::= [l_i = l_i \ i \in 1..n]$	result	(l_i distinct)
$\sigma ::= l_i \mapsto (\zeta(x_i)b_i, S_i) \ i \in 1..n$	store	(l_i distinct)
$S ::= x_i \mapsto v_i \ i \in 1..n$	stack	(x_i distinct)
$\sigma \vdash \diamond$	well-formed store judgment	
$\sigma \cdot S \vdash \diamond$	well-formed stack judgment	
$\sigma \cdot S \vdash a \rightsquigarrow v \cdot \sigma'$	term reduction judgment	

(Store \emptyset)	(Store l)
	$\sigma \cdot S \vdash \diamond \quad l \notin \text{dom}(\sigma)$
$\emptyset \vdash \diamond$	$\sigma, l \mapsto (\zeta(x)b, S) \vdash \diamond$

(Stack \emptyset)	(Stack x)	(l_i, l_i distinct)
$\sigma \vdash \diamond$	$\sigma \cdot S \vdash \diamond \quad x \notin \text{dom}(S) \quad \forall i \in 1..n$	
$\sigma \cdot \emptyset \vdash \diamond$	$\sigma \cdot (S, x \mapsto [l_i = l_i \ i \in 1..n]) \vdash \diamond$	

(Red x)

$$\frac{\sigma \cdot (S', x \mapsto v, S'') \vdash \diamond}{\sigma \cdot (S', x \mapsto v, S'') \vdash x \rightsquigarrow v \cdot \sigma}$$

(Red Object) (l_i, l_i distinct)

$$\frac{\sigma \cdot S \vdash \diamond \quad l_i \notin \text{dom}(\sigma) \quad \forall i \in 1..n}{\sigma \cdot S \vdash [l_i = \zeta(x_i)b_i]^{i \in 1..n} \rightsquigarrow [l_i = l_i]^{i \in 1..n} \cdot (\sigma, l_i \mapsto (\zeta(x_i)b_i, S)^{i \in 1..n})}$$

(Red Select)

$$\frac{\sigma \cdot S \vdash a \rightsquigarrow [l_i = l_i]^{i \in 1..n} \cdot \sigma' \quad \sigma'(l_j) = (\zeta(x_j)b_j, S') \quad x_j \notin \text{dom}(S') \quad j \in 1..n}{\sigma' \cdot (S', x_j \mapsto [l_i = l_i]^{i \in 1..n}) \vdash b_j \rightsquigarrow v \cdot \sigma''}$$

$$\sigma \cdot S \vdash a.l_j \rightsquigarrow v \cdot \sigma''$$

(Red Update)

$$\frac{\sigma \cdot S \vdash a \rightsquigarrow [l_i = l_i]^{i \in 1..n} \cdot \sigma' \quad j \in 1..n \quad l_i \in \text{dom}(\sigma')}{\sigma \cdot S \vdash a.l_j \leftarrow \zeta(x)b \rightsquigarrow [l_i = l_i]^{i \in 1..n} \cdot (\sigma'.l_j \leftarrow (\zeta(x)b, S))}$$

(Red Clone) (l_i' distinct)

$$\frac{\sigma \cdot S \vdash a \rightsquigarrow [l_i = l_i]^{i \in 1..n} \cdot \sigma' \quad l_i \in \text{dom}(\sigma') \quad l_i' \notin \text{dom}(\sigma') \quad \forall i \in 1..n}{\sigma \cdot S \vdash \text{clone}(a) \rightsquigarrow [l_i = l_i']^{i \in 1..n} \cdot (\sigma', l_i' \mapsto \sigma'(l_i)^{i \in 1..n})}$$

(Red Let)

$$\frac{\sigma \cdot S \vdash a \rightsquigarrow v' \cdot \sigma' \quad \sigma' \cdot (S, x \mapsto v') \vdash b \rightsquigarrow v'' \cdot \sigma''}{\sigma \cdot S \vdash \text{let } x=a \text{ in } b \rightsquigarrow v'' \cdot \sigma''}$$

The term reduction rules form the core of the semantics. A variable reduces to the result it denotes in the current stack. An object reduces to a result consisting of a fresh collection of locations; the store is extended to associate method closures to those locations. A selection operation reduces its object to a result, and activates the appropriate method closure. An update operation reduces its object to a result, and updates the appropriate store location with a new method closure. A cloning operation reduces its object to a result; then it allocates a fresh collection of locations that are associated to the existing method closures from the object. Finally, a *let* construct reduces to the result of reducing its body in a stack extended with the bound variable and the result of its associated term.

An algorithm for reduction can easily be extracted from these rules; it parallels standard implementations of objects.

10.5.2 Examples of Reductions

To illustrate the behavior of imperative terms, we detail some reductions using the notation of Section 7.1.

The first example is a simple terminating reduction.

$$\begin{array}{lcl} \emptyset \cdot \emptyset \vdash [l = \zeta(x)[]] \rightsquigarrow [l = 0] \cdot (0 \rightarrow (\zeta(x)[], \emptyset)) & & \text{by (Red Object)} \\ \downarrow \\ ((0 \rightarrow (\zeta(x)[], \emptyset)) \cdot (x \mapsto [l = 0])) \vdash [] \rightsquigarrow [] \cdot (0 \rightarrow (\zeta(x)[], \emptyset)) & & \text{by (Red Object)} \\ \emptyset \cdot \emptyset \vdash [l = \zeta(x)[]].l \rightsquigarrow [] \cdot (0 \rightarrow (\zeta(x)[], \emptyset)) & & \text{(Red Select)} \end{array}$$

The next one is a divergent reduction; we show the incomplete derivation tree that is built in the attempt to obtain a judgment of the form $\emptyset \cdot \emptyset \vdash [l = \zeta(x)x.l].l \rightsquigarrow ?\cdot?$. The tree displays a repeating pattern on an infinite branch.

$$\begin{array}{lcl} \emptyset \cdot \emptyset \vdash [l = \zeta(x)x.l] \rightsquigarrow [l = 0] \cdot (0 \rightarrow (\zeta(x)x.l, \emptyset)) & & \text{by (Red Object)} \\ \downarrow \\ ((0 \rightarrow (\zeta(x)x.l, \emptyset)) \cdot (x \mapsto [l = 0])) \vdash x \rightsquigarrow [l = 0] \cdot (0 \rightarrow (\zeta(x)x.l, \emptyset)) & & \text{by (Red } x\text{)} \\ \downarrow \\ \dots & & \dots \\ \downarrow \\ ((0 \rightarrow (\zeta(x)x.l, \emptyset)) \cdot (x \mapsto [l = 0])) \vdash x.l \rightsquigarrow ?\cdot? & & \text{by (Red Select)} \\ \downarrow \\ ((0 \rightarrow (\zeta(x)x.l, \emptyset)) \cdot (x \mapsto [l = 0])) \vdash x.l \rightsquigarrow ?\cdot? & & \text{(Red Select)} \\ \emptyset \cdot \emptyset \vdash [l = \zeta(x)x.l].l \rightsquigarrow ?\cdot? & & \text{(Red Select)} \end{array}$$

As a variation of this example, we can have a divergent reduction that keeps allocating storage. Read from the bottom up, the tree for this reduction displays judgments with increasingly large stores, $\sigma_0, \sigma_1, \dots$:

$$\begin{array}{lcl} \sigma_0 \triangleq 0 \rightarrow (\zeta(x)\text{clone}(x).l, \emptyset) & & \\ \sigma_1 \triangleq \sigma_0, 1 \rightarrow (\zeta(x)\text{clone}(x).l, \emptyset) & & \\ \emptyset \cdot \emptyset \vdash [l = \zeta(x)\text{clone}(x).l] \rightsquigarrow [l = 0] \cdot \sigma_0 & & \text{by (Red Object)} \\ \downarrow \\ \left(\begin{array}{l} \sigma_0 \cdot (x \mapsto [l = 0]) \vdash x \rightsquigarrow [l = 0] \cdot \sigma_0 \\ \sigma_0 \cdot (x \mapsto [l = 0]) \vdash \text{clone}(x) \rightsquigarrow [l = 1] \cdot \sigma_1 \end{array} \right) & & \begin{array}{l} \text{by (Red } x\text{)} \\ \text{(Red Clone)} \end{array} \\ \downarrow \\ \dots & & \dots \\ \downarrow \\ \sigma_1 \cdot (x \mapsto [l = 0]) \vdash \text{clone}(x).l \rightsquigarrow ?\cdot? & & \text{by (Red Select)} \\ \downarrow \\ \sigma_0 \cdot (x \mapsto [l = 0]) \vdash \text{clone}(x).l \rightsquigarrow ?\cdot? & & \text{(Red Select)} \\ \emptyset \cdot \emptyset \vdash [l = \zeta(x)\text{clone}(x).l].l \rightsquigarrow ?\cdot? & & \text{(Red Select)} \end{array}$$

Another sort of incomplete derivation tree arises from dynamic error conditions. In the next example the error consists in attempting to invoke a method from an object that does not have it.

$$\begin{array}{lcl} \emptyset \cdot \emptyset \vdash [] \rightsquigarrow [] \cdot \emptyset & & \text{by (Red Object)} \\ \downarrow \\ \emptyset \cdot \emptyset \vdash [].l \rightsquigarrow ?\cdot? & & \text{STUCK} \\ & & \text{(Red Select)} \end{array}$$

The final example illustrates method updating, and creating loops in the store:

$$\begin{array}{lcl} \sigma_0 \triangleq 0 \rightarrow (\zeta(x)x.l \Leftarrow \zeta(y)x, \emptyset) & & \\ \sigma_1 \triangleq 0 \rightarrow (\zeta(y)x, (x \mapsto [l = 0])) & & \end{array}$$

$$\begin{array}{c}
 \frac{\emptyset \cdot \emptyset \vdash [l = \zeta(x)x.l \Leftarrow \zeta(y)x] \rightsquigarrow [l = 0] \cdot \sigma_0}{\sigma_0 \cdot (x \mapsto [l = 0]) \vdash x \rightsquigarrow [l = 0] \cdot \sigma_0} \text{ by (Red Object)} \\
 \downarrow \\
 \sigma_0 \cdot (x \mapsto [l = 0]) \vdash x.l \Leftarrow \zeta(y)x \rightsquigarrow [l = 0] \cdot \sigma_1 \text{ by (Red } x\text{)} \\
 \text{(Red Update)} \\
 \emptyset \cdot \emptyset \vdash [l = \zeta(x)x.l \Leftarrow \zeta(y)x].l \rightsquigarrow [l = 0] \cdot \sigma_1 \text{ (Red Select)}
 \end{array}$$

The store σ_1 contains a loop, because it maps the index 0 to a closure that binds the variable x to a value that contains index 0. Hence an attempt to read out the result of $[l = \zeta(x)x.l \Leftarrow \zeta(y)x].l$ by “Inlining” the store and stack mappings would produce the infinite term $[l = \zeta(y)[l = \zeta(y)[l = \zeta(y)\dots]]]$.

The potential for creating loops in the store is characteristic of imperative semantics. Loops in the store complicate reasoning about programs and, as we see in the next chapter, they also demand special attention in the treatment of type soundness.

11 FIRST-ORDER IMPERATIVE CALCULI

Continuing the study of imperative calculi, we present a first-order type system with subtyping for the untyped calculus imp_ζ of Chapter 10. The subject reduction proof for this system illustrates a basic proof technique for imperative operational semantics.

We give examples that demonstrate the relative simplicity of imperative typing: a method may store results in global locations, but its result type will not reflect this. When encoding classes, the imperative features enable side-effects at the class level and global changes to class instances.

Our main concern in this chapter is to treat imperative semantics without distractions or unnecessary complications. Therefore we keep the type system to a minimum: the only type constructor is the one for object types. Some of the type constructions we have already studied could be integrated into the type system. In Chapter 17 we study a more complete imperative calculus that, in particular, includes variance annotations for distinguishing between updatable and non-updatable components.

11.1 Typing

Instead of introducing typing fragments one by one, as we have done for functional calculi, we describe a whole typing system for imperative objects. Our typing rules for imperative objects are similar to the ones for functional objects. They are, in fact, a superset of the rules for objects in $\mathbf{Ob}_{1<}$, except that terms do not contain type annotations (to match our untyped operational semantics).

Typing rules

$(\text{Env } \emptyset)$	$(\text{Env } x)$	
		$E \vdash A \quad x \notin \text{dom}(E)$
$\frac{}{\emptyset \vdash \diamond}$	$\frac{}{E, x:A \vdash \diamond}$	
$(\text{Type Object}) \quad (l_i \text{ distinct})$		
$E \vdash B_i \quad \forall i \in 1..n$		
$\frac{}{E \vdash [l_i:B_i]^{i \in 1..n}}$		
$(\text{Sub Refl}) \quad E \vdash A$	$(\text{Sub Trans}) \quad E \vdash A <: B \quad E \vdash B <: C$	$(\text{Sub Object}) \quad (l_i \text{ distinct}) \quad E \vdash B_i \quad \forall i \in 1..n+m$
$\frac{}{E \vdash A <: A}$	$\frac{}{E \vdash A <: C}$	$\frac{}{E \vdash [l_i:B_i]^{i \in 1..n+m} <: [l_i:B_i]^{i \in 1..n}}$

(Val Subsumption)	(Val x)	(Val Object) (where $A \equiv [l_i; B_i]^{i \in 1..n}$)
$\frac{E \vdash a : A \quad E \vdash A <: B}{E \vdash a : B}$	$\frac{E', x:A, E'' \vdash \diamond}{E', x:A, E'' \vdash x : A}$	$\frac{E, x_i:A \vdash b_i : B_i \quad \forall i \in 1..n}{E \vdash [l_i = \zeta(x_i)b_i]^{i \in 1..n} : A}$
(Val Select)	(Val Update) (where $A \equiv [l_i; B_i]^{i \in 1..n}$)	
$E \vdash a : [l_j; B_i]^{i \in 1..n} \quad j \in 1..n$	$E \vdash a : A \quad E, x:A \vdash b : B_j \quad j \in 1..n$	$E \vdash a.l_j = \zeta(x)b : A$
(Val Clone) (where $A \equiv [l_i; B_i]^{i \in 1..n}$)	(Val Let)	
$E \vdash a : A$	$E \vdash a : A \quad E, x:A \vdash b : B$	$E \vdash \text{let } x=a \text{ in } b : B$
$E \vdash \text{clone}(a) : A$		

The rules (Env \emptyset) and (Env x) are used for building typing environments. The rules (Type Object), (Sub Refl), (Sub Trans), and (Sub Object) concern the formation of types and the subtype relation. An object type $[l_i; B_i]^{i \in 1..n}$ is a collection of method names l_i and associated method result types B_i . A longer object type is a subtype of a shorter one, without variation in the common type components. (This invariance is necessary for soundness, as in previous chapters.) The remaining rules concern the typing of values. There is one rule for each construct in the calculus; in addition, a subsumption rule connects the value typing and subtyping judgments.

The rules for cloning and let are new, but straightforward. The rule for method update is the same as in functional calculi; according to this rule, an update may modify an object whose complete set of methods is statically unknown.

11.2 Examples of typings

This section illustrates how to type some imperative examples.

We use $\text{imp}\lambda\zeta_f$ of Section 10.3 for writing terms. When we claim that a term has a type, we mean that its translation into $\text{imp}\zeta$ has the corresponding translated type in the type system of Section 11.1. A function type $A \rightarrow B$ is translated as an object type $[\text{arg}:A, \text{val}:B]$.

11.2.1 Movable Points

Trivial as it may seem, the example of movable points has been a notorious source of difficulties in functional settings (see Section 9.5). These difficulties have resulted in the use of sophisticated type theories. In an imperative setting, however, some of these difficulties can be avoided altogether.

Consider one-dimensional and two-dimensional points, with integer coordinate fields (x and y) and methods that modify these fields (mv_x and mv_y). The origin points are:

```

let p1 =
  [x = 0, mv_x =  $\zeta(s) \lambda(dx) s.x := s.x+dx$ ];
let p2 =
  [x = 0, y = 0,
   mv_x =  $\zeta(s) \lambda(dx) s.x := s.x+dx$ ,
   mv_y =  $\zeta(s) \lambda(dy) s.y := s.y+dy$ ];

```

In the type system of Section 11.1, p_1 and p_2 can be given the types:

$$\begin{aligned} P_1 &\triangleq [x:Int, mv_x:Int \rightarrow []] \\ P_2 &\triangleq [x,y:Int, mv_x,mv_y:Int \rightarrow []] \end{aligned}$$

where P_2 is a subtype of P_1 . This result type $[]$ is obtained by subsumption. The imperative operational semantics produces the desired effect of moving a point without requiring any particular result type for move methods. In contrast, an informative result type is necessary in a functional framework.

Imperatively, there is no loss of type information when moving a point. For example, suppose that f is defined with one-dimensional points in mind, with the type $P_1 \rightarrow []$, and that $norm_2$ is defined for two-dimensional points, with the type $P_2 \rightarrow Real$:

$$\begin{aligned} let f: P_1 \rightarrow [] &= \lambda(p) p.mv_x(1); \\ let norm_2: P_2 \rightarrow Real &= \lambda(p) sqrt(p.x^2 + p.y^2); \end{aligned}$$

Because P_2 is a subtype of P_1 , $f(p_2)$ is a legal call for $p_2 : P_2$. Therefore the following code typechecks and, as expected, returns 1:

$$f(p_2); norm_2(p_2)$$

Thus we have applied a procedure that expects a P_1 point to a P_2 point, and after this we are still able to use the point as a member of P_2 . In contrast, in a functional setting we may try to write $norm_2(f(p_2))$, which is not well-typed if $f: P_1 \rightarrow []$.

Even in an imperative setting, however, it is common to define methods that produce new objects, as opposed to modifying existing ones. If mv_x is to return a new object, one must declare it with result type P_1 or P_2 , to take advantage of any change to the x coordinate. One may try to redefine P_1 and P_2 as recursive types (for example, $P_1 \triangleq \mu(X)[x:Int, mv_x:Int \rightarrow X]$), but then P_2 is not a subtype of P_1 . With this definition, all the typing difficulties common in functional settings resurface.

11.2.2 A Calculator

We can write a typed imperative version of the calculator of Section 6.5.4, adding a method for clearing the state:

$$Calc \triangleq [arg, acc:Real, clear:[], enter:Real \rightarrow [], add, sub:[], equals:Real]$$

```

let calculator =
  [arg = 0.0,
  acc = 0.0,
  clear =  $\zeta(s)$  s.arg := 0.0; s.acc := 0.0; s.equals  $\neq \zeta(s)$  s.arg,
  enter =  $\zeta(s)$   $\lambda(n)$ .s.arg := n,
  add =  $\zeta(s)$  s.acc := s.equals; s.equals  $\neq \zeta(s')$  s'.acc+s'.arg,
  sub =  $\zeta(s)$  s.acc := s.equals; s.equals  $\neq \zeta(s')$  s'.acc-s'.arg,
  equals =  $\zeta(s)$  s.arg];

```

For example:

```
calculator.enter(5.0); calculator.add; calculator.equals = 10.0
```

As in the example of movable points, we do not need recursive types because of the imperative semantics.

11.3 Classes and Global Behavior Change

Our treatment of traits and classes carries over to imperative calculi, with some interesting twists.

11.3.1 Classes and Update

Let us consider an appropriate variant of the classes c and c' of Section 6.6.1:

```

let c =
  [new= $\zeta(z)$ ][ $l_i=\zeta(s)z.l_i(s)$   $i\in 1..n$ ],
   $l_i=\zeta(z)\lambda(s)b_i$   $i\in 1..n$ ];

let c' =
  [new= $\zeta(z)$ ][ $l_i=\zeta(s)z.l_i(s)$   $i\in 1..n+m$ ],
   $l_j=\zeta(z)c.l_j$   $j\in 1..n$ ,
   $l_k=\zeta(z)\lambda(s)b_k$   $k\in n+1..n+m$ ];

```

The class c evaluates to a set of locations $[new=l_0, l_i=l_i \ i\in 1..n]$ pointing to closures for new and the pre-methods l_i . When $c.new$ is invoked, a set of locations is allocated for the new object, containing closures for its methods. These closures contain the code $\zeta(s)z.l_i(s)$, where z is bound to $[new=l_0, l_i=l_i \ i\in 1..n]$. When a method of the new object is invoked, the corresponding pre-method is fetched from the class and applied to the object.

When the pre-method l_j is inherited from c to c' , the evaluation of $c.l_j$ is suspended by $\zeta(z)$. Therefore, whenever the method l_j is invoked on an instance of c' , the pre-method l_j is fetched from c . The binders $\zeta(z)$ are introduced uniformly in order to suspend evaluation and to achieve this dynamic lookup of pre-methods inherited from c . When $c'.new$ is invoked, the methods of the new object refer to c' and, indirectly, to c .

Suppose that, after c and c' have been created, and after instances of c and c' have been allocated, we replace the pre-method l_1 of c using a side-effect. The instances of c reflect the change, because each method invocation goes back to c to fetch the pre-method. The instances of c' also reflect the change, via the indirection through c' .

Therefore the effect of replacing a pre-method in a class is, by default, to modify the behavior of all the instances of the class and of classes that inherited the pre-method. This default is inhibited by independent updates to objects and to inheriting classes.

Our definition of classes is tailored to obtain this global-change effect. If one is not interested in global change, one can optimize the definition and remove some of the run-time indirections. In particular, we can replace the proper method $l_j = \zeta(z).c.l_j$ in the subclass c' with a field $l_j = c.l_j$, so that a change to $c.l_j$ after the definition of c' will not affect c' or instances of c' . Similarly, we can make $c.\text{new}$ evaluate the pre-methods of c , so that a change to c will not affect existing instances.

Combining these techniques, we obtain the following eager variants e and e' of c and c' :

```
let e =
  [new =  $\zeta(z)$  let  $w_1 = z.l_1$  in ... let  $w_n = z.l_n$  in [ $l_i = \zeta(s)w_i(s)$   $i \in 1..n$ ],
   $l_i = \zeta(z)\lambda(s)b_i$   $i \in 1..n$ ];
let e' =
  [new =  $\zeta(z)$  let  $w_1 = z.l_1$  in ... let  $w_{n+m} = z.l_{n+m}$  in [ $l_i = \zeta(s)w_i(s)$   $i \in 1..n+m$ ],
   $l_j = e.l_j$   $j \in 1..n$ ,
   $l_k = \zeta(z)\lambda(s)b_k$   $k \in n+1..n+m$ ];
```

In all cases, the typing of classes works just like in functional calculi (see Section 8.5), using variant function types. The typing of traits also works similarly, resulting in a type-safe way to perform global change in object-based languages.

11.3.2 Examples

As an example, we reconsider the movable points of Section 11.2.1. Corresponding to the object types P_1 and P_2 we have the class types:

```
Class( $P_1$ )  $\equiv$  [new: $P_1$ ,  $x:P_1 \rightarrow \text{Int}$ ,  $mv\_x:P_1 \rightarrow \text{Int} \rightarrow []$ ]
Class( $P_2$ )  $\equiv$  [new: $P_2$ ,  $x,y:P_2 \rightarrow \text{Int}$ ,  $mv\_x, mv\_y:P_2 \rightarrow \text{Int} \rightarrow []$ ]
```

We can construct a class cp_1 for P_1 in the standard manner; because P_2 is a subtype of P_1 , a class cp_2 for P_2 may inherit pre-methods from cp_1 :

```
let  $cp_1 : Class(P_1) =$ 
  [new =  $\zeta(z)[...]$ ,
   $x = \zeta(z)\lambda(s)0$ ,
   $mv\_x = \zeta(z)\lambda(s)\lambda(dx)s.x := s.x + dx$ ];
```

```

let cp2 : Class(P2) =
  [new = ζ(z)[...],
   x = ζ(z) cp1.x,
   y = ζ(z) λ(s) 0,
   mv_x = ζ(z) cp1.mv_x,
   mv_y = ζ(z) λ(s) λ(dy) s.y := s.y+dy]

```

We define points p_1 and p_2 by generating them from the classes cp_1 and cp_2 :

```

let p1 = cp1.new;
let p2 = cp2.new;

```

Now we may update the class cp_1 ; we change the mv_x pre-method so that it does not set the x coordinate of a point to a negative number:

```
cp1.mv_x = ζ(z) λ(s) λ(dx) s.x := max(s.x+dx, 0)
```

The update is seen by p_1 because p_1 was generated from cp_1 ; the update is seen also by p_2 because p_2 was generated from cp_2 which inherited mv_x from cp_1 :

```

p1.mv_x(-3).x = 0
p2.mv_x(-3).x = 0

```

11.4 Subject Reduction

We show that the operational semantics of Section 10.5 is consistent with the type system of Section 11.1. We use an approach based on subject reduction, adapting the techniques recently developed by Tofte, Wright and Felleisen, Leroy, and Harper [69, 77, 120, 128]. This approach yields manageable proofs and deals with typing rules that seem hard to justify with other approaches (e.g., denotational semantics).

11.4.1 typings with Stores

In order to prove a subject reduction theorem, we need to be able to give types to results. Unfortunately, the typing of results is delicate. We would not be able to determine the type of a result by examining its substructures recursively, including the ones accessed through the store, because stores may contain loops. Store types, introduced next, allow us to type results independently of particular stores. This is possible because type-sound computations do not store results of different types in the same location. Next we formalize store types and other notions necessary for the proof of subject reduction.

A store type Σ associates a method type to each store location. A method type has the form $[l; B_i]_{i \in 1..n} \Rightarrow B_j$, where $[l; B_i]_{i \in 1..n}$ is the type of self, and B_j is the result type, for $j \in 1..n$. If $\Sigma(l) = A \Rightarrow B$ is the method type associated with l in Σ , then we define $\Sigma_1(l) \triangleq A$ and $\Sigma_2(l) \triangleq B$.

The statement of subject reduction relies on a new judgment, result typing:

$$\Sigma \models v : A$$

This means that the result v has type A with respect to the store type Σ . The locations contained in v are assigned types in Σ .

Since in the operational semantics a result is interpreted in a store, we need to connect stores and store types; we use a judgment to express that a store is compatible with a store type:

$$\Sigma \models \sigma$$

Checking this judgment reduces to checking that the contents of every store location has the type determined by the store type for that location. Since locations contain closures, we need to determine when a closure has a method type. For this, it is sufficient to check that a stack is compatible with an environment; the environment is then used to type the method. We write:

$$\Sigma \models S : E$$

to mean that the stack S is compatible with the environment E in Σ .

Now, since stacks contain results and environments contain types, we can define $\Sigma \models S : E$ via the result typing judgment, which we have already discussed.

Store typing

$M ::= [l_i; B_i]^{i \in 1..n} \Rightarrow B_j$	method type	$(j \in 1..n)$
$\Sigma ::= \downarrow_l \rightarrow M_i^{i \in 1..n}$	store type	$(l_i \text{ distinct})$
$\Sigma_1(l) \triangleq [l_i; B_i]^{i \in 1..n} \Rightarrow B_j$	if $\Sigma(l) = [l_i; B_i]^{i \in 1..n} \Rightarrow B_j$	
$\Sigma_2(l) \triangleq B_j$	if $\Sigma(l) = [l_i; B_i]^{i \in 1..n} \Rightarrow B_j$	
$\vdash M \in \text{Meth}$	well-formed method type judgment	
$\Sigma \models \diamond$	well-formed store type judgment	
$\Sigma \models v : A$	result typing judgment	
$\Sigma \models S : E$	stack typing judgment	
$\Sigma \models \sigma$	store typing judgment	

(Method Type)

$$j \in 1..n$$

$$\vdash [l_i; B_i]^{i \in 1..n} \Rightarrow B_j \in \text{Meth}$$

(Store Type) $(l_i \text{ distinct})$

$$\vdash M_i \in \text{Meth} \quad \forall i \in 1..n$$

$$\downarrow_l \rightarrow M_i^{i \in 1..n} \models \diamond$$

(Result Object)

$$\frac{\Sigma \models \diamond \quad \Sigma_1(\mathbf{i}_i) \equiv [l_i; \Sigma_2(\mathbf{i}_i)]^{i \in 1..n} \quad \forall i \in 1..n}{\Sigma \models [l_i = \mathbf{i}_i]^{i \in 1..n} : [l_i; \Sigma_2(\mathbf{i}_i)]^{i \in 1..n}}$$

(Stack \emptyset Typing)

$$\frac{\Sigma \models \diamond}{\Sigma \models \emptyset : \emptyset}$$

(Stack x Typing)

$$\frac{\Sigma \models S : E \quad \Sigma \models v : A \quad x \notin \text{dom}(E)}{\Sigma \models S, x \mapsto v : E, x : A}$$

(Store Typing)

$$\frac{\Sigma \models S_i : E_i \quad E_i, x_i : \Sigma_1(\mathbf{i}_i) \vdash b_i : \Sigma_2(\mathbf{i}_i) \quad \forall i \in 1..n}{\Sigma \models \mathbf{i}_i \mapsto (\zeta(x_i)b_i, S_i)^{i \in 1..n}}$$

Note that the (Store Typing) rule types each closure with respect to the whole store, accounting for cycles in the store.

11.4.2 Proof of Subject Reduction

We state two lemmas, and then state and prove the subject reduction theorem. Note that, because of our use of stacks, there is no need for a substitution lemma.

We say that Σ' is an extension of Σ (and write $\Sigma' \succcurlyeq \Sigma$) iff $\text{dom}(\Sigma) \subseteq \text{dom}(\Sigma')$ and for all $\mathbf{i} \in \text{dom}(\Sigma)$, $\Sigma'(\mathbf{i}) = \Sigma(\mathbf{i})$.

Lemma 11.4-1

If $\Sigma \models S : E$ and $\Sigma' \models \diamond$ with $\Sigma' \succcurlyeq \Sigma$, then $\Sigma' \models S : E$.

□

Lemma 11.4-2 (Bound weakening)

If $E, x:D, E' \vdash \mathfrak{S}$ and $E \vdash D' <: D$, then $E, x:D', E' \vdash \mathfrak{S}$.

□

The subject reduction theorem is:

Theorem 11.4-3 (Subject reduction)

If $E \vdash a : A \wedge \sigma.S \vdash a \rightsquigarrow v.\sigma^+$ and $\Sigma \models \sigma \wedge \text{dom}(\sigma) = \text{dom}(\Sigma) \wedge \Sigma \models S : E$, then there exist a type A^+ and a store type Σ^+ such that $\Sigma^+ \succcurlyeq \Sigma \wedge \Sigma^+ \models \sigma^+ \wedge \text{dom}(\sigma^+) = \text{dom}(\Sigma^+) \wedge \Sigma^+ \models v : A^+ \wedge A^+ <: A$.

Proof

We proceed by induction on the derivation of $\sigma.S \vdash a \rightsquigarrow v.\sigma^+$.

Case (Red x)

$$\frac{\sigma \cdot (S', x \mapsto [l_i = l_i^{i \in 1..n}], S'') \vdash \diamond}{\sigma \cdot (S', x \mapsto [l_i = l_i^{i \in 1..n}], S'') \vdash x \rightsquigarrow [l_i = l_i^{i \in 1..n}] \cdot \sigma}$$

By hypothesis $E \vdash x : A \wedge \Sigma \models \sigma \wedge \text{dom}(\sigma) = \text{dom}(\Sigma) \wedge \Sigma \models S', x \mapsto [l_i = l_i^{i \in 1..n}], S'' : E$. Because of $E \vdash x : A$, we must have $E \equiv E', x : A^\dagger, E''$ for some $A^\dagger <: A$. Now $\Sigma \models S', x \mapsto [l_i = l_i^{i \in 1..n}], S'' : E$ must have been derived via several applications of (Stack x Typing) from, among others, $\Sigma \models [l_i = l_i^{i \in 1..n}] : A^\dagger$.

Take $\Sigma^\dagger \equiv \Sigma$. We conclude $\Sigma^\dagger \models \sigma \wedge \text{dom}(\sigma) = \text{dom}(\Sigma^\dagger) \wedge \Sigma^\dagger \models [l_i = l_i^{i \in 1..n}] : A^\dagger \wedge A^\dagger <: A$.

Case (Red Object) (l_i, ι_i distinct)

$$\frac{\sigma \cdot S \vdash \diamond \quad \iota_i \notin \text{dom}(\sigma) \quad \forall i \in 1..n}{\sigma \cdot S \vdash [l_i = \zeta(x_i)b_i]^{i \in 1..n} \rightsquigarrow [l_i = l_i^{i \in 1..n}] \cdot (\sigma, \iota_i \mapsto (\zeta(x_i)b_i, S)^{i \in 1..n})}$$

By hypothesis $E \vdash [l_i = \zeta(x_i)b_i]^{i \in 1..n} : A \wedge \Sigma \models \sigma \wedge \text{dom}(\sigma) = \text{dom}(\Sigma) \wedge \Sigma \models S : E$. Because of $E \vdash [l_i = \zeta(x_i)b_i]^{i \in 1..n} : A$, we must have $E \vdash [l_i = \zeta(x_i)b_i]^{i \in 1..n} : [l_i : B_i]^{i \in 1..n}$ by (Val Object), for some $[l_i : B_i]^{i \in 1..n} <: A$. Take $A^\dagger \equiv [l_i : B_i]^{i \in 1..n}$.

Take $\Sigma^\dagger \equiv \Sigma$, $\iota_j \mapsto (A^\dagger \Rightarrow B_j)^{j \in 1..n}$; by (Store Type) we have $\Sigma^\dagger \models \diamond$, because $\iota_i \notin \text{dom}(\sigma)$, and hence $\iota_i \notin \text{dom}(\Sigma)$, and because $\models A^\dagger \Rightarrow B_j \in \text{Meth}$ for $j \in 1..n$.

(1) Since Σ^\dagger is an extension of Σ , by Lemma 11.4-1 we also have $\Sigma^\dagger \models S : E$. Since $E \vdash [l_i = \zeta(x_i)b_i]^{i \in 1..n} : A^\dagger$, we must have $E, x_i : A^\dagger \vdash b_i : B_i$, that is, $E, x_i : \Sigma_1^\dagger(\iota_i) \vdash b_i : \Sigma_2^\dagger(\iota_i)$.

(2) We have that σ is of the form $\varepsilon_k \mapsto (\zeta(x_k)b_k, S_k)^{k \in 1..m}$. Now $\Sigma \models \sigma$ must come from the (Store Typing) rule, with $\Sigma \models S_k : E_k$ and $E_k, x_k : \Sigma_1(\varepsilon_k) \vdash b_k : \Sigma_2(\varepsilon_k)$. By Lemma 11.4-1, $\Sigma^\dagger \models S_k : E_k$; moreover, $E_k, x_k : \Sigma_1^\dagger(\varepsilon_k) \vdash b_k : \Sigma_2^\dagger(\varepsilon_k)$, because $\Sigma^\dagger(\varepsilon_k) = \Sigma(\varepsilon_k)$ for $k \in 1..m$ since $\text{dom}(\sigma) = \text{dom}(\Sigma) = \{\varepsilon_k^{k \in 1..m}\}$ and Σ^\dagger extends Σ .

By (1) and (2), via the (Store Typing) rule, we have $\Sigma^\dagger \models (\sigma, \iota_i \mapsto (\zeta(x_i)b_i, S)^{i \in 1..n})$. Since $\Sigma^\dagger \models \diamond$ and $\Sigma^\dagger \equiv \Sigma$, $\iota_j \mapsto (A^\dagger \Rightarrow B_j)^{j \in 1..n}$, we have $\Sigma^\dagger \models [l_i = l_i^{i \in 1..n}] : A^\dagger$ by the (Result Object) rule.

We conclude $\Sigma^\dagger \geq \Sigma \wedge \Sigma^\dagger \models (\sigma, \iota_i \mapsto (\zeta(x_i)b_i, S)^{i \in 1..n}) \wedge \text{dom}(\sigma, \iota_i \mapsto (\zeta(x_i)b_i, S)^{i \in 1..n}) = \text{dom}(\Sigma^\dagger) \wedge \Sigma^\dagger \models [l_i = l_i^{i \in 1..n}] : A^\dagger \wedge A^\dagger <: A$.

Case (Red Select)

$$\frac{\sigma \cdot S \vdash a \rightsquigarrow [l_i = l_i^{i \in 1..n}] \cdot \sigma' \quad \sigma'(\iota_j) = (\zeta(x_j)b_j, S') \quad x_j \notin \text{dom}(S') \quad j \in 1..n}{\sigma \cdot (S', x_j \mapsto [l_i = l_i^{i \in 1..n}]) \vdash b_j \rightsquigarrow v \cdot \sigma''}$$

$$\sigma \cdot S \vdash a.l_j \rightsquigarrow v \cdot \sigma''$$

By hypothesis $E \vdash a.l_j : A \wedge \Sigma \models \sigma \wedge \text{dom}(\sigma) = \text{dom}(\Sigma) \wedge \Sigma \models S : E$. Since $E \vdash a.l_j : A$, we must have $E \vdash a : [l_j : B_j, \dots]$, for some $[l_j : B_j, \dots]$ with $B_j <: A$.

By induction hypothesis, since $E \vdash a : [l_j; B_j, \dots] \wedge \sigma \cdot S \vdash a \rightsquigarrow [l_i = l_i^{i \in 1..n}] \cdot \sigma' \wedge \Sigma \models \sigma \wedge \text{dom}(\sigma) = \text{dom}(\Sigma) \wedge \Sigma \models S : E$, there exist a type A' and a store type Σ' such that $\Sigma' \geq \Sigma \wedge \Sigma' \models \sigma' \wedge \text{dom}(\sigma') = \text{dom}(\Sigma') \wedge \Sigma' \models [l_i = l_i^{i \in 1..n}] : A' \wedge A' <: [l_j; B_j, \dots]$.

Since $\sigma'(l_j) = \langle \zeta(x_j) b_j, S' \rangle$, the judgment $\Sigma' \models \sigma'$ must come via (Store Typing) from $\Sigma' \models S' : E_j$ and $E_j, x_j : \Sigma'_1(l_j) \vdash b_j : \Sigma'_2(l_j)$ for some E_j . Since $\Sigma' \models [l_i = l_i^{i \in 1..n}] : A'$ must come from (Result Object), we have $A' \equiv [l_i; \Sigma'_2(l_i)]^{i \in 1..n} \equiv \Sigma'_1(l_i)$. Since $A' <: [l_j; B_j, \dots]$, we have $\Sigma'_2(l_j) \equiv B_j$. Then from $E_j, x_j : \Sigma'_1(l_j) \vdash b_j : \Sigma'_2(l_j)$ we obtain $E_j, x_j : A' \vdash b_j : B_j$. Moreover, we get $\Sigma' \models S', x_j \rightsquigarrow [l_i = l_i^{i \in 1..n}] : E_j, x_j : A'$ by the (Stack x Typing) rule.

Let $E' \equiv E_j, x_j : A'$. By induction hypothesis, since $E' \vdash b_j : B_j \wedge \sigma' \cdot (S', x_j \rightsquigarrow [l_i = l_i^{i \in 1..n}]) \vdash b_j \rightsquigarrow v \cdot \sigma'' \wedge \Sigma' \models \sigma' \wedge \text{dom}(\sigma') = \text{dom}(\Sigma') \wedge \Sigma' \models S', x_j \rightsquigarrow [l_i = l_i^{i \in 1..n}] : E'$, there exist A^\dagger and Σ^\dagger such that $\Sigma^\dagger \geq \Sigma' \wedge \Sigma^\dagger \models \sigma'' \wedge \text{dom}(\sigma'') = \text{dom}(\Sigma^\dagger) \wedge \Sigma^\dagger \models v : A^\dagger \wedge A^\dagger <: B_j$.

We conclude:

- $\Sigma^\dagger \geq \Sigma$ by transitivity from $\Sigma^\dagger \geq \Sigma'$ and $\Sigma' \geq \Sigma$.
- $\Sigma^\dagger \models \sigma''$ with $\text{dom}(\sigma'') = \text{dom}(\Sigma^\dagger)$.
- $\Sigma^\dagger \models v : A^\dagger$ with $A^\dagger <: A$, by transitivity from $A^\dagger <: B_j$ and $B_j <: A$.

Case (Red Update)

$$\frac{\sigma \cdot S \vdash a \rightsquigarrow [l_i = l_i^{i \in 1..n}] \cdot \sigma' \quad j \in 1..n \quad l_j \in \text{dom}(\sigma')}{\sigma \cdot S \vdash a, l_j \neq \zeta(x) b \rightsquigarrow [l_i = l_i^{i \in 1..n}] \cdot (\sigma' \cdot l_j \leftarrow \langle \zeta(x) b, S \rangle)}$$

By hypothesis $E \vdash a, l_j \neq \zeta(x) b : A \wedge \Sigma \models \sigma \wedge \text{dom}(\sigma) = \text{dom}(\Sigma) \wedge \Sigma \models S : E$. Since $E \vdash a, l_j \neq \zeta(x) b : A$, we must have $E \vdash a : [l_j; B_j, \dots]$ and $E, x : [l_j; B_j, \dots] \vdash b : B_j$ for some $[l_j; B_j, \dots] <: A$.

By induction hypothesis, since $E \vdash a : [l_j; B_j, \dots] \wedge \sigma \cdot S \vdash a \rightsquigarrow [l_i = l_i^{i \in 1..n}] \cdot \sigma' \wedge \Sigma \models \sigma \wedge \text{dom}(\sigma) = \text{dom}(\Sigma) \wedge \Sigma \models S : E$, there exist a type A^\dagger and a store type Σ^\dagger such that $\Sigma^\dagger \geq \Sigma \wedge \Sigma^\dagger \models \sigma' \wedge \text{dom}(\sigma') = \text{dom}(\Sigma^\dagger) \wedge \Sigma^\dagger \models [l_i = l_i^{i \in 1..n}] : A^\dagger \wedge A^\dagger <: [l_j; B_j, \dots]$.

By assumption $l_j \in \text{dom}(\sigma')$, hence $l_j \in \text{dom}(\Sigma^\dagger)$. But $\Sigma^\dagger \models [l_i = l_i^{i \in 1..n}] : A^\dagger$ must have been derived via (Result Object) from $\Sigma_2^\dagger(l_j) \equiv [l_i; \Sigma_2^\dagger(l_i)]^{i \in 1..n} \equiv A^\dagger$ for all $i \in 1..n$. Hence, since $A^\dagger <: [l_j; B_j, \dots]$, we have $\Sigma_2^\dagger(l_j) = B_j$.

(1) We have $\Sigma^\dagger \models S : E$ by Lemma 11.4-1. We also have $E, x : A^\dagger \vdash b : B_j$ by Lemma 11.4-2 (from $E, x : [l_j; B_j, \dots] \vdash b : B_j$ and $A^\dagger <: [l_j; B_j, \dots]$), that is, $E, x : \Sigma_1^\dagger(l_j) \vdash b : \Sigma_2^\dagger(l_j)$.

(2) Since $\Sigma^\dagger \models \sigma'$ must come from (Store Typing), σ' is of the form $\varepsilon_k \rightsquigarrow \langle \zeta(x_k) b_k, S_k \rangle^{k \in 1..m}$, and for all k such that $\varepsilon_k \neq l_j$ and for some E_k we have $\Sigma^\dagger \models S_k : E_k$, and $E_k, x_k : \Sigma_1^\dagger(\varepsilon_k) \vdash b_k : \Sigma_2^\dagger(\varepsilon_k)$.

Take $\sigma^\dagger \equiv \sigma' \cdot l_j \leftarrow \langle \zeta(x) b, S \rangle$. Then by (1) and (2) we have $\Sigma^\dagger \models \sigma^\dagger$, by the (Store Typing) rule. Since $\text{dom}(\sigma') = \text{dom}(\sigma^\dagger)$, we have also $\text{dom}(\sigma^\dagger) = \text{dom}(\Sigma^\dagger)$.

We conclude $\Sigma^\dagger \geq \Sigma \wedge \Sigma^\dagger \models \sigma^\dagger \wedge \text{dom}(\sigma^\dagger) = \text{dom}(\Sigma^\dagger) \wedge \Sigma^\dagger \models [l_i = l_i^{i \in 1..n}] : A^\dagger \wedge A^\dagger <: A$, with the last conjunct derived from $A^\dagger <: [l_j; B_j, \dots]$ and $[l_j; B_j, \dots] <: A$ by transitivity.

Case (Red Clone) (ι_i' distinct)

$$\frac{\sigma \cdot S \vdash a \rightsquigarrow [l_i = \iota_i]^{i \in 1..n} \cdot \sigma' \quad \iota_i \in \text{dom}(\sigma') \quad \iota_i' \notin \text{dom}(\sigma') \quad \forall i \in 1..n}{\sigma \cdot S \vdash \text{clone}(a) \rightsquigarrow [l_i = \iota_i']^{i \in 1..n} \cdot (\sigma', \iota_i' \mapsto \sigma'(\iota_i))^{i \in 1..n}}$$

By hypothesis $E \vdash \text{clone}(a) : A \wedge \Sigma \models \sigma \wedge \text{dom}(\sigma) = \text{dom}(\Sigma) \wedge \Sigma \models S : E$. Since $E \vdash \text{clone}(a) : A$, we must have $E \vdash a : A$.

By induction hypothesis, since $E \vdash a : A \wedge \sigma \cdot S \vdash a \rightsquigarrow [l_i = \iota_i]^{i \in 1..n} \cdot \sigma' \wedge \Sigma \models \sigma \wedge \text{dom}(\sigma) = \text{dom}(\Sigma) \wedge \Sigma \models S : E$, there exist a type A^+ and a store type Σ' such that $\Sigma' \geq \Sigma \wedge \Sigma' \models \sigma' \wedge \text{dom}(\sigma') = \text{dom}(\Sigma') \wedge \Sigma' \models [l_i = \iota_i]^{i \in 1..n} : A^+ \wedge A^+ <: A$.

Let $\Sigma^+ \equiv (\Sigma', \iota_i' \mapsto \Sigma'(\iota_i))^{i \in 1..n}$ and $\sigma^+ \equiv (\sigma', \iota_i' \mapsto \sigma'(\iota_i))^{i \in 1..n}$. We have $\Sigma^+ \models \diamond$ by (Store Type) because $\iota_i' \notin \text{dom}(\sigma') = \text{dom}(\Sigma')$, ι_i' are all distinct for $i \in 1..n$, and $\Sigma' \models \diamond$ is a prerequisite of $\Sigma' \models \sigma'$.

We conclude:

- $A^+ <: A$.
- $\Sigma^+ \geq \Sigma$, because $\Sigma' \geq \Sigma$ and $\Sigma^+ \geq \Sigma'$.
- $\text{dom}(\sigma^+) = \text{dom}(\Sigma^+)$, by construction and $\text{dom}(\sigma') = \text{dom}(\Sigma')$.
- $\Sigma^+ \models \sigma^+$. Since $\Sigma' \models \sigma'$ must come from (Store Typing), σ' has the shape $\varepsilon_k \mapsto (\zeta(x_k)b_k, S_k)$ $k \in 1..m$, and for all $k \in 1..m$ and for some E_k we have $\Sigma' \models S_k : E_k$ and $E_k, x_k : \Sigma'_1(\varepsilon_k) \vdash b_k : \Sigma'_2(\varepsilon_k)$. Then we also have $E_k, x_k : \Sigma'_1(\varepsilon_k) \vdash b_k : \Sigma^+_2(\varepsilon_k)$, and by Lemma 11.4-1 $\Sigma^+ \models S_k : E_k$. Let $f : 1..n \rightarrow 1..m$ be ε^{-1}_σ , so that for all $i \in 1..n$, $\iota_i = \varepsilon_{f(i)}$. We have $E_{f(i)}, x_{f(i)} : \Sigma'_1(\varepsilon_{f(i)}) \vdash b_{f(i)} : \Sigma'_2(\varepsilon_{f(i)})$ for $i \in 1..n$, so $E_{f(i)}, x_{f(i)} : \Sigma'_1(\iota_i) \vdash b_{f(i)} : \Sigma^+_2(\iota_i)$. Moreover, since $\Sigma'(\iota_i) = \Sigma^+(\iota_i)$, we have $E_{f(i)}, x_{f(i)} : \Sigma^+_1(\iota_i) \vdash b_{f(i)} : \Sigma^+_2(\iota_i)$. The result follows by (Store Typing) from $\Sigma^+ \models S_k : E_k$ and $\Sigma^+ \models S_{f(i)} : E_{f(i)}$, and $E_k, x_k : \Sigma^+_1(\varepsilon_k) \vdash b_k : \Sigma^+_2(\varepsilon_k)$ and $E_{f(i)}, x_{f(i)} : \Sigma^+_1(\iota_i) \vdash b_{f(i)} : \Sigma^+_2(\iota_i)$, for $k \in 1..m$ and $i \in 1..n$.
- $\Sigma^+ \models [l_i = \iota_i]^{i \in 1..n} : A^+$. First, $\Sigma' \models [l_i = \iota_i]^{i \in 1..n} : A^+$ must come from the (Result Object) rule with $A^+ \equiv \Sigma'_1(\iota_i) \equiv [l_i : \Sigma'_2(\iota_i)]^{i \in 1..n}$ for $i \in 1..n$, and $\Sigma' \models \diamond$. But $\Sigma^+(\iota_i) \equiv \Sigma'(\iota_i)$ for $i \in 1..n$. So $\Sigma^+(\iota_i) \equiv [l_i : \Sigma^+_2(\iota_i)]^{i \in 1..n} \equiv A^+$, and we obtain $\Sigma^+ \models [l_i = \iota_i]^{i \in 1..n} : [l_i : \Sigma^+_2(\iota_i)]^{i \in 1..n}$ by (Result Object).

Case (Red Let)

$$\frac{\sigma \cdot S \vdash c \rightsquigarrow v' \cdot \sigma' \quad \sigma' \cdot S, x \mapsto v' \vdash b \rightsquigarrow v'' \cdot \sigma''}{\sigma \cdot S \vdash \text{let } x=c \text{ in } b \rightsquigarrow v'' \cdot \sigma''}$$

By hypothesis $E \vdash \text{let } x=c \text{ in } b : A \wedge \Sigma \models \sigma \wedge \text{dom}(\sigma) = \text{dom}(\Sigma) \wedge \Sigma \models S : E$. Since $E \vdash \text{let } x=c \text{ in } b : A$, we must have $E \vdash c : C$ for some C , and $E, x:C \vdash b : A$.

By induction hypothesis, since $E \vdash c : C \wedge \sigma \cdot S \vdash c \rightsquigarrow v' \cdot \sigma' \wedge \Sigma \models \sigma \wedge \text{dom}(\sigma) = \text{dom}(\Sigma) \wedge \Sigma \models S : E$, there exist a type C' and a store type Σ' such that $\Sigma' \geq \Sigma \wedge \Sigma' \models \sigma' \wedge \text{dom}(\sigma') = \text{dom}(\Sigma') \wedge \Sigma' \models v' : C' \wedge C' <: C$.

By Lemma 11.4-1, $\Sigma' \models S : E$, hence by (Stack x Typing) $\Sigma' \models S, x \mapsto v' : E, x : C'$. From $E, x : C \vdash b : A$, we obtain $E, x : C' \vdash b : A$ by Lemma 11.4-2.

By induction hypothesis, since $E, x : C' \vdash b : A \wedge \sigma' \cdot S, x \mapsto v' \vdash b \rightsquigarrow v'' \cdot \sigma'' \wedge \Sigma' \models \sigma' \wedge \text{dom}(\sigma') = \text{dom}(\Sigma') \wedge \Sigma' \models S, x \mapsto v' : E, x : C'$, there exist a type A^\dagger and a store type Σ^\dagger such that $\Sigma^\dagger \geq \Sigma' \wedge \Sigma^\dagger \models \sigma'' \wedge \text{dom}(\sigma'') = \text{dom}(\Sigma^\dagger) \wedge \Sigma^\dagger \models v'' : A^\dagger \wedge A^\dagger <: A$.

We conclude that $\Sigma^\dagger \geq \Sigma$ (by transitivity), $\Sigma^\dagger \models \sigma''$ with $\text{dom}(\sigma'') = \text{dom}(\Sigma^\dagger)$, and $\Sigma^\dagger \models v'' : A^\dagger$ with $A^\dagger <: A$.

□

Corollary 11.4-4

If $\emptyset \vdash a : A$ and $\emptyset \cdot \emptyset \vdash a \rightsquigarrow v \cdot \sigma$, then there exist a type A^\dagger and a store type Σ^\dagger such that $\Sigma^\dagger \models \sigma$ and $\Sigma^\dagger \models v : A^\dagger$, with $A^\dagger <: A$.

□

Therefore, if a term has a type and the term reduces to a result in a store, then the result can be assigned that type in that store. That is, if a term produces a result, it does so by respecting the type that it had been assigned statically.

As discussed in Section 7.5.2, this statement is vacuous if the term does not produce a result. This can happen either because reduction diverges, or because it gets stuck (no rule is applicable at a certain stage).

For each term there is at most one rule whose conclusion matches the syntactic form of the term, and hence is potentially applicable to the term. The rule (Red x) is applicable to x unless x is not defined in the stack. Assuming that b reduces to v , the rule (Red Update) is applicable to $b.l_j \not\models \zeta(x)c$ provided v has l_j . The applicability of the rule (Red Select) is determined with an analogous condition. Assuming the appropriate subterms converge, the rules (Red Object), (Red Clone), and (Red Let) are always applicable to terms of the corresponding forms.

Examining these cases, and adapting the proof of Theorem 11.4-3, we can deduce that the reduction of a well-typed term in a well-typed store cannot get stuck (although it may diverge). We omit a proof of this claim.

12 A FIRST-ORDER LANGUAGE

In Part I of this book, we have introduced a sequence of first-order object calculi of increasing sophistication. It may be time to assess the contributions that these calculi bring to the task of modeling programming language constructs. For this purpose, we invent and study a simple language named O-1.

We describe the syntax of O-1 and its typing rules; we argue that these rules are sound by showing how O-1 can be translated into a first-order object calculus. The object calculus in question, though never assembled explicitly, combines many of the notations and rules of the previous chapters. O-1 includes both class-based and object-based constructs, first-order object types with subtyping and variance annotations, recursion, and a typecase construct. The study of O-1 exemplifies how we can use object calculi for analyzing the constructs of object-oriented languages.

12.1 Features

O-1 includes a number of characteristic object-oriented constructs, similar to those of the pseudo-notation of the Review and to those of common object-oriented languages. It includes some more exotic constructs as well. These constructs are suggested by our calculi; they enrich O-1 without significant complications in the underlying theory.

In the terminology of the Review, O-1 includes both class-based and object-based features, demonstrating that we can model both kinds of features, and even combine them. In O-1, objects have embedded methods, which can be updated. Classes can be created by single inheritance from previously defined classes; methods of the super-classes may be overridden and specialized. Because of a distinction between classes and object types, O-1 separates interfaces from implementations, and inheritance from subtyping.

The type system of O-1, though, is rather restrictive in its treatment of the type of self. We work around this limitation by providing a typecase construct, in spite of the drawbacks discussed in Section 2.5.

O-1 is not a synthesis of all the notations and techniques of Part I. We have selected a set of features that are both interesting from a programming perspective, and sufficiently easy to explain in terms of object calculi. However, other useful features could be added without fundamental difficulty. For example, O-1 does not provide any mechanism for hiding the components of a class, but we could add such a mechanism following the treatment in Section 8.5. Eventually we could turn the O-1 core into a rich and practical language, not too different from conventional object-oriented languages.

12.2 Syntax

The following table gives the syntax of O-1 types:

Syntax of O-1 types

$A, B ::=$	types
X	type variable
Top	the biggest type
$\text{Object}(X)[l_i; v_i; B_i]^{i \in 1..n}$	object type (l_i distinct)
$\text{Class}(A)$	class type

O-1 includes both object types and class types; in addition, **Top** is the biggest type. For simplicity, O-1 does not include standard basic types (such as the type of booleans), or function types. Those types could be easily added or, as we have seen, they could be encoded.

An object type is written in the form $\text{Object}(X)[l_i; v_i; B_i]^{i \in 1..n}$, where X is a type variable, each l_i is a label, each v_i is a variance annotation ($-$, \circ , or $+$), and each $B_i[X]$ is a type. An object type may be defined recursively: the variable X stands for the whole object type. Each component $l_i; v_i; B_i[X]$ corresponds to an attribute, either a field or a proper method, with type $B_i[X]$. A subtype relation between object types supports subsumption; the precise rules for subtyping are given in Section 12.4.

The variance annotation $+$ identifies read-only attributes, while the variance annotation \circ identifies read-write attributes. Proper methods are usually read-only attributes and fields are usually read-write attributes, but this is not a requirement. We also include the variance annotation $-$ (write-only) because it is easy do so, but we do not use it in examples. As discussed in Chapter 8, variance annotations are useful for flexible subtyping and for protection, and also for certain encodings.

If A is an object type, then $\text{Class}(A)$ is a class type; it is the type of classes for generating objects of type A . Classes and object types are different entities, so we distinguish between subtyping and inheritance. Furthermore, object types and class types do not carry any implementation information, so we separate interfaces from implementations.

The syntax of terms is given below. We do not formalize the operational semantics of these terms; it could be either functional or imperative. Previous chapters have explored both kinds of semantics, and it is possible to adapt the techniques developed there to O-1. For an imperative semantics it is important to distinguish fields from proper methods, because different evaluation orders are appropriate. Rather than complicate the syntax and type rules of O-1 with this distinction, we assume that attribute names obey some convention that distinguishes field names from proper method names. With the assumption that a suitable convention exists, the syntax can be read either functionally or imperatively.

Syntax of O-1 terms

$a, b, c ::=$	terms
x	variable
object ($x:A$) $l_i=b_i \ i \in 1..n$ end	direct object construction
$a.l$	field selection / method invocation
$a.l := b$	update with a term
$a.l := \text{method}(x:A) b$ end	update with a method
new c	object construction from a class
root	root class
subclass of $c:C$ with ($x:A$) $l_i=b_i \ i \in n+1..n+m$	subclass
override $l_i=b_i \ i \in Ovr \subseteq 1..n$ end	additional attributes overridden attributes
$c^l(a)$	class selection
typecase a when ($x:A$) b_1 else b_2 end	typecase

As the syntax of terms shows, O-1 includes both object-based and class-based constructs. The two kinds of constructs coexist harmoniously. One could drop the object-based constructs (object construction and method update); the result would be a language expressive enough for traditional class-based programming. Alternatively, one could drop the class-based construct (root class, subclass, new, and class selection), obtaining an object-based language.

The construct for creating a single object is **object**($x:A$) $l_i=b_i\{x\} \ i \in 1..n$ **end**. The resulting object has attributes $l_i=b_i\{x\}$, where x is the self variable for all the attributes. Attributes can be fields or proper methods. The fields should not contain occurrences of the self variable.

The construct for invoking a method is $a.l$; as a special case, this is also the construct for field selection. There are two different constructs for update. The first one, $a.l := b$, updates a component to a field. The second one, $a.l := \text{method}(x:A) b$ **end**, updates a component to a method.

The base class is **root**. All other classes are defined as subclasses of **root**. Given a class c of a type C with methods named $l_i \ i \in 1..n$, the construct **subclass of** $c:C$ **with**($x:A$) $l_i=b_i\{x\} \ i \in n+1..n+m$ **override** $l_i=b_i\{x\} \ i \in Ovr \subseteq 1..n$ **end** introduces a new subclass. This subclass extends the class c with the methods named $l_i \ i \in n+1..n+m$ and overrides the methods named $l_i \ i \in Ovr$, where $Ovr \subseteq 1..n$. The variable x is the self variable for all the code; it is given the type A , which is the type of the objects generated from the subclass. Objects generated from other classes may have type A too; in this sense, types are independent from classes. However, if C is **Class**(A'), then A must be a subtype of A' ; thus subclasses correspond to subtypes.

Given a class c , the construct **new** c generates an object from c . The class selection construct $c^l(a)$ extracts the method named l from c and applies it to a . This latest con-

struct provides the functionality of super, since $c^l(x)$ may occur as part of the code for a subclass of c with self variable x .

Classes, as well as objects, are first-class values. For example, in O-1 an object may have a class as one of its fields and this field can be updated. A parametric class can be obtained simply as a function that returns a class.

Finally, O-1 includes a typecase construct. The evaluation of **typecase** a **when** $(x:A)b_1$ **else** b_2 **end** causes the evaluation of a to some value v ; if v has type A , then b_1 is evaluated with x bound to v ; if v does not have type A , then b_2 is evaluated instead. For the reasons explained in Chapter 9, and as shown in the examples of Section 12.3, the typecase construct plays a fairly important role in first-order languages such as O-1.

As an abbreviation, we sometimes omit the variance annotation $^\circ$. In addition, we have the following useful notations:

$$\begin{aligned}\mathbf{Root} &\triangleq \\ \mathbf{Class}(\mathbf{Object}(X)[]) &\\ \mathbf{class\ with}(x:A)\ l_i=b_i\ i\in 1..n\ \mathbf{end} &\triangleq \\ \quad \mathbf{subclass\ of\ root:Root\ with}(x:A)\ l_i=b_i\ i\in 1..n\ \mathbf{override\ end} &\\ \mathbf{subclass\ of\ c:C\ with\ (x:A)\ ...}\ \mathbf{super}.l\ \dots\ \mathbf{end} &\triangleq \\ \quad \mathbf{subclass\ of\ c:C\ with\ (x:A)\ ...}\ c^l(x)\ \dots\ \mathbf{end} &\\ \mathbf{object}(x:A)\ ... l\ \mathbf{copied\ from\ c}\ \dots\ \mathbf{end} &\triangleq \\ \quad \mathbf{object}(x:A)\ ... l=c^l(x)\ \dots\ \mathbf{end} &\end{aligned}$$

Root is the class type for the built-in class **root**, whose objects have type **Object(X)[]**. A subclass of **root** can be defined with the syntax **class ... end** without explicit mention of **root**. (We could take this construct as primitive, and then the **subclass** construct would be definable from it and class selection.) Given a class c , **super**. l stands for $c^l(x)$ in the scope of the definition of a subclass of c where x is the self variable. The notation **l copied from c** is similarly derived from $c^l(x)$: it can be used in object-based style for copying methods from classes directly into objects.

12.3 Examples

We write some small examples in O-1. These examples are meant to illustrate the notation, and to help explain the type rules that are defined formally in Section 12.4. The examples also illustrate the interplay between subtyping and typecase.

In order to write the examples, we assume basic types (*Bool*, *Int*) and function types ($A \rightarrow B$, contravariant in A and covariant in B). We use the notation **fun**($x:A$) b **end** for a function with argument x of type A and body b . Since we have not endowed O-1 with definition constructs, we use an informal top-level notation to this effect.

We begin by defining two object types:

$$\mathbf{Point} \triangleq \mathbf{Object}(X)[x: \mathbf{Int}, eq^+: X \rightarrow \mathbf{Bool}, mv^+: \mathbf{Int} \rightarrow X]$$

$$\mathbf{CPoint} \triangleq \mathbf{Object}(X)[x: \mathbf{Int}, c: \mathbf{Color}, eq^+: \mathbf{Point} \rightarrow \mathbf{Bool}, mv^+: \mathbf{Int} \rightarrow \mathbf{Point}]$$

The type *Point* has three components: an integer coordinate *x*; a binary method *eq*, which compares *self* with another point and returns a boolean; and a method *mv*, which takes an integer and returns a point. The type *CPoint* adds a color *c*. When the type variable *X* occurs in *Point*, it stands for *Point*; in *CPoint*, it stands for *CPoint*.

According to the rules of Section 12.4, the definitions yield *CPoint* $<:$ *Point*, because *CPoint* is longer than *Point*, and the occurrences of *X* in *Point* are matched with occurrences of *Point* in *CPoint*. Our examples show that subtyping such as this one are desirable in programming; many object-oriented languages allow them. The type of *mv* in *CPoint* is *Int* \rightarrow *Point*; later we explore the effect of changing it to *Int* \rightarrow *X*. The type *eq* in *CPoint* is *Point* \rightarrow *Bool*; if we were to change it to *X* \rightarrow *Bool* we would lose the subtyping *CPoint* $<:$ *Point*.

Corresponding to the two types *Point* and *CPoint*, we have two classes:

```

pointClass : Class(Point) 
  class with (self: Point)
    x = 0,
    eq = fun(other: Point) self.x = other.x end,
    mv = fun(dx: Int) self.x := self.x+dx end
  end

cPointClass : Class(CPoint) 
  subclass of pointClass: Class(Point)
  with (self: CPoint)
    c = black
  override
    eq = fun(other: Point)
      typecase other
        when (other': CPoint) super.eq(other') and self.c = other'.c
        else false
      end
    end
  end
end

```

The class *cPointClass* inherits *x* and *mv* from its superclass *pointClass*. Although it could inherit *eq* as well, *cPointClass* overrides this method as follows. The definition of *Point* requires that *eq* work with any argument *other* of type *Point*. In the *eq* code for *cPointClass*, the typecase on *other* determines whether *other* has a color. If so, *eq* works as in *pointClass* and in addition tests the color of *other*. If not, *eq* returns *false*.

We can use *cPointClass* to create color points of type *CPoint*:

```
cPoint : CPoint  new cPointClass
```

Calls to *mv* lose the color information. In order to access the color of a point after it has been moved, a typecase is necessary:

```
movedColor : Color  $\triangleq$ 
  typecase cPoint.mv(1)
  when (cp: CPoint) cp.c
  else black
  end
```

As this code fragment suggests, we may want to define a stronger type of color points that would preserve type information on move. We can do this by changing the type of *mv* to $\text{Int} \rightarrow X$, where *X* stands for the whole object type being defined. More precisely, we write:

```
CPoint2  $\triangleq$  Object(X)[x: Int, c: Color, eq+: Point  $\rightarrow$  Bool, mv+: Int  $\rightarrow$  X]
```

The subtyping $\text{CPoint2} <: \text{Point}$ follows from the rules of Section 12.4. The soundness of this subtyping is nontrivial and depends on the read-only annotation on *mv*.

Corresponding to this new object type, we have a new subclass of *pointClass*:

```
cPointClass2 : Class(CPoint2)  $\triangleq$ 
  subclass of pointClass: Class(Point)
  with (self: CPoint2)
    c = black
  override
    eq = fun(other: Point)
      typecase other
      when (other' : CPoint2) super.eq(other') and self.c = other'.c
      else false
      end
    end,
    mv = fun(dx: Int)
      typecase super.mv(dx)
      when (res: CPoint2) res
      else ... (error)
      end
    end
  end
```

This subclass overrides *eq* and *mv*. Again the override of *eq* is optional. Instead, the override of *mv* is now necessary. The *mv* method of *pointClass* does not have an appropriate type for *cPointClass2*: the type definitions say only that it has type $\text{Int} \rightarrow \text{Point}$ rather than $\text{Int} \rightarrow \text{CPoint2}$ as *cPointClass2* requires.

Quite incidentally, the code for *mv* in the superclass could be given the type $\text{Int} \rightarrow \text{CPoint2}$, but this fact cannot be known without looking at the implementation of *pointClass*. To obtain such knowledge, a compiler would have to violate the separation

of interfaces and implementations and would obstruct the separate compilation of classes.

In writing the overriding *mv* method, one option would be to make a textual copy of the code of *mv* into *cPointClass2*, so that it could be adequately typed in the new context. However, even without copying it, we can reuse that code by a call to *super.mv*, which invokes the *mv* method of the superclass. The result of this call needs to be a color point, and the necessary typing is achieved with a typecase.

In contrast to the code based on *CPoint*, typecase is no longer needed after a color point is moved. For example, we can write:

$$\begin{aligned} cPoint2 : CPoint2 &\triangleq \text{new } cPoint2Class \\ movedColor2 : Color &\triangleq cPoint2.mv(1).c \end{aligned}$$

In sum, by switching from *CPoint* to *CPoint2* we have shifted typecase from the code that uses color points to the code for the class *cPoint2Class*. This shift may be attractive, for example because it may help in localizing the use of typecase.

As this example illustrates, O–1 allows subtyping without inheritability: there are types *A* and *B* (namely, *CPoint2* and *Point*) such that *A* $<: B$, and such that a class for *A* can be constructed as a subclass of a class for *B*, but with the requirement that some methods be overridden in the subclass (namely, *mv*).

12.4 Typing

We formalize the type rules of O–1 in our usual format. The rules are based on the following judgments:

Judgments

$E \vdash \diamond$	environment <i>E</i> is well-formed
$E \vdash A$	<i>A</i> is a well-formed type in <i>E</i>
$E \vdash A <: B$	<i>A</i> is a subtype of <i>B</i> in <i>E</i>
$E \vdash vA <: v'B$	<i>A</i> is a subtype of <i>B</i> in <i>E</i> , with variance annotations <i>v</i> and <i>v'</i>
$E \vdash a : A$	<i>a</i> has type <i>A</i> in <i>E</i>

The rules for environments are standard:

Environments

(Env \emptyset)	(Env $X <:$)	(Env x)
	$E \vdash A \quad X \notin \text{dom}(E)$	$E \vdash A \quad x \notin \text{dom}(E)$
$\emptyset \vdash \diamond$	$E, X <: A \vdash \diamond$	
	$E, x : A \vdash \diamond$	

In the rules for types, the main innovation is that **Class(A)** is a type when *A* is an object type:

Types

$$\frac{\begin{array}{c} (\text{Type } X) \\ E', X <: A, E'' \vdash \diamond \end{array}}{E', X <: A, E'' \vdash X} \quad \frac{\begin{array}{c} (\text{Type Top}) \\ E \vdash \diamond \end{array}}{E \vdash \text{Top}}$$

$$\frac{\begin{array}{c} (\text{Type Object}) \quad (l_i \text{ distinct}, v_i \in \{^0, ^+\}) \\ E, X <: \text{Top} \vdash B_i \quad \forall i \in 1..n \end{array}}{E \vdash \text{Object}(X)[l_i v_i : B_i]_{i \in 1..n}} \quad \frac{\begin{array}{c} (\text{Type Class}) \quad (\text{where } A \equiv \text{Object}(X)[l_i v_i : B_i | X]_{i \in 1..n}) \\ E \vdash A \end{array}}{E \vdash \text{Class}(A)}$$

For subtyping, we have:

Subtyping

$$\frac{\begin{array}{c} (\text{Sub Refl}) \\ E \vdash A \end{array}}{E \vdash A <: A} \quad \frac{\begin{array}{c} (\text{Sub Trans}) \\ E \vdash A <: B \quad E \vdash B <: C \end{array}}{E \vdash A <: C} \quad \frac{\begin{array}{c} (\text{Sub X}) \\ E', X <: A, E'' \vdash \diamond \end{array}}{E', X <: A, E'' \vdash X <: A} \quad \frac{\begin{array}{c} (\text{Sub Top}) \\ E \vdash A \end{array}}{E \vdash A <: \text{Top}}$$

$$\frac{\begin{array}{c} (\text{Sub Object}) \quad (\text{where } A \equiv \text{Object}(X)[l_i v_i : B_i | X]_{i \in 1..n+m}, A' \equiv \text{Object}(X')[l_i v'_i : B'_i | X']_{i \in 1..n}) \\ E \vdash A \quad E \vdash A' \quad E, X <: A \vdash v_i B_i | X <: v'_i B'_i | A' \quad \forall i \in 1..n \end{array}}{E \vdash A <: A'}$$

$$\frac{\begin{array}{c} (\text{Sub Invariant}) \\ E \vdash B \end{array}}{E \vdash {}^0 B <: {}^0 B} \quad \frac{\begin{array}{c} (\text{Sub Covariant}) \\ E \vdash B <: B' \quad v \in \{^0, ^+\} \end{array}}{E \vdash v B <: {}^+ B'} \quad \frac{\begin{array}{c} (\text{Sub Contravariant}) \\ E \vdash B' <: B \quad v \in \{^0, ^-\} \end{array}}{E \vdash v B <: {}^- B'}$$

The most interesting rule is (Sub Object), which deals with subtypings between object types with variance annotations. It is more complex than the analogous rules that we have considered so far because of the occurrence of a bound variable X in the construct $\text{Object}(X)[\dots]$. When testing $A <: A'$, component by component, the occurrences of X' in A' are replaced with A' ; for the occurrences of X in A , there is the assumption that $X <: A'$. As in Chapter 8, three rules complement (Sub Object) for proving subtypings with variance annotations. In the example of Section 12.3, $CPoint2 <: Point$ is establishing by checking that ${}^+ Int \rightarrow X <: {}^+ Int \rightarrow Point$ under the assumption $X <: Point$.

The remaining rules for subtyping are basic ones (such as that for reflexivity). Note in particular that there is no rule for subtyping class types.

Finally, O-1 has the following rules for terms:

Terms

$$\begin{array}{c} \text{(Val Subsumption)} \\ E \vdash a : A \quad E \vdash A <: B \\ \hline E \vdash a : B \end{array} \qquad \begin{array}{c} \text{(Val } x\text{)} \\ E', x:A, E'' \vdash \diamond \\ \hline E', x:A, E'' \vdash x : A \end{array}$$

$$\begin{array}{c} \text{(Val Object) (where } A \equiv \mathbf{Object}(X)[l_i v_i; B_i\{X\}^{i \in 1..n}]\text{)} \\ E, x:A \vdash b_i : B_i\{A\} \quad \forall i \in 1..n \\ \hline E \vdash \mathbf{object}(x:A) \ l_i = b_i^{i \in 1..n} \ \mathbf{end} : A \end{array}$$

$$\begin{array}{c} \text{(Val Select) (where } A \equiv \mathbf{Object}(X)[l_i v_i; B_i\{X\}^{i \in 1..n}]\text{)} \\ E \vdash a : A \quad v_j \in \{^0, +\} \quad j \in 1..n \\ \hline E \vdash a.l_j : B_j\{A\} \end{array}$$

$$\begin{array}{c} \text{(Val Update) (where } A \equiv \mathbf{Object}(X)[l_i v_i; B_i\{X\}^{i \in 1..n}]\text{)} \\ E \vdash a : A \quad E \vdash b : B_j\{A\} \quad v_j \in \{^0, -\} \quad j \in 1..n \\ \hline E \vdash a.l_j := b : A \end{array}$$

$$\begin{array}{c} \text{(Val Method Update) (where } A \equiv \mathbf{Object}(X)[l_i v_i; B_i\{X\}^{i \in 1..n}]\text{)} \\ E \vdash a : A \quad E, x:A \vdash b : B_j\{A\} \quad v_j \in \{^0, -\} \quad j \in 1..n \\ \hline E \vdash a.l_j := \mathbf{method}(x:A)b \ \mathbf{end} : A \end{array}$$

$$\begin{array}{c} \text{(Val New)} \\ E \vdash c : \mathbf{Class}(A) \\ \hline E \vdash \mathbf{new} c : A \end{array}$$

$$\begin{array}{c} \text{(Val Root)} \\ E \vdash \diamond \\ \hline E \vdash \mathbf{root} : \mathbf{Class}(\mathbf{Object}(X)[]) \end{array}$$

$$\begin{array}{c} \text{(Val Subclass) (where } A \equiv \mathbf{Object}(X)[l_i v_i; B_i\{X\}^{i \in 1..n+m}], \ A' \equiv \mathbf{Object}(X')[l'_i v'_i; B'_i\{X'\}^{i \in 1..n}], \\ Ovr \subseteq 1..n) \\ E \vdash c' : \mathbf{Class}(A') \quad E \vdash A <: A' \\ E \vdash B'_i\{A'\} <: B_i\{A\} \quad \forall i \in 1..n-Ovr \\ E, x:A \vdash b_i : B_i\{A\} \quad \forall i \in Ovr \cup n+1..n+m \\ \hline E \vdash \mathbf{subclass_of} \ c' : \mathbf{Class}(A') \ \mathbf{with} (x:A) \ l_i = b_i^{i \in n+1..n+m} \ \mathbf{override} \ l_i = b_i^{i \in Ovr} \ \mathbf{end} : \mathbf{Class}(A) \end{array}$$

(Val Class Select) (where $A \equiv \text{Object}(X)[l_i; v_i; B_i\{X\}^{i \in 1..n}]$)

$$\frac{E \vdash a : A \quad E \vdash c : \text{Class}(A) \quad j \in 1..n}{E \vdash c^{\wedge l_j}(a) : B_j\{A\}}$$

(Val Typecase)

$$\frac{E \vdash a : A' \quad E, x:A \vdash b_1 : D \quad E \vdash b_2 : D}{E \vdash \text{typecase } a \text{ when } (x:A)b_1 \text{ else } b_2 \text{ end} : D}$$

The rules (Val x) and (Val Subsumption) are standard. The rule (Val Typecase) is as in Section 9.7. For constructing directly objects of type $A \equiv \text{Object}(X)[l_i; v_i; B_i\{X\}^{i \in 1..n}]$, according to (Val Object), it suffices to provide attributes of types $B_i\{A\}$, with the recursion variable X replaced with the entire object type A ; the self variable is given type A . Similar assumptions are required for the rules (Val Update) and (Val Method Update). In the rule for selection, (Val Select), a component l_j of an object of type $A \equiv \text{Object}(X)[l_i; v_i; B_i\{X\}^{i \in 1..n}]$ is selected; the resulting type is $B_j\{A\}$, where the recursion variable X is replaced with the type A .

As for classes, the rules (Val New) and (Val Root) are straightforward. The rule (Val Class Select) is also simple; it says that if a class c is for a type $\text{Object}(X)[l_i; v_i; B_i\{X\}^{i \in 1..n}]$ then $c^{\wedge l_j}$ behaves as a function from $\text{Object}(X)[l_i; v_i; B_i\{X\}^{i \in 1..n}]$ to B_j .

The rule for constructing subclasses, (Val Subclass), is fairly delicate. It requires that the subclass and the class are for object types A and A' , respectively, and that $A <: A'$; therefore subclassing requires subtyping. But this is not all; additional conditions are necessary for inheritability: a subclass of a class c' may inherit a method l_i only under certain subtyping assumptions (in the rule, $E \vdash B_i\{A'\} <: B_i\{A\}$). These assumptions guarantee that a method of the superclass behaves soundly when inherited in the subclass. For the methods that c' does not inherit (the new ones and the overridden ones), the rule stipulates simply that they must have the types expected in the subclass (in the rule, $E, x:A \vdash b_i : B_i\{A\}$).

In short, if a subclass redefines the type of a method, it is forced to override the method. We choose this subclassing mechanism for its flexibility. It would be possible to restrict the rule (Val Subclass) to guarantee that methods are always inheritable. The issue of inheritability will receive a different treatment in Parts II and III.

12.5 Translation

We relate O-1 to our calculi via a translation. In general, such a translation would have to be defined by induction on the typing derivations of the source language. For simplicity, instead, we give a more direct description of the translation: we add some type information to terms, and proceed by induction on the syntax of the source language. Specifically, we only need to add type information to field update, adopting the syntax $(a:A).l_j := b$ (instead of $a.l_j := b$) and the rule:

$$\begin{array}{c}
 (\text{Val Update}) \quad (\text{where } A \equiv \text{Object}(X)[l_i v_i : B_i[X]^{i \in 1..n}]) \\
 E \vdash a : A \quad E \vdash b : B_j[A] \quad v_j \in \{^0, ^-\} \quad j \in 1..n \\
 \hline
 E \vdash (a:A).l_j := b : A
 \end{array}$$

It is straightforward to translate O-1 to this type-enriched variant of O-1, by induction on derivations.

We show a translation into a functional target calculus; a similar translation could be given into an appropriate imperative calculus. We build a target calculus from many of the rules studied in Part I. We take $\text{FOB}_{1<\mu}$ from Chapter 9, modified with the stronger rule (Sub Rec') of Section 9.2 instead of the standard rule (Sub Rec), and extended with the rules for variance annotations of Chapter 8, and with (Val Typecase) from Section 9.7. The use of (Sub Rec') is necessary for validating the rule for subtyping object types in O-1; in particular it is necessary for the subtyping $CPoint2 <: Point$ in the example of Section 12.3.

At the level of types, the translation is reasonably simple. We write $\langle\!\langle A \rangle\!\rangle$ for the translation of A . We map an object type $\text{Object}(X)[l_i v_i : B_i[X]^{i \in 1..n}]$ to a recursive object type $\mu(X)[l_i v_i : \langle\!\langle B_i \rangle\!\rangle^{i \in 1..n}]$. Much as in Section 8.5, we map a class type $\text{Class}(\text{Object}(X)[l_i v_i : B_i[X]^{i \in 1..n}])$ to an object type that contains components for pre-methods and a *new* component. We assume that the label *new* does not occur in the source language.

Translation of O-1 types

$$\begin{aligned}
 \langle\!\langle X \rangle\!\rangle &\triangleq X \\
 \langle\!\langle \text{Top} \rangle\!\rangle &\triangleq \text{Top} \\
 \langle\!\langle \text{Object}(X)[l_i v_i : B_i[X]^{i \in 1..n}] \rangle\!\rangle &\triangleq \mu(X)[l_i v_i : \langle\!\langle B_i \rangle\!\rangle^{i \in 1..n}] \\
 \langle\!\langle \text{Class}(A) \rangle\!\rangle &\triangleq [\text{new}^+ : \langle\!\langle A \rangle\!\rangle, l_i^+ : \langle\!\langle A \rangle\!\rangle \rightarrow \langle\!\langle B_i \rangle\!\rangle[\langle\!\langle A \rangle\!\rangle]^{i \in 1..n}] \\
 &\text{where } A \equiv \text{Object}(X)[l_i v_i : B_i[X]^{i \in 1..n}]
 \end{aligned}$$

This translation of types induces a translation of environments:

Translation of O-1 environments

$$\begin{aligned}
 \langle\!\langle \emptyset \rangle\!\rangle &\triangleq \emptyset \\
 \langle\!\langle E, X <: A \rangle\!\rangle &\triangleq \langle\!\langle E \rangle\!\rangle, X <: \langle\!\langle A \rangle\!\rangle \\
 \langle\!\langle E, x : A \rangle\!\rangle &\triangleq \langle\!\langle E \rangle\!\rangle, x : \langle\!\langle A \rangle\!\rangle
 \end{aligned}$$

The translation of terms is virtually forced by the translation of types, but is not altogether simple. We write $\langle\!\langle a \rangle\!\rangle$ for the translation of a . The precise definition of $\langle\!\langle a \rangle\!\rangle$ relies on many uses of *fold* and *unfold*, necessary in connection with recursive types; it is easiest to ignore them at first. It is also convenient to use our notations for fields in object calculi, for example writing $[..., l=b, ...]$ instead of $[..., l=\zeta(y:A)b, ...]$ for an

unused y and an appropriate A . We obtain the following preliminary version of the definition:

Preliminary translation of O-1 terms

$\langle x \rangle \triangleq x$
$\langle \text{object}(x:A) l_i=b_i \text{ for } i \in 1..n \text{ end} \rangle \triangleq [l_i=\zeta(x:\langle A \rangle)(\langle b_i \rangle) \text{ for } i \in 1..n]$
$\langle a.l \rangle \triangleq \langle a \rangle.l$
$\langle (a:A).l := b \rangle \triangleq \langle a \rangle.l := \langle b \rangle$
$\langle a.l := \text{method}(x:A) b \text{ end} \rangle \triangleq \langle a \rangle.l \leftarrow \zeta(x:\langle A \rangle)(\langle b \rangle)$
$\langle \text{new } c \rangle \triangleq \langle c \rangle.\text{new}$
$\langle \text{root} \rangle \triangleq [\text{new} = []]$
$\langle \text{subclass of } c':\text{Class}(A') \text{ with } (x:A) l_i=b_i \text{ for } i \in n+1..n+m \text{ override } l_i=b_i \text{ for } i \in \text{Ovr} \text{ end} \rangle \triangleq$
$[\text{new} = \zeta(z:\langle \text{Class}(A) \rangle)[l_i=\zeta(s:\langle A \rangle)z.l_i(s) \text{ for } i \in 1..n+m],$
$l_i=\langle c' \rangle.l_i \text{ for } i \in 1..n-\text{Ovr},$
$l_i=\lambda(x:\langle A \rangle)(\langle b_i \rangle) \text{ for } i \in \text{Ovr} \cup n+1..n+m]$
$\langle c^l(a) \rangle \triangleq \langle c \rangle.l(\langle a \rangle)$
$\langle \text{typecase } a \text{ when } (x:A)b_1 \text{ else } b_2 \text{ end} \rangle \triangleq \text{typecase } \langle a \rangle \mid (x:\langle A \rangle)(\langle b_1 \rangle) \mid \langle b_2 \rangle$

The preliminary translation deals trivially with typecase and with the object-based constructs of O-1. More interestingly, it maps a class to a collection of pre-methods plus a *new* method. For the base class *root*, there are no pre-methods. For a class *subclass of c' ... end*, the collection of pre-methods consists of the pre-methods of *c'* that are not overridden, plus all the pre-methods given explicitly. The *new* method assembles the pre-methods into an object; the construct *new c* is interpreted as an invocation of the *new* method of $\langle c \rangle$. The construct $\langle c^l(a) \rangle$ is interpreted as the extraction and the application of a pre-method.

These ideas lead to the precise definition of the translation. In this definition, we write C^\dagger for $B\{C\}$ when C is a recursive type $\mu(X)B\{X\}$.

Translation of O-1 terms

$\langle x \rangle \triangleq x$
$\langle \text{object}(x:A) l_i=b_i \text{ for } i \in 1..n \text{ end} \rangle \triangleq \text{fold}(\langle A \rangle, [l_i=\zeta(x:\langle A \rangle^\dagger)(\langle b_i \rangle) \mid \text{fold}(\langle A \rangle, x) \text{ for } i \in 1..n])$
where $A \equiv \text{Object}(X)[l_i; v_i: B_i \text{ for } i \in 1..n]$
$\langle a.l \rangle \triangleq \text{unfold}(\langle a \rangle).l$
$\langle (a:A).l := b \rangle \triangleq \text{fold}(\langle A \rangle, \text{unfold}(\langle a \rangle).l \leftarrow \zeta(x:\langle A \rangle^\dagger)(\langle b \rangle))$
where $A \equiv \text{Object}(X)[l_i; v_i: B_i \text{ for } i \in 1..n]$ and $x \notin FV(\langle b \rangle)$

$$\begin{aligned}
\langle a.l := \mathbf{method}(x:A) b\{x\} \mathbf{end} \rangle &\triangleq \text{fold}(\langle A \rangle, \text{unfold}(\langle a \rangle).l \in \zeta(x:\langle A \rangle^\dagger) \langle b \rangle \langle \text{fold}(\langle A \rangle, x) \rangle) \\
\text{where } A &\equiv \mathbf{Object}(X)[l_i v_i : B_i]_{i \in 1..n} \\
\langle \mathbf{new} c \rangle &\triangleq \langle c \rangle.\mathbf{new} \\
\langle \mathbf{root} \rangle &\triangleq [\mathbf{new} = \zeta(z:[\mathbf{new}^+ : \mu(X)[\cdot]])) \text{fold}(\mu(X)[\cdot], [\cdot])] \\
\langle \mathbf{subclass} \text{ of } c' : \mathbf{Class}(A') \text{ with } (x:A) l_i = b_i^{i \in n+1..n+m} \mathbf{override} l_i = b_i^{i \in Ovr} \mathbf{end} \rangle &\triangleq \\
&[\mathbf{new} = \zeta(z:\langle \mathbf{Class}(A) \rangle) \text{fold}(\langle A \rangle, [l_i = \zeta(s:\langle A \rangle^\dagger) z.l_i \langle \text{fold}(\langle A \rangle, s) \rangle^{i \in 1..n+m}]), \\
&l_i = \zeta(z:\langle \mathbf{Class}(A) \rangle) \langle c' \rangle.l_i^{i \in 1..n-Ovr}, \\
&l_i = \zeta(z:\langle \mathbf{Class}(A) \rangle) \lambda(x:\langle A \rangle) \langle b_i \rangle^{i \in Ovr \cup n+1..n+m}] \\
&\text{where } A \equiv \mathbf{Object}(X)[l_i v_i : B_i]_{i \in 1..n+m} \text{ and } A' \equiv \mathbf{Object}(X)[l_i v_i' : B_i']_{i \in 1..n} \\
&\text{and } z \notin FV(\langle c' \rangle) \cup FV(\langle b_i \rangle)^{i \in Ovr \cup n+1..n+m}] \\
\langle c \wedge l(a) \rangle &\triangleq \langle c \rangle.l(\langle a \rangle) \\
\langle \mathbf{typecase} \ a \ \mathbf{when} \ (x:A)b_1 \ \mathbf{else} \ b_2 \ \mathbf{end} \rangle &\triangleq \text{typecase } \langle a \rangle \mid (x:\langle A \rangle)\langle b_1 \rangle \mid \langle b_2 \rangle
\end{aligned}$$

Combining the translations of types, environments, and terms, we can translate whole judgments, for example mapping $E \vdash \diamond$ to $\langle E \rangle \vdash \diamond$ and $E \vdash a : A$ to $\langle E \rangle \vdash \langle a \rangle : \langle A \rangle$. We write $\langle E \vdash \mathfrak{I} \rangle$ for the translation of $E \vdash \mathfrak{I}$. As a soundness result, we obtain that if $E \vdash \mathfrak{I}$ is derivable in the source language then $\langle E \vdash \mathfrak{I} \rangle$ is derivable in the target calculus. This result can be proved by induction on derivations.

Thus we have succeeded in using object calculi as a platform for explaining a relatively rich object-oriented language and for validating its type rules.

PART II

Second-Order Calculi

13 SECOND-ORDER CALCULI

Part II is primarily concerned with second-order concepts: type quantification and the type *Self*. In this chapter, we start by adding second-order type quantifiers to our first-order calculi. These quantifiers are standard; they were not designed specifically with objects in mind. However, these quantifiers can be combined with recursive types to produce an interesting new construct that is recognizable as formalizing the type *Self*. The interaction of *Self* with object types is the subject of Chapter 15. Subsequent chapters concern the direct axiomatization of *Self* types, and their use in a programming language.

In this chapter, we first introduce universal quantifiers and existential quantifiers. From existential quantifiers and recursion we define the *Self quantifier*. For the purpose of defining the *Self* quantifier, only existential quantifiers and recursion are needed; one could dispense with universal quantifiers. For the purposes of object-oriented languages, only the *Self* quantifier is needed; one could dispense with most of the second-order baggage. However, universal quantifiers are still useful in an object-oriented language to provide parametric polymorphism, and existential quantifiers to provide data abstraction.

13.1 The Universal Quantifier

We begin by reviewing simple universal quantification over types, of the kind traditionally used to model parametric polymorphism. We then consider bounded quantification, which combines parametric polymorphism with subtyping.

13.1.1 Parametric Polymorphism

As before, we use the notation $b\{X\}$ and $B\{X\}$ to single out the occurrences of X free in b and in B . Moreover, we use our substitution conventions for types, so $b[A]$ stands for $b[X \leftarrow A]$ and $B[A]$ stands for $B[X \leftarrow A]$ when X is clear from context.

A term $\lambda(X)b\{X\}$ represents a term b parameterized with respect to a type variable X ; we call this a *type abstraction*. Correspondingly, a term $a(A)$ is the application of a term a to a type A ; we call this a *type application*. Thus $b[A]$ is an instantiation of the type abstraction $\lambda(X)b\{X\}$ for a specific type A , and can be obtained as the result of a type application $(\lambda(X)b\{X\})(A)$. We write $\forall(X)B\{X\}$ for the type of those type abstractions $\lambda(X)b\{X\}$ that for any type A produce a result $b[A]$ of type $B\{A\}$.

For example, we may write:

$id : \forall(X) X \rightarrow X \triangleq \lambda(X) \lambda(x:X) x$	the polymorphic identity function
$id(Int) : Int \rightarrow Int$	its instantiation to the integer type
$id(Int)(3) : Int$	its application to an integer

Polymorphism is obtained from the Δ_X fragment for type variables of Section 9.1, plus the following fragments for universal quantifiers.

 Δ_\forall

(Type All)	(Val Fun2)	(Val Appl2)
$E, X \vdash B$	$E, X \vdash b : B$	$E \vdash b : \forall(X)B\{X\} \quad E \vdash A$
$E \vdash \forall(X)B$	$E \vdash \lambda(X)b : \forall(X)B$	$E \vdash b(A) : B\{A\}$

 $\Delta_{=\forall}$

(Eq Fun2)	(Eq Appl2)
$E, X \vdash b \leftrightarrow b' : B$	$E \vdash b \leftrightarrow b' : \forall(X)B\{X\} \quad E \vdash A$
$E \vdash \lambda(X)b \leftrightarrow \lambda(X)b' : \forall(X)B$	$E \vdash b(A) \leftrightarrow b'(A) : B\{A\}$
(Eval Beta2)	(Eval Eta2)
$E \vdash \lambda(X)b\{X\} : \forall(X)B\{X\} \quad E \vdash A$	$E \vdash b : \forall(X)B \quad X \notin \text{dom}(E)$
$E \vdash (\lambda(X)b\{X\})(A) \leftrightarrow b\{A\} : B\{A\}$	$E \vdash \lambda(X)b(X) \leftrightarrow b : \forall(X)B$

The rule (Type All) forms a quantified type $\forall(X)B$ in E , provided that B is well-formed in E extended with X . The rule (Val Fun2) constructs a type abstraction $\lambda(X)b$ of type $\forall(X)B$, provided that the body b has type B for an arbitrary type parameter X (which may occur in b and B). The rule (Val Appl2) applies such a type abstraction to a type A .

The equational rules (Eq Fun2) and (Eq Appl2) are simply congruence rules. The rules (Eval Beta2) and (Eval Eta2) are the equational versions of β -reduction and η -reduction for type abstractions.

The second-order λ -calculus F_2 , usually called F [63], is assembled below, along with related calculi. The ground type K used in first-order calculi is left out because free algebras can be encoded within F [28].

Starting in Section 13.2, we include existential quantifiers in all our second-order calculi. We indicate calculi without existentials by italic letters:

$$\begin{array}{lll} F & \triangleq & \Delta_x \cup \Delta_\rightarrow \cup \Delta_X \cup \Delta_\forall \\ Ob & \triangleq & \Delta_x \cup \Delta_{Ob} \cup \Delta_X \cup \Delta_\forall \\ FO\!b & \triangleq & F \cup \Delta_{Ob} \end{array} \quad \begin{array}{lll} F_\mu & \triangleq & F \cup \Delta_\mu \\ Ob_\mu & \triangleq & Ob \cup \Delta_\mu \\ FO\!b_\mu & \triangleq & FO\!b \cup \Delta_\mu \end{array}$$

The encoding of function types as object types (Section 7.7) extends trivially to second-order calculi, so Ob can encode F and $FO\!b$. It is evident that F cannot encode Ob ,

since F is strongly normalizing [63] and Ob is not (Section 7.4.1). However, F_μ can encode Ob and Ob_μ via a typed version of the self-application semantics (Section 6.7.1).

13.1.2 Bounded Parametric Polymorphism

We proceed to second-order calculi with subtyping. We add a subtyping judgment, as for first-order systems, and we extend universally quantified types $\forall(X)B$ to bounded universally quantified types $\forall(X<:A)B$, where A is the bound on X . The bounded type abstraction $\lambda(X<:A)b[X]$ has type $\forall(X<:A)B\{X\}$ if, for any subtype A' of A , the instantiation $b[A']$ has type $B\{A'\}$.

We reuse the $\Delta_{<:x}$ fragment of Section 9.2, where type variables have bounds in environments, and add:

$\Delta_{<:\forall}$

(Type All $<:$)	(Sub All)
$E, X<:A \vdash B$	$E \vdash A' <: A \quad E, X<:A' \vdash B <: B'$
	$E \vdash \forall(X<:A)B <: \forall(X<:A')B'$
(Val Fun2 $<:$)	(Val Appl2 $<:$)
$E, X<:A \vdash b : B$	$E \vdash b : \forall(X<:A)B\{X\} \quad E \vdash A' <: A$
$E \vdash \lambda(X<:A)b : \forall(X<:A)B$	$E \vdash b(A') : B\{A'\}$

These rules generalize those of the fragment Δ_\forall to bounded universal quantifiers. The rule (Sub All) describes the subtype relation between the new quantifiers; note the inverse inclusion of the bounds, and the assumption under which the bodies are compared. This rule determines the variance behavior of the universal quantifier. If A is contravariant in Y and B is covariant in Y , then $\forall(X<:A)B$ is covariant in Y . If A is covariant in Y and B is contravariant in Y , then $\forall(X<:A)B$ is contravariant in Y . Therefore we say that a universally quantified type is contravariant in its bound and covariant in its body.

The equational rules are simple generalizations of those of the fragment Δ_\forall .

$\Delta_{=:\forall}$

(Eq Fun2 $<:$)	(Eq Appl2 $<:$)
$E, X<:A \vdash b \leftrightarrow b' : B$	$E \vdash b \leftrightarrow b' : \forall(X<:A)B\{X\} \quad E \vdash A' <: A$
$E \vdash \lambda(X<:A)b \leftrightarrow \lambda(X<:A)b' : \forall(X<:A)B$	$E \vdash b(A') \leftrightarrow b'(A') : B\{A'\}$
(Eval Beta2 $<:$)	(Eval Eta2 $<:$)
$E \vdash \lambda(X<:A)b\{X\} : \forall(X<:A)B\{X\} \quad E \vdash C <: A$	$E \vdash b : \forall(X<:A)B \quad X \notin \text{dom}(E)$
$E \vdash (\lambda(X<:A)b\{X\})(C) \leftrightarrow b\{C\} : B\{C\}$	$E \vdash \lambda(X<:A)b(X) \leftrightarrow b : \forall(X<:A)B$

We obtain the calculi:

$$\begin{aligned}
 F_{<} &\triangleq \Delta_x \cup \Delta_{\rightarrow} \cup \Delta_{<} \cup \Delta_{<:X} \cup \Delta_{<:\rightarrow} \cup \Delta_{<:\forall} \\
 Ob_{<} &\triangleq \Delta_x \cup \Delta_{Ob} \cup \Delta_{<} \cup \Delta_{<:X} \cup \Delta_{<:Ob} \cup \Delta_{<:\forall} \\
 FO_{Ob_{<}} &\triangleq F_{<} \cup \Delta_{Ob} \cup \Delta_{<:Ob} \\
 F_{<:\mu} &\triangleq F_{<} \cup \Delta_{<:\mu} \\
 Ob_{<:\mu} &\triangleq Ob_{<} \cup \Delta_{<:\mu} \\
 FO_{Ob_{<:\mu}} &\triangleq FO_{Ob_{<}} \cup \Delta_{<:\mu}
 \end{aligned}$$

$F_{<}$ is described in [44], but we assume only the simpler equational theory used in [56], as given by the fragment $\Delta_{=:\forall}$.

13.1.3 Structural Update

While adding subtyping and polymorphism to our calculi, we have consciously chosen not to modify the Δ_{Ob} fragment. However, it is tempting to change the (Val Update) rule as follows:

$$\frac{\text{(Val Structural Update)} \quad (\text{where } A \equiv [l_i; B_i]_{i \in 1..n})}{E \vdash a : C \quad E \vdash C <: A \quad E, x:C \vdash b : B_j \quad j \in 1..n}
 \frac{}{E \vdash a.l_j \in \zeta(x:C)b : C}$$

The difference between (Val Update) and (Val Structural Update) can be seen when C is a type variable, as in the following example:

$$\begin{aligned}
 \lambda(C <: [l:Nat]) \lambda(a:C) a.l := 3 &: \forall(C <: [l:Nat]) C \rightarrow [l:Nat] \quad \text{via (Val Update)} \\
 \lambda(C <: [l:Nat]) \lambda(a:C) a.l := 3 &: \forall(C <: [l:Nat]) C \rightarrow C \quad \text{via (Val Structural Update)}
 \end{aligned}$$

The new rule (Val Structural Update) appears intuitively sound. It implicitly relies on the invariance of object types, and on the “structural subtyping” assumption that every subtype of an object type is an object type. Such an assumption is quite easily realized in programming languages and can be shown to hold in formal systems such as ours.

However, structural subtyping is not satisfied by the semantics of Chapter 14, where the subtype relation is simply the “unstructured” subset relation. In that semantics, the type $\forall(C <: [l:Nat]) C \rightarrow C$ contains only an identity function and its approximations, and hence does not contain the preceding program, $\lambda(C <: [l:Nat]) \lambda(a:C) a.l := 3$. This fact is not a special property of our semantics, but rather a consequence of the parametricity of polymorphic functions [108]. A semantics that satisfies structural subtyping could probably be devised, but would be more complex and might have a rather syntactic flavor (see [43] for a first-order example of such a semantics).

This semantic difficulty suggests that we should proceed with caution. Hence we do not adopt the rule (Val Structural Update) at once, but we return to the topic of structural assumptions later on, in Section 15.7 and in Chapter 16.

13.2 The Existential Quantifier

We introduce the bounded version of the existential quantifier directly. The unbounded version is a special case.

The existentially quantified type $\exists(X \subset A)B[X]$ is the type of the pairs (A', b) where A' is a subtype of A and b is a term of type $B[A']$. The type $\exists(X \subset A)B[X]$ can be seen as a partially abstract data type with *interface* $B[X]$ and with *representation type* X known only to be a subtype of A [45, 93]. It is partially abstract in that it gives some information about the representation type, namely, a bound. The pair (A', b) describes an element of the partially abstract data type with representation type A' and *implementation* b . In order to be fully explicit, we write the pair (A', b) more verbosely in the form:

pack X $\subset A = A'$ *with b* $\{X\} : B[X]$

where $X \subset A = A'$ indicates that $X \subset A$ and $X = A'$. This notation expresses that the representation type X is A' , and it uniquely determines the type $\exists(X \subset A)B[X]$.

An element c of type $\exists(X \subset A)B[X]$ can be used in the construct:

open c as X $\subset A, x : B[X]$ *in d* $\{X, x\} : D$

where d has access to the representation type X and the implementation x of c , and must produce a result of a type D that does not depend on X . The requirement that X does not occur in D corresponds to the informal requirement that the type abstraction be respected. At evaluation time, if c is (A', b) , then the result is $d[A', b]$ of type D . For example, we may write:

$p : \exists(X \subset \text{Int})X \times (X \rightarrow X) \triangleq \text{pack } X \subset \text{Int} = \text{Nat} \text{ with } (0, \text{succ}_{\text{Nat}}) : X \times (X \rightarrow X)$

$a : \text{Int} \triangleq \text{open } p \text{ as } X \subset \text{Int}, x : X \times (X \rightarrow X) \text{ in } \text{snd}(x)(\text{fst}(x)) : \text{Int}$

and then $a = 1$.

We adopt the following rules for existentially quantified types:

Δ_{\exists}

$$\frac{\begin{array}{c} (\text{Type Exists}\subset:) \\ E, X \subset A \vdash B \end{array} \quad \begin{array}{c} (\text{Sub Exists}) \\ E \vdash A \subset A' \quad E, X \subset A \vdash B \subset B' \end{array}}{E \vdash \exists(X \subset A)B \quad E \vdash \exists(X \subset A)B \subset \exists(X \subset A)B'}$$

$$\frac{(\text{Val Pack}\subset:) \quad \begin{array}{c} E \vdash C \subset A \quad E \vdash b\{C\} : B\{C\} \end{array}}{E \vdash \text{pack } X \subset A = C \text{ with } b\{X\} : B\{X\} : \exists(X \subset A)B[X]}$$

$$\frac{\begin{array}{c} (\text{Val Open}\subset:) \\ E \vdash c : \exists(X \subset A)B \quad E \vdash D \quad E, X \subset A, x : B \vdash d : D \end{array}}{E \vdash \text{open } c \text{ as } X \subset A, x : B \text{ in } d : D : D}$$

The rule (Sub Exists) determines the variance behavior of the existential quantifier. If A and B are covariant in Y , then so is $\exists(X \subset : A)B$. If A and B are contravariant in Y , then so is $\exists(X \subset : A)B$. Therefore we say that an existentially quantified type is covariant in its bound and its body.

If B is covariant in X , then $\exists(X \subset : A)B[X]$ is isomorphic to $B[A]$; the isomorphism can be shown formally for particular systems [44, 104]. If B is not covariant in X , then $\exists(X \subset : A)B[X]$ may not be isomorphic to $B[A]$. In particular, $\exists(X \subset : A)[l; B_i[X]^{i \in 1..n}]$ is not isomorphic to $[l; B_i[A]^{i \in 1..n}]$, even if each B_i is covariant in X , because there is no appropriate function from $\exists(X \subset : A)[l; B_i[X]^{i \in 1..n}]$ to $[l; B_i[A]^{i \in 1..n}]$. For instance, $[l:X]$ is not covariant in X , and $\exists(X \subset : Int)[l:X]$ is not isomorphic to $[l:Int]$. The failure of this isomorphism is fairly clear since values of types $\exists(X \subset : Int)[l:X]$ and $[l:Int]$ cannot be used in the same ways; for example, if o has type $[l:Int]$ then it is legal to set $o.l:=3$, but there is no corresponding operation on values of type $\exists(X \subset : Int)[l:X]$.

The fragment $\Delta_{\leqslant \subset : \exists}$ gives an equational theory for bounded existentials.

$\Delta_{\leqslant \subset : \exists}$

(Eq Pack $\subset :$)

$$E \vdash C \subset : A' \quad E \vdash A' \subset : A \quad E, X \subset : A' \vdash B'[X] \subset : B[X] \quad E \vdash b[C] \leftrightarrow b'[C] : B'[C]$$

$$E \vdash \text{pack } X \subset : A = C \text{ with } b[X]:B[X] \leftrightarrow \text{pack } X \subset : A' = C \text{ with } b'[X]:B'[X] : \exists(X \subset : A)B[X]$$

(Eq Open $\subset :$)

$$E \vdash c \leftrightarrow c' : \exists(X \subset : A)B \quad E \vdash D \quad E, X \subset : A, x:B \vdash d \leftrightarrow d' : D$$

$$E \vdash \text{open } c \text{ as } X \subset : A, x:B \text{ in } d:D \leftrightarrow \text{open } c' \text{ as } X \subset : A, x:B \text{ in } d':D : D$$

(Eval Unpack $\subset :$) (where $c \equiv \text{pack } X \subset : A = C \text{ with } b[X]:B[X]$)

$$E \vdash c : \exists(X \subset : A)B[X] \quad E \vdash D \quad E, X \subset : A, x:B[X] \vdash d[X,x] : D$$

$$E \vdash \text{open } c \text{ as } X \subset : A, x:B[X] \text{ in } d[X,x]:D \leftrightarrow d[C,b[C]] : D$$

(Eval Repack $\subset :$)

$$E \vdash b : \exists(X \subset : A)B[X] \quad E, y:\exists(X \subset : A)B[X] \vdash d[y] : D$$

$$E \vdash \text{open } b \text{ as } X \subset : A, x:B[X] \text{ in } d[\text{pack } X' \subset : A = X \text{ with } x:B[X']] : D \leftrightarrow d[b] : D$$

The rules (Eq Pack $\subset :$) and (Eq Open $\subset :$) are congruence rules. The rule (Eq Pack $\subset :$) takes subtyping into account, equating *pack* terms with different type bounds A and A' and type bodies $B[X]$ and $B'[X]$. A more flexible congruence rule is derivable:

(Eq Pack $\subset :$ Lemma)

$$E \vdash C \subset : A' \quad E \vdash A' \subset : A \quad E, X \subset : A' \vdash B'[X] \subset : B[X]$$

$$E \vdash b[C] \leftrightarrow b'[C] : B[C] \quad E \vdash b'[C] : B'[C]$$

$$E \vdash \text{pack } X \subset : A = C \text{ with } b[X]:B[X] \leftrightarrow \text{pack } X \subset : A' = C \text{ with } b'[X]:B'[X] : \exists(X \subset : A)B[X]$$

For the derivation of (Eq Pack $<:$ Lemma), we use (Eq Pack $<:$) twice, first with $b \equiv b'$ and then with $A \equiv A'$ and $B \equiv B'$, and apply transitivity to the two results.

The rules (Eval Unpack $<:$) and (Eval Repack $<:$) are evaluation rules. The rule (Eval Unpack $<:$) requires c to contain the types A and B mentioned in the *open* construct. This may not be the case if c has been obtained via subsumption: c could have the form (*pack* $X <: A' = C$ with $b:B'$) where $A' <: A$ and $B' <: B$. However, then, the rules (Eq Pack $<:$) and (Eq Open $<:$) can be used to replace the inadequate c with an adequate one.

The equational rules are somewhat conservative. In particular, they do not equate *pack* terms based on different representation types. For example, we might expect that the terms (*pack* $X <: \text{Int} = \text{Nat}$ with $\langle 0, \text{succ}_{\text{Nat}} \rangle$) and (*pack* $X <: \text{Int} = \text{Int}$ with $\langle 0, \text{succ}_{\text{Int}} \rangle$) be equal at type $\exists(X <: \text{Int})X \times (X \rightarrow X)$ because they have the same behavior, but this does not seem to be derivable. This equality can be proved in a logic that includes an axiom of parametricity [104]. For simplicity we restrict attention to our conservative rules, although more ambitious rules may have advantages in combination with objects.

Within $F_{<:}$ (and hence $FOb_{<:}$) it is possible to encode bounded existential quantifiers with the following definitions:

$$\begin{aligned}\exists(X <: A)B\{X\} &\triangleq && (Y \text{ not occurring in } A \text{ or } B) \\ \forall(Y <: \text{Top})(\forall(X <: A)B\{X\} \rightarrow Y) \rightarrow Y \\ \text{pack } X <: A = C \text{ with } b\{X\}:B\{X\} &\triangleq \lambda(Y <: \text{Top})\lambda(f:\forall(X <: A)B\{X\} \rightarrow Y)f(C)(b\{C\}) \\ \text{open } c \text{ as } X <: A, x:B\{X\} \text{ in } d\{X,x\}:D &\triangleq c(D)(\lambda(X <: A)\lambda(x:B\{X\})d\{X,x\})\end{aligned}$$

All our rules for \exists , except for (Eval Repack $<:$), are derivable from the encoding. (In particular, the encoding of (Eq Pack $<:$) is derivable in $F_{<:}$ using the Domain Restriction Lemma of [44].) We take \exists as primitive in order to have (Eval Repack $<:$), which is useful in the sequel. The primitive \exists is also important as an addition to the calculi $Ob_{<:}$ and $Ob_{<:\mu}$. These calculi can express only invariant function types (as far as we know), so the encoding of \exists in terms of \forall and \rightarrow just shown does not validate (Sub Exists) in $Ob_{<:}$ and $Ob_{<:\mu}$.

We now add \exists to our calculi. The fragments for the unbounded existential quantifier, Δ_\exists and $\Delta_{=\exists}$, are in the Appendices A.2 and A.3. The definition of $F_{<:\mu}$ is repeated in Appendix B.2.

$$\begin{array}{lll} F & \triangleq & F \cup \Delta_\exists \\ Ob & \triangleq & Ob \cup \Delta_\exists \\ FOb & \triangleq & FOb \cup \Delta_\exists \\ F_{<:} & \triangleq & F_{<:} \cup \Delta_{<:\exists} \\ Ob_{<:} & \triangleq & Ob_{<:} \cup \Delta_{<:\exists} \\ FOb_{<:} & \triangleq & FOb_{<:} \cup \Delta_{<:\exists} \end{array} \quad \begin{array}{lll} F_\mu & \triangleq & F_\mu \cup \Delta_\exists \\ Ob_\mu & \triangleq & Ob_\mu \cup \Delta_\exists \\ FOb_\mu & \triangleq & FOb_\mu \cup \Delta_\exists \\ F_{<:\mu} & \triangleq & F_{<:\mu} \cup \Delta_{<:\exists} \\ Ob_{<:\mu} & \triangleq & Ob_{<:\mu} \cup \Delta_{<:\exists} \\ FOb_{<:\mu} & \triangleq & FOb_{<:\mu} \cup \Delta_{<:\exists} \end{array}$$

In Section 13.4 we show that $Ob_{<:}$ can encode $F_{<:}$, and that $Ob_{<:\mu}$ can encode $F_{<:\mu}$. Conversely, we discuss several more or less satisfactory interpretations of $Ob_{<:}$ in $F_{<:}$ in Chapter 18.

We summarize the definition of $\text{FOb}_{<:\mu}$, the largest of the calculi defined above, by giving its syntax and scoping rules:

Syntax of the $\text{FOb}_{<:\mu}$ calculus

$A, B, C, D ::=$	types
X	type variable
Top	the biggest type
$[l_i:B_i]^{i \in 1..n}$	object type (l_i distinct)
$A \rightarrow B$	function type
$\mu(X)A$	recursive type
$\forall(X <: A)B$	bounded universal type
$\exists(X <: A)B$	bounded existential type
$a, b, c, d ::=$	terms
x	variable
$[l_i=\zeta(x_i:A_i)b_i]^{i \in 1..n}$	object (l_i distinct)
$a.l$	method invocation
$a.l = \zeta(x:A)b$	method update
$\lambda(x:A)b$	function
$b(a)$	application
$\text{fold}(A, a)$	recursive fold
$\text{unfold}(a)$	recursive unfold
$\lambda(X <: A)b$	type abstraction
$b(A)$	type application
$\text{pack } X <: A = C \text{ with } b : B$	data abstraction packaging
$\text{open } c \text{ as } X <: A, x : B \text{ in } d : D$	data abstraction opening

Scoping for the $\text{FOb}_{<:\mu}$ calculus

$FV(X)$	$\triangleq \{X\}$
$FV(\text{Top})$	$\triangleq \{\}$
$FV([l_i:B_i]^{i \in 1..n})$	$\triangleq \cup^{i \in 1..n} FV(B_i)$
$FV(A \rightarrow B)$	$\triangleq FV(A) \cup FV(B)$
$FV(\mu(X)A)$	$\triangleq FV(A) - \{X\}$
$FV(\forall(X <: A)B)$	$\triangleq FV(A) \cup (FV(B) - \{X\})$
$FV(\exists(X <: A)B)$	$\triangleq FV(A) \cup (FV(B) - \{X\})$
$FV(\zeta(x:A)b)$	$\triangleq FV(A) \cup (FV(b) - \{x\})$
$FV(x)$	$\triangleq \{x\}$
$FV([l_i=\zeta(x_i:A_i)b_i]^{i \in 1..n})$	$\triangleq \cup^{i \in 1..n} FV(\zeta(x_i:A_i)b_i)$
$FV(a.l)$	$\triangleq FV(a)$

$FV(a.l \in \zeta(x:A)b)$	$\triangleq FV(a) \cup FV(\zeta(x:A)b)$
$FV(\lambda(x:A)b)$	$\triangleq FV(A) \cup (FV(b)-\{x\})$
$FV(b(a))$	$\triangleq FV(b) \cup FV(a)$
$FV(fold(A,a))$	$\triangleq FV(A) \cup FV(a)$
$FV(unfold(a))$	$\triangleq FV(a)$
$FV(\lambda(X <: A)b)$	$\triangleq FV(A) \cup (FV(b)-\{X\})$
$FV(b(A))$	$\triangleq FV(b) \cup FV(A)$
$FV(pack X <: A = C \text{ with } b:B)$	$\triangleq FV(A) \cup FV(C) \cup ((FV(b) \cup FV(B))-\{X\})$
$FV(open c \text{ as } X <: A, x:B \text{ in } d:D)$	$\triangleq FV(c) \cup FV(A) \cup (FV(B)-\{X\})$ $\cup (FV(d)-\{X,x\}) \cup (FV(D)-\{X\})$

In Chapter 14 we give a denotational semantics of $\mathbf{FOB}_{\leq:\mu}$ that guarantees its soundness. For the sake of brevity, therefore, we do not prove subject reduction results. Also for the sake of brevity, we do not prove minimum-types properties.

13.3 Variance Properties

So far we have dealt with variance rather informally. We now give a syntactic definition of variance that yields the expected formal properties.

Recall that the notation $A\{X\}$ identifies all the free occurrences of X in A , so that $A\{B\}$ is the result of substituting B for X in A . We extend this convention as follows. The notation $A\{X^+\}$ represents the assertion that X occurs *positively* (or not at all) in A . Similarly, $A\{X^-\}$ represents the assertion that X occurs *negatively* (or not at all) in A . Finally, $A\{X^0\}$ means that X occurs neither positively nor negatively in A . The following table defines the notation for variant occurrences.

Variant occurrences

$Y\{X^+\}$	whether $X = Y$ or $X \neq Y$
$Top\{X^+\}$	always
$[l_i:B_i]^{i \in 1..n}\{X^+\}$	if $X \notin \cup^{i \in 1..n} FV(B_i)$
$(A \rightarrow B)\{X^+\}$	if $A\{X^-\}$ and $B\{X^+\}$
$(\mu(Y)A)\{X^+\}$	if $X = Y$, or both $A\{Y^+\}$ and $A\{X^+\}$, or $X \notin FV(A)$
$(\forall(Y <: A)B)\{X^+\}$	if $X = Y$ or both $A\{X^-\}$ and $B\{X^+\}$
$(\exists(Y <: A)B)\{X^+\}$	if $X = Y$ or both $A\{X^+\}$ and $B\{X^+\}$
$Y\{X^-\}$	if $X \neq Y$
$Top\{X^-\}$	always
$[l_i:B_i]^{i \in 1..n}\{X^-\}$	if $X \notin \cup^{i \in 1..n} FV(B_i)$
$(A \rightarrow B)\{X^-\}$	if $A\{X^+\}$ and $B\{X^-\}$
$(\mu(Y)A)\{X^-\}$	if $X = Y$, or both $A\{Y^+\}$ and $A\{X^-\}$, or $X \notin FV(A)$

$(\forall(Y<:A)B)\{X^-\}$	if $X = Y$ or both $A\{X^+\}$ and $B\{X^-\}$
$(\exists(Y<:A)B)\{X^-\}$	if $X = Y$ or both $A\{X^-\}$ and $B\{X^-\}$
$A\{X^0\}$	if neither $A\{X^+\}$ nor $A\{X^-\}$

Notation

- We write $A\{X^+, Y^-\}$ to mean that both $A\{X^+\}$ and $A\{Y^-\}$.
- We may use $A\{X^+\}$ as a type, and then it stands for A together with the assertion $A\{X^+\}$. We may similarly use $A\{X^+, Y^-\}$, $A\{X^-\}$, and $A\{X^0\}$ as types. For example, $E \vdash a : B'\{X^-\} \rightarrow B''\{X^+\}$ means that $E \vdash a : B' \rightarrow B''$ and that the assertions $B'\{X^-\}$ and $B''\{X^+\}$ hold.

The following lemmas establish that positive occurrence is a sufficient condition for covariance, and negative occurrence for contravariance. They are proved in Appendix C.1.

Lemma 13.3-1

Assume $E'\{Y^+, X^-\}$ and $B\{X^+\}$.

If $E, X <: A, E'\{X, X\} \vdash B\{X\}$, then $E, X <: A, E'\{X, A\} \vdash B\{X\} <: B\{A\}$.

□

Lemma 13.3-2

Assume $B\{Y^+, X^-\}$.

If $E, X <: A \vdash B\{X, X\}$ and $E \vdash C_1 <: C_2$ and $E \vdash C_2 <: A$, then $E \vdash B\{C_1, C_2\} <: B\{C_2, C_1\}$.

□

13.4 Variant Product and Function Types, Encoded in $\mathbf{Ob}_{<:}$

Within the first-order object calculus $\mathbf{Ob}_{1<:}$, we were unable to find a full translation of λ -calculi with subtyping, because the variance properties of \rightarrow could not be obtained with invariant object types only (Section 8.1). One solution was to add variance annotations (Section 8.7).

Now we have at our disposal bounded universal types, which are contravariant in their bounds, and bounded existential types, which are covariant in their bounds. Combining these variant constructs with invariant object types, we can define a variant version of function types as follows:

$$A \rightarrow^{\forall 3} B \triangleq \forall(X <: A) \exists(Y <: B)[arg:X, val:Y]$$

We obtain $A \rightarrow^{\forall 3} B <: A' \rightarrow^{\forall 3} B'$ if $A' <: A$ and $B <: B'$.

This idea gives rise to an encoding of $\mathbf{F}_{1<:}$ into $\mathbf{Ob}_{<:}$. As before, this encoding is defined on typing derivations, but for simplicity we write it as a translation of type-annotated λ -terms.

Translation of the first-order λ -calculus with subtyping

$\rho \in Var \rightarrow \mathbf{Ob}_{\leq:\mu}\text{-term}$

$$\begin{aligned} \rho(x) &\triangleq x \\ (\rho(y \leftarrow a))(x) &\triangleq \text{if } x = y \text{ then } a \text{ else } \rho(x) \\ \langle\emptyset\rangle &\triangleq \emptyset \\ \langle E, x:A \rangle &\triangleq \langle E \rangle, x:\langle A \rangle \\ \langle K \rangle &\triangleq (\text{any closed type}) \\ \langle A \rightarrow B \rangle &\triangleq \forall(X \leq:\langle A \rangle)\exists(Y \leq:\langle B \rangle)[arg:X, val:Y] \\ \langle x_A \rangle_\rho &\triangleq \rho(x) \\ \langle b_{A \rightarrow B}(a_A) \rangle_\rho &\triangleq \\ &\quad \text{open } \langle b \rangle_\rho(\langle A \rangle) \text{ as } Y \leq:\langle B \rangle, y:[arg:\langle A \rangle, val:Y] \\ &\quad \text{in } (y.\text{arg} = \zeta(x:[arg:\langle A \rangle, val:Y])\langle a \rangle_\rho).\text{val} : \langle B \rangle \quad \text{for } y, x \notin FV(\langle a \rangle_\rho) \\ \langle \lambda(x:A)b_B \rangle_\rho &\triangleq \\ &\quad \lambda(X \leq:\langle A \rangle) \\ &\quad (\text{pack } Y \leq:\langle B \rangle = \langle B \rangle \text{ with} \\ &\quad \quad [arg = \zeta(x:[arg:X, val:\langle B \rangle])x.\text{arg}, \\ &\quad \quad val = \zeta(x:[arg:X, val:\langle B \rangle])\langle b \rangle_\rho|_{x \leftarrow x.\text{arg}}] \\ &\quad \quad : [arg:X, val:Y]) \end{aligned}$$

This translation can be extended trivially to recursive types and to second-order quantifiers. Hence our largest calculus, $\mathbf{FOb}_{\leq:\mu}$ can be embedded inside $\mathbf{Ob}_{\leq:\mu}$.

We can obtain covariant product types, since these can be represented in terms of universal quantifiers and variant function types in \mathbf{F}_{\leq} . However, a more direct encoding is available:

$$A \times^{\exists} B \triangleq \exists(X \leq:A)\exists(Y \leq:B)[fst:X, snd:Y]$$

We obtain $A \times^{\exists} B \leq: A' \times^{\exists} B'$ if $A \leq: A'$ and $B \leq: B'$.

In [40] it is shown that record types $\langle l_i; A_i \rangle_{i \in 1..n}$ can be represented in \mathbf{F}_{\leq} , using covariant product types. These record types are covariant, in the sense that $\langle l_i; A_i \rangle_{i \in 1..n+m} \leq: \langle l_i; A'_i \rangle_{i \in 1..n}$ if $A_i \leq: A'_i$ for $i \in 1..n$. We can therefore encode covariant record types in \mathbf{Ob}_{\leq} .

In sum, these encodings confirm the expressiveness of \mathbf{Ob}_{\leq} . We are able to represent function, product, and record types, with their desired variance properties, in terms of the most basic object types and standard bounded quantifiers.

13.5 The Self Quantifier

Within the second-order ζ -calculus with bounded quantifiers and recursion, $\mathbf{Ob}_{\leq:\mu}$ we can form an interesting construct that we call the Self quantifier. In this section we study the formal properties of the Self quantifier independently of objects. In a later

chapter we show how the Self quantifier interacts with object types, and how it helps in representing the type Self discussed in Sections 2.8 and 9.6.

13.5.1 Defining the Self Quantifier

The Self quantifier ζ is a combination of recursion and bounded existentials, with recursion going “through the bound”:

$$\zeta(X)B \triangleq \mu(Y)\exists(X <: Y)B \quad (Y \text{ not occurring in } B)$$

Given a type $B\{X\}$, any type $B\{A\}$ can be transformed into a related type $\exists(Y <: A)B\{Y\}$ covariant in A . An analogous technique applies to recursive types, and motivates our definition of the Self quantifier. Given an equation $X = B\{X\}$, we transform it into $X = \exists(Y <: X)B\{Y\}$. The solution to this equation, $\zeta(X)B\{X\}$, satisfies the subtyping property:

$$\text{if } B\{X\} <: B'\{X\}, \text{ then } \zeta(X)B\{X\} <: \zeta(X)B'\{X\}$$

even though we may not have $\mu(X)B\{X\} <: \mu(X)B'\{X\}$. This property suggests that ζ is preferable to μ in some situations where subtyping is important, and it is the main motivation for the Self quantifier.

The elements of $\zeta(X)B\{X\}$ can be understood informally as pairs. Modulo an unfolding, $\zeta(X)B$ is the same as $\exists(X <: \zeta(X)B)B$. Hence by analogy with the interpretation of existential types, $\zeta(X)B\{X\}$ can be understood as the type of pairs $\langle C, c \rangle$ consisting of a subtype C of $\zeta(X)B\{X\}$ and an element c of $B\{C\}$.

For example, suppose we have an element x of type $\zeta(X)X$. Then, choosing $\zeta(X)X$ as the required subtype of $\zeta(X)X$, we obtain $\langle \zeta(X)X, x \rangle : \zeta(X)X$. Therefore we can construct:

$$\mu(x)\langle \zeta(X)X, x \rangle : \zeta(X)X$$

Less trivially, and still informally, suppose we want to define a type St of storage cells, and to build a storage cell $st : St$ having a read method get with result type Nat and a write method set with parameter type Nat and result type St . We can define:

$$St \triangleq \zeta(X)[get:Nat, set:Nat \rightarrow St]$$

where the set method should use its argument to update the get field. For convenience, we adopt the following abbreviation to unfold a Self quantifier.

Notation

- $A(C) \triangleq B\{C\}$ whenever $A \equiv \zeta(X)B\{X\}$ and $C <: A$.

So, for example, $St(St) \equiv [get:Nat, set:Nat \rightarrow St]$.

To define a storage cell, we are going to use twice the fact that if $x : St(St)$, then $\langle St, x \rangle : St$. (This follows simply by the typing rule for $\langle _, _ \rangle$ explained previously.) First we need a method body for set that, with self $s : St(St)$ and argument $n : Nat$, produces a result of type St . Since $s.get := n$ has the same type as s , namely $St(St)$, we can use $\langle St,$

$s.\text{get} := n : St$ as the body of the set method. Using this method body, we can build an object of type $St(St)$. Therefore we have:

$$st : St \triangleq \langle St, [get = 0, set = \zeta(s:St(St)) \lambda(n:Nat) \langle St, s.\text{get} := n \rangle] \rangle$$

13.5.2 Derived Rules for the Self Quantifier

Beyond these initial and rather informal examples, the best way to understand the Self quantifier is to look at its specific application to objects in Chapter 15. Still, it is important to examine the intrinsic properties of types of the form $\zeta(X)B$, because these properties help in deriving some complex rules for object types with Self types. The rest of this chapter is therefore dedicated to the study of the abstract properties of types of the form $\zeta(X)B$.

The two basic operations for the Self quantifier are similar to the ones for the existential quantifier. One operation constructs an element of $\zeta(X)B$, given a subtype of $\zeta(X)B$ and an appropriate value; it is the composition of *pack* for existentials and *fold* for recursive types. The other operation inspects an element of $\zeta(X)B$ (as much as possible) and computes with its contents; it is the composition of *unfold* for recursive types and *open* for existentials.

The operation for constructing elements of type $A \equiv \zeta(X)B\{X\}$, in full generality, binds a type variable. Hence we need a more complex syntax than the pairing $\langle -, - \rangle$ used above. We refine the notation $\langle C, b \rangle$ to *wrap*($Y <: A = C$) b .

The term *wrap*($Y <: A = C$) b binds C to Y in b , and requires C to be a subtype of A . We often choose $C = A$, but the ability to choose $C <: A$ is important, particularly when C is a variable. (Section 15.5 shows that this generality has a price.) Within b , the types Y and C are equivalent. The type of the whole term is A .

The term *use* c as $Y <: A, y:B\{Y\}$ in $d:D$ decomposes a term c of type A into its two components, naming them Y and y , respectively. The type variable Y is a subtype of A , and the term variable y has type $B\{Y\}$. These variables are used in the term d of type D . The type D cannot contain occurrences of Y .

We define:

The Self quantifier

$$\zeta(X)B \triangleq \mu(Y)\exists(X <: Y)B \\ (\text{with } Y \notin FV(B))$$

$$\text{wrap}(Y <: A = C)b(Y) \triangleq \text{fold}(A, \text{pack } Y <: A = C \text{ with } b(Y):B\{Y\}) \\ (\text{with } A \equiv \zeta(X)B\{X\}, C <: A, b[C] : B\{C\})$$

$$\text{use } c \text{ as } Y <: A, y:B\{Y\} \text{ in } d\{Y,y\}:D \triangleq \text{open } \text{unfold}(c) \text{ as } Y <: A, y:B\{Y\} \text{ in } d\{Y,y\}:D \\ (\text{with } A \equiv \zeta(X)B\{X\}, c : A, d\{Y,y\} : D, Y \notin FV(D))$$

Now the following rules for the Self quantifier can be derived from the rules for μ and for \exists .

Δ_ζ

(Type Self) $E, X <: Top \vdash B$ $E \vdash \zeta(X)B$	(Sub Self) $E, X <: Top \vdash B <: B'$ $E \vdash \zeta(X)B <: \zeta(X)B'$
(Val Wrap) (where $A \equiv \zeta(X)B\{X\}$) $E \vdash C <: A$ $E \vdash b[C] : B\{C\}$ $E \vdash wrap(Y <: A = C)b\{Y\} : A$	(Val Use) (where $A \equiv \zeta(X)B\{X\}$) $E \vdash c : A$ $E \vdash D$ $E, Y <: A, y : B\{Y\} \vdash d : D$ $E \vdash use c \text{ as } Y <: A, y : B\{Y\} \text{ in } d : D : D$

 $\Delta_{=\zeta}$

(Eq Wrap) (where $A \equiv \zeta(X)B\{X\}$, $A' \equiv \zeta(X)B'\{X\}$) $E \vdash C <: A'$ $E \vdash A$ $E, Y \vdash B'\{Y\} <: B\{Y\}$ $E \vdash b[C] \leftrightarrow b'[C] : B'\{C\}$
$E \vdash wrap(Y <: A = C)b\{Y\} \leftrightarrow wrap(Y <: A' = C)b'\{Y\} : A$

(Eq Use) (where $A \equiv \zeta(X)B\{X\}$) $E \vdash c \leftrightarrow c' : A$ $E \vdash D$ $E, Y <: A, y : B\{Y\} \vdash d \leftrightarrow d' : D$
$E \vdash use c \text{ as } Y <: A, y : B\{Y\} \text{ in } d : D \leftrightarrow use c' \text{ as } Y <: A, y : B\{Y\} \text{ in } d' : D : D$

(Eval Unwrap) (where $A \equiv \zeta(X)B\{X\}$, $c \equiv wrap(Z <: A = C)b\{Z\}$) $E \vdash c : A$ $E \vdash D$ $E, Y <: A, y : B\{Y\} \vdash d\{Y, y\} : D$
$E \vdash use c \text{ as } Y <: A, y : B\{Y\} \text{ in } d\{Y, y\} : D \leftrightarrow d\{C, b[C]\} : D$

(Eval Rewrap) (where $A \equiv \zeta(X)B\{X\}$) $E \vdash b : A$ $E, y : A \vdash d[y] : D$
$E \vdash use b \text{ as } Y <: A, y : B\{Y\} \text{ in } d\{wrap(Y <: A = Y)\} : D \leftrightarrow d\{b\} : D$

Notation

- $wrap(A, c)$ stands for $wrap(X <: A = A)c$ when X does not occur in c .
- $wrap(X = A)c\{X\}$ stands for $wrap(X <: A = A)c\{X\}$.

Note the important rule of covariance, (Sub Self). This rule holds because existential types are covariant in their bounds, and because recursion in $\mu(Y)\exists(X <: Y)B\{X\}$ involves only a single covariant position. This rule can be verified as follows:

$E, X \vdash B <: B'$	antecedent
$E, Z, Y <: Z, X <: Y \vdash B <: B'$	by weakening lemmas, for fresh Y, Z
$E, Z, Y <: Z \vdash \exists(X <: Y)B <: \exists(X <: Z)B'$	by (Sub Exists)
$E \vdash \mu(Y)\exists(X <: Y)B <: \mu(Z)\exists(X <: Z)B'$	by (Sub Rec)

The other rules are roughly analogous to those for μ and for \exists . The derivation of (Eq Wrap) relies on (Eq Fold $<$: Lemma2) of Section 9.2. Much as we have done for the (Eq Pack $<$) rule for \exists , we derive a flexible congruence rule from (Eq Wrap):

$$\begin{array}{c} \text{(Eq Wrap Lemma)} \quad (\text{where } A \equiv \zeta(X)B[X], A' \equiv \zeta(X)B'[X]) \\ E \vdash C <: A' \quad E \vdash A \quad E, Y \vdash B'\{Y\} <: B\{Y\} \\ E \vdash b[C] \leftrightarrow b'[C] : B[C] \quad E \vdash b'[C] : B'[C] \\ \hline E \vdash \text{wrap}(Y <: A = C)b[Y] \leftrightarrow \text{wrap}(Y <: A' = C)b'[Y] : A \end{array}$$

The storage cell definition of Section 13.5.1 can now be understood formally. We rewrite the example taking advantage of the new notation for wrapping:

$$\begin{aligned} St &\triangleq \zeta(\text{Self})[\text{get:Nat}, \text{set:Nat} \rightarrow \text{Self}] \\ st : St &\triangleq \text{wrap}(\text{Self}=St)[\text{get} = 0, \text{set} = \zeta(s:St(\text{Self})) \lambda(n:\text{Nat}) \text{wrap}(\text{Self}, s.\text{get}:=n)] \end{aligned}$$

Later examples frequently adopt this choice of *Self* as a name for a variable bound by the *Self* quantifier.

13.5.3 A Pure Second-Order Object Calculus

We conclude this section by considering a pure second-order object calculus, ζOb , based exclusively on object types and the *Self* quantifier. (See Appendix B.3.)

$$\zeta\text{Ob} \triangleq \Delta_x \cup \Delta_{\text{Ob}} \cup \Delta_{<} \cup \Delta_{<:x} \cup \Delta_{<:\text{Ob}} \cup \Delta_\zeta$$

The syntax of this calculus is summarized in the following table.

Syntax of the ζOb calculus

$A, B ::=$	types
X	type variable
Top	the biggest type
$[l_i:B_i \ i \in 1..n]$	object type (l_i distinct)
$\zeta(X)B$	Self quantified type
$a, b ::=$	terms
x	variable
$[l_i=\zeta(x_i:A_i)b_i \ i \in 1..n]$	object (l_i distinct)
$a.l$	method invocation
$a.l \Leftarrow \zeta(x)b$	method update
$\text{wrap}(X <: A = B)b$	wrap
$\text{use } a \text{ as } X <: A, y: B \text{ in } b: D$	use

This calculus departs from second-order λ -calculi by omitting function types and the standard quantifiers. It seems that, in ζOb , the only function types that can be encoded are invariant, and that the standard second-order quantifiers are not expressible. Still,

as Chapter 15 demonstrates, the Self quantifier provides an essential second-order feature of object calculi, namely, the Self type. Interestingly, then, ζOb is a small second-order object calculus that covers a spectrum of object-oriented notions.

14 A SEMANTICS

In Section 6.7 we reviewed two syntactic interpretations of objects in terms of records. In one of them, the self-application semantics, an object is a record of functions; correspondingly, an object type is interpreted as a recursive record type. For example, the object type $\text{Point} \triangleq [x,y:\text{Int}]$ is interpreted as the record type obtained by solving the type equation $\text{Point} = \langle x,y:\text{Point} \rightarrow \text{Int} \rangle$. Unfortunately, this type does not include the solution of $\text{ColorPoint} = \langle x,y:\text{ColorPoint} \rightarrow \text{Int}, c:\text{ColorPoint} \rightarrow \text{Color} \rangle$, which is the interpretation of $\text{ColorPoint} \triangleq [x,y:\text{Int}, c:\text{Color}]$. Therefore, the self-application semantics does not validate subtypings such as $\text{ColorPoint} <: \text{Point}$.

The denotational semantics that we give in this chapter can be understood as a variant of the self-application semantics. Specifically, we interpret the type Point as the union of all the solutions to the equations of the form $X = \langle x,y:X \rightarrow \text{Int}, \dots \rangle$, including, for example, $X = \langle x,y:X \rightarrow \text{Int} \rangle$ and $X = \langle x,y:X \rightarrow \text{Int}, c:X \rightarrow \text{Color} \rangle$. With this definition, ColorPoint is a subtype of Point . The semantics is based on the simple idea just described. On the other hand, the details of the semantics are fairly intricate, and understanding them requires some familiarity with denotational techniques; the casual reader should be able to skip those details.

We describe a semantics for the largest calculus considered in Chapter 13, $\mathbf{FOb}_{\leq:\mu}$. The semantics provides a justification for our type theories and our equational theories, and to some extent guided us in their choice.

The semantics is based on a metric approach [82]. Following Amadio and Cardone, we interpret types as complete uniform pers, and in particular we interpret recursive types as fixpoints of contractive functions on complete uniform pers [17, 46]. It might be possible to obtain a semantics for $\mathbf{FOb}_{\leq:\mu}$ with standard O -categorical methods [112] instead of metric methods. However, even the fragment $F_{\leq:\mu}$ poses problems, as noted in [10]. We expect object types to give rise to further interesting difficulties.

14.1 The Untyped Universe

We interpret our calculus in an untyped universe: we map terms to elements of a single set irrespectively of their types. The assumptions on the universe are fairly standard and technical (see, e.g., [10, 17, 46, 47, 82]). Essentially, we assume a complete partial order D that contains a value $*$ and that includes the continuous function space ($D \rightarrow D$) and the function space ($L \rightarrow D$), where L is a countable set of labels $\{m_0, m_1, \dots\}$. We use $*$ to represent an error, a function in $(L \rightarrow D)$ to represent a record, and a record of functions to represent an object.

A suitable D can be constructed by applying the usual “limit of a sequence of iterates” method to solve the domain equation $D = W + (D \rightarrow D) + (L \rightarrow D)_{\perp}$, where:

- W is the set $\{\perp, *\}$ with the order $\perp \leq *$;
- $(D \rightarrow D)$ is the complete partial order of continuous functions from D to D ;
- $(L \rightarrow D)$ is the complete partial order of functions from L to D , and $(L \rightarrow D)_{\perp}$ is its lifting;
- $+$ is the coalesced sum operator, so the least elements of the summands are identified;
- the equation is solved over complete partial orders, up to isomorphism: the solution D is a complete partial order isomorphic to $W + (D \rightarrow D) + (L \rightarrow D)_{\perp}$.

It is important for our purposes to make sure that at each stage of the “sequence of iterates” construction we have a finite set. Therefore the iterate D_{i+1} should not contain the full $(L \rightarrow D_i)_{\perp}$ but only $(L_i \rightarrow D_i)_{\perp}$ where L_i is a finite subset of L ; we take $L_i = \{m_0, \dots, m_i\}$. Thus D_0 is $\{\perp\}$ and D_{i+1} is $W + (D_i \rightarrow D_i) + (L_i \rightarrow D_i)_{\perp}$. The solution D is obtained from D_0, D_1, \dots by a limiting process.

We do not detail the construction, but instead list some essential requirements. We need to have a partial order (D, \sqsubseteq) such that:

- (D, \sqsubseteq) is a complete partial order: \perp is its least element and every sequence $\langle x_i \rangle$ such that $x_0 \sqsubseteq x_1 \sqsubseteq \dots \sqsubseteq x_i \sqsubseteq \dots$ has a least upper bound $\sqcup_i x_i$.
- There are strict, continuous embedding-retraction pairs $(e_1, r_1), (e_2, r_2)$, and (e_3, r_3) , between D and W , $(D \rightarrow D)$, and $(L \rightarrow D)_{\perp}$, respectively.

$$\begin{array}{ccccc} & e_1 & & r_1 & \\ W & \xrightarrow{\quad} & D & \xrightarrow{\quad} & W \\ (D \rightarrow D) & \xrightarrow{\quad} & D & \xrightarrow{\quad} & (D \rightarrow D) \\ (L \rightarrow D)_{\perp} & \xrightarrow{\quad} & D & \xrightarrow{\quad} & (L \rightarrow D)_{\perp} \end{array}$$

- There is an increasing sequence $p_n : D \rightarrow D$ of continuous projections with finite range and with least upper bound the identity. Further, p_0 constantly equals \perp .
- Let ; denote function composition. Let $\restriction{h}{L_i}$ denote function restriction (so $(h \restriction{L_i})(m)$ is $h(m)$ if $m \in L_i$ and is \perp otherwise). For all i ,

$$\begin{aligned} p_{i+1}(e_1(*)) &= e_1(*) \\ p_{i+1}(e_2(f)) &= e_2(p_i; f; p_i) & \text{for } f \in D \rightarrow D \\ p_{i+1}(e_3(o)) &= e_3((o; p_i) \restriction{L_i}) & \text{for } o \in L \rightarrow D \end{aligned}$$

Thus, omitting e 's and r 's, $p_{i+1}(*)$ is $*$; $p_{i+1}(f)(x)$ is $p_i(f(p_i(x)))$; and $p_{i+1}(o)(m_j)$ is $p_i(o(m_j))$ if $j \leq i$ and \perp otherwise.

If $x \in D$ is such that $p_n(x) = x$ for some n , then x is *finite*, and the least n for which $p_n(x) = x$ is the *rank* of x . If x is finite and $\langle y_i \rangle$ is an increasing sequence, then $x \sqsubseteq \sqcup_i y_i$ implies that $x \sqsubseteq y_k$ for some k .

We commonly view W , $(D \rightarrow D)$, and $(L \rightarrow D)$ as subsets of D , and omit the various e 's and r 's. We may write $*$ for $e_1(*)$, $x \in (D \rightarrow D)$ for $x \in e_2(D \rightarrow D)$, and $x \in (L \rightarrow D)$ for $x \in e_3(L \rightarrow D)$. When $x, y \in D$ and $m \in L$, we write $x(y)$ for $r_2(x)(y)$; similarly, we write $x(m)$ for $r_3(x)(m)$ if $r_3(x) \in (L \rightarrow D)$ and for \perp if $r_3(x) = \perp$. Further, we use λ -notation for elements of $(D \rightarrow D)$; and we write $\langle\langle m_1=x_1, \dots, m_n=x_n \rangle\rangle$ for the function in $(L \rightarrow D)$ that maps m_1 to x_1, \dots, m_n to x_n , and maps all other labels to $*$. When f is a function, we write $f(l \leftarrow x)$ for the function that maps l to x and is identical to f elsewhere.

14.2 Types in the Untyped Universe

Having described an untyped universe, we view the types as certain binary relations on this untyped universe. Intuitively, if A is a type and R_A is the associated relation, then $(x, y) \in R_A$ means that x and y are equal elements of A . Subtyping is simply interpreted as relation containment.

After some preliminaries, Section 14.2.2 introduces several operations on binary relations and Section 14.2.3 discusses subtyping. Section 14.2.4 provides lemmas about the union of binary relations, and Section 14.2.5 provides some results about metric properties.

14.2.1 Preliminaries

When we discuss binary relations over D , by convention we always mean binary relations that do not have $*$ in their domains. It is easy to show that all our constructions preserve this property.

A per (partial equivalence relation) is a symmetric, transitive, binary relation on D . A binary relation P is uniform if $x P y$ implies $(p_i(x)) P (p_i(y))$ for all i . It is closed under limits of increasing sequences if, whenever $\langle x_i \rangle$ and $\langle y_i \rangle$ are increasing sequences and $x_i P y_i$ for all i , then $(\sqcup_i x_i) P (\sqcup_i y_i)$. It is complete if it is closed under limits of increasing sequences and $\perp P \perp$.

A cuper is a complete uniform per. The set of all cupers is **CUPER**. Below, all types are interpreted as pers in **CUPER**. Throughout, the variables R , S , and T range over **CUPER**, and P and Q range over arbitrary binary relations.

14.2.2 Semantic Definitions for Type Constructors

First we describe some usual constructions on pers.

The largest cuper in which we are interested is *Univ*:

$$\text{Univ} \triangleq (D - \{*\}) \times (D - \{*\})$$

It relates any two elements different from $*$.

The function-space operation is given by:

$$P \rightarrow Q \triangleq \{(f,g) \in (D \rightarrow D) \times (D \rightarrow D) \mid \text{for all } x, y, \text{ if } xPy, \text{ then } f(x)Qg(y)\}$$

If $P, Q \in \mathbf{CUPER}$ then $P \rightarrow Q \in \mathbf{CUPER}$.

We can calculate meets and joins for any family of cupers:

$$\begin{aligned}\sqcap_{i \in I} P_i &\triangleq \cap_{i \in I} P_i \\ \sqcup_{i \in I} P_i &\triangleq C(\cup_{i \in I} P_i)\end{aligned}$$

where $C(P)$ is the least cuper that contains P ; the cuper $C(P)$ is always defined since the conditions that describe cupers are monotone closure conditions. If $P_i \in \mathbf{CUPER}$ for all $i \in I$, then $\sqcap_{i \in I} P_i \in \mathbf{CUPER}$ and $\sqcup_{i \in I} P_i \in \mathbf{CUPER}$.

A pair (x,y) is finite if each of x and y is finite; the rank of (x,y) is the greatest of the ranks of x and y . The restriction of a cuper R to elements of rank no greater than r is $p_r(R) \triangleq \{(x,y) \in R \mid \text{rank of } (x,y) \leq r\}$. A cuper is determined by its finite elements: by uniformity and completeness, if $p_r(R) = p_r(T)$ for all r , then $R = T$. The distance between two cupers is 2^{-r} , where r is the minimum rank where the two cupers differ, and it is 0 if the two cupers are equal:

$$\text{distance}(R,T) \triangleq \max(\{0\} \cup \{2^{-r} \mid p_r(R) \neq p_r(T)\})$$

The set of all cupers with this distance function is a complete metric space (in fact, an ultrametric space). In this complete metric space, F is a nonexpansive function if $\text{distance}(F(R), F(T)) \leq \text{distance}(R, T)$ for all R and T ; that is, F maps any two cupers that are equal up to rank r to two cupers that are equal up to rank r . Similarly, F is a contractive function if $\text{distance}(F(R), F(T)) \leq \text{distance}(R, T)/2$ for all R and T ; that is, F maps any two cupers that are equal up to rank r to two cupers that are equal up to rank $r+1$. Amadio [17] has verified that the usual type constructors \rightarrow and μ are contractive or nonexpansive on cupers as on ideals [82]; in Section 14.2.5 we review some of Amadio's results and prove other similar ones.

By the Banach Fixpoint Theorem, if F is a contractive function, then it has a unique fixpoint; this gives rise to an interpretation of recursive types as follows. If F is a contractive function, then for any S the sequence $S, F(S), \dots, F^n(S), \dots$ converges; moreover, the distances between consecutive elements get uniformly smaller:

$$\text{distance}(F^n(S), F^{n+1}(S)) \leq \text{distance}(S, F(S))/2^n \leq 1/2^{n+1}$$

Therefore, the sequence has a limit F^∞ . This limit is the unique fixpoint of F (independently of the choice of S):

$$\begin{aligned}\text{distance}(F(F^\infty), F^\infty) &\leq \max(\text{distance}(F(F^\infty), F^{n+1}(S)), \text{distance}(F^{n+1}(S), F^\infty)) \\ &\leq \max(\text{distance}(F^\infty, F^n(S)), \text{distance}(F^{n+1}(S), F^\infty)) \\ &\leq \text{distance}(F^\infty, F^n(S)) \\ &\leq 1/2^{n+1} \quad \text{for every } n\end{aligned}$$

and hence $\text{distance}(F(F^\infty), F^\infty) = 0$. We write $\mu(S)F(S)$ for this unique fixpoint.

In order to give a semantics to object types, we first define:

$$\langle\langle m_i; T_i^{ieI} \rangle\rangle \triangleq \{(\perp, \perp)\} \cup \{(o, o') \in (L \rightarrow D) \times (L \rightarrow D) \mid \forall i \in I. (o(m_i), o'(m_i)) \in T_i\}$$

The type $\langle\langle m_i; T_i^{ieI} \rangle\rangle$ can be viewed as a record type, with fields m_i and types T_i . If $T_i \in \mathbf{CUPER}$ for all $i \in I$ then $\langle\langle m_i; T_i^{ieI} \rangle\rangle \in \mathbf{CUPER}$. Uniformity holds because, if $(o, o') \in \langle\langle m_i; T_i^{ieI} \rangle\rangle$, then $(p_0(o), p_0(o')) = (\perp, \perp) \in \langle\langle m_i; T_i^{ieI} \rangle\rangle$, and either $(p_{j+1}(o)(m_i), p_{j+1}(o')(m_i)) = (p_j(o(m_i)), p_j(o'(m_i)))$ or $(p_{j+1}(o)(m_i), p_{j+1}(o')(m_i)) = (\perp, \perp)$, so $(p_{j+1}(o), p_{j+1}(o')) \in \langle\langle m_i; T_i^{ieI} \rangle\rangle$. Moreover, as we prove in Section 14.2.5, $\langle\langle m_i; T_i^{ieI} \rangle\rangle$ is contractive in each T_i .

We call **Gen** ("generators") the set of functions on cupers of the form $\lambda(S)\langle\langle m_i; S \rightarrow T_i^{ieI} \rangle\rangle$. Such a function can be written in the form $\lambda(S)\langle\langle m_i; S \rightarrow T_i^{ieI} \rangle\rangle$ uniquely:

Proposition 14.2-1

If $\langle\langle m_i; S \rightarrow T_i^{ieI} \rangle\rangle = \langle\langle m_j; S \rightarrow R_j^{jeI} \rangle\rangle$ for some cuper S , then $I = J$ and $T_i = R_i$ for all $i \in I$.

Proof

If $j \notin I$ but $j \in J$, then there exists x such that $x(m_j) = *$, $(x, x) \in \langle\langle m_i; S \rightarrow T_i^{ieI} \rangle\rangle$ but $(x, x) \notin \langle\langle m_j; S \rightarrow R_j^{jeI} \rangle\rangle$; for example, let x be $\langle\langle m_j = *, m_k = \perp^{k \neq j} \rangle\rangle$. If $I = J$ but $(x, y) \notin T_i$ and $(x, y) \in R_i$, then $(\lambda(z)x, \lambda(z)y) \notin S \rightarrow T_i$ (since S must be nonempty) while $(\lambda(z)x, \lambda(z)y) \in S \rightarrow R_i$, hence $\langle\langle m_i = \lambda(z)x, m_k = \perp^{k \neq i} \rangle\rangle, \langle\langle m_i = \lambda(z)y, m_k = \perp^{k \neq i} \rangle\rangle \notin \langle\langle m_i; S \rightarrow T_i^{ieI} \rangle\rangle$ while $\langle\langle m_i = \lambda(z)x, m_k = \perp^{k \neq i} \rangle\rangle, \langle\langle m_i = \lambda(z)y, m_k = \perp^{k \neq i} \rangle\rangle \in \langle\langle m_j; S \rightarrow R_j^{jeI} \rangle\rangle$.

□

We say that $\lambda(S)\langle\langle m_i; S \rightarrow T_i^{ieI} \rangle\rangle$ extends $\lambda(S)\langle\langle m_j; S \rightarrow T_j^{jeI} \rangle\rangle$ if $I \supseteq J$, and write:

$$\lambda(S)\langle\langle m_i; S \rightarrow T_i^{ieI} \rangle\rangle \preccurlyeq \lambda(S)\langle\langle m_j; S \rightarrow T_j^{jeI} \rangle\rangle$$

Using this relation on **Gen**, we define a new construction on cupers:

$$[[m_i; T_i^{ieI}]] \triangleq \sqcup \{\mu(S)F(S) \mid F \in \mathbf{Gen}, F \preccurlyeq \lambda(S)\langle\langle m_i; S \rightarrow T_i^{ieI} \rangle\rangle\}$$

The type $[[m_i; T_i^{ieI}]]$ can be viewed as an object type, with methods m_i and result types T_i . The definition is proper because if $F \in \mathbf{Gen}$ then $F(S)$ is contractive in S , and hence $\mu(S)F(S)$ exists and is unique. We explain this definition at some length in the next section; later we show that $[[m_i; T_i^{ieI}]]$ is contractive in each T_i .

14.2.3 Inclusion and Subtyping

Our intent is to interpret subtyping as inclusion of cupers. We now consider some of the features of our semantic definitions of the previous section from the point of view of subtyping.

The properties of \mathbf{Univ} , \rightarrow , \sqcap , and \sqcup are the expected ones. Specifically, \mathbf{Univ} is the largest cuper without * in its domain; $R \rightarrow T$ is contravariant in R and covariant in T ; $\sqcap_{i \in I} R_i$ is covariant in each R_i and contravariant in I ; $\sqcup_{i \in I} R_i$ is covariant in each R_i and in I . We have a less obvious inclusion property for fixpoints:

Proposition 14.2-2

Assume that F and G are contractive and that for every R and T if $R \subseteq T$ then $F(R) \subseteq G(T)$. Then $\mu(S)F(S) \subseteq \mu(S)G(S)$.

Proof

If $(x, x') \in \mu(S)F(S)$ then $(p_n(x), p_n(x')) \in \mu(S)F(S)$ by uniformity, and hence $(p_n(x), p_n(x')) \in F^m(S)$ for every S and for almost all m since $\text{distance}(\mu(S)F(S), F^m(S)) \leq 1/2^{m+1}$. Moreover, we obtain that $F^m(S) \subseteq G^m(S)$ by induction on m . Hence $(p_n(x), p_n(x')) \in G^m(S)$ for almost all m ; by completeness, $(x, x') \in \mu(S)G(S)$.

□

Finally, let us consider object types. In the framework of the self-application semantics, we may attempt to model object types as recursive record types, but then we fail to obtain all the expected subtypings. Our semantic approach is based on that attempt. Now, however, we have joins available, so we define an object type to be a join of recursive record types. We make the join large enough to obtain all the expected subtypings: each object type is designed to contain all longer object types. For example, we interpret the type *Point* as:

$$\sqcup \{\mu(S)F(S) \mid F \in \mathbf{Gen}, F \preccurlyeq \lambda(S)(\langle x, y : S \rightarrow \text{Int} \rangle)\}$$

and the type *ColorPoint* as:

$$\sqcup \{\mu(S)F(S) \mid F \in \mathbf{Gen}, F \preccurlyeq \lambda(S)(\langle x, y : S \rightarrow \text{Int}, c : S \rightarrow \text{Color} \rangle)\}$$

(Here we write *Int* and *Color* for their respective interpretations, informally.) Since $F \preccurlyeq \lambda(S)(\langle x, y : S \rightarrow \text{Int}, c : S \rightarrow \text{Color} \rangle)$ implies $F \preccurlyeq \lambda(S)(\langle x, y : S \rightarrow \text{Int} \rangle)$, we obtain that:

$$\begin{aligned} & \sqcup \{\mu(S)F(S) \mid F \in \mathbf{Gen}, F \preccurlyeq \lambda(S)(\langle x, y : S \rightarrow \text{Int}, c : S \rightarrow \text{Color} \rangle)\} \\ & \subseteq \sqcup \{\mu(S)F(S) \mid F \in \mathbf{Gen}, F \preccurlyeq \lambda(S)(\langle x, y : S \rightarrow \text{Int} \rangle)\} \end{aligned}$$

and thus we obtain that *ColorPoint* is a subtype of *Point*.

More generally, we have the following property:

Proposition 14.2-3

If $I \subseteq J$, then $[(m_j : T_j^{j \in J})] \subseteq [(m_i : T_i^{i \in I})]$.

Proof

All extensions of $\lambda(S)(\langle m_j : S \rightarrow T_j^{j \in J} \rangle)$ are also extensions of $\lambda(S)(\langle m_i : S \rightarrow T_i^{i \in I} \rangle)$, hence the result.

□

This proposition suffices for our purposes. We may already note, however, that we will not be able to account for “structural subtyping” of object types. Recall that in Section 13.1.3 we discussed an alternative rule for update:

$$\frac{\text{(Val Structural Update)} \quad (\text{where } A \equiv [l_i : B_i^{i \in 1..n}])}{E \vdash a : C \quad E \vdash C \lessdot A \quad E, x : C \vdash b : B_j \quad j \in 1..n} \frac{}{E \vdash a.l_j \Leftarrow_{\zeta} (x : C)b : C}$$

As we argued, this rule relies on the structural subtyping assumption that every subtype of an object type is an object type. This assumption does not hold in our model: it

is not necessarily the case that if R is included in $[(m_i; T_i^{i \in I})]$ then R can be expressed in the form $[(m_j; T_j^{j \in J})]$. For example, if $(o,o) \in [(m_i; T_i^{i \in I})]$, then R could be $C(\{(o,o)\})$. In light of this example, we are led to restrict R to range over cupers of the form $\sqcup \{\mu(S)F(S) \mid F \in \mathbf{Gen}, F \leq G\}$ for some $G \leq \lambda(S)(\langle m_i; S \rightarrow T_i^{i \in I} \rangle)$. This argument suggests that we may account for structural subtyping by focusing attention on cupers of the form $\sqcup \{\mu(S)F(S) \mid F \in \mathbf{Gen}, F \leq G\}$, and by saying that $\sqcup \{\mu(S)F(S) \mid F \in \mathbf{Gen}, F \leq G\}$ is a “structural subtype” of $\sqcup \{\mu(S)F(S) \mid F \in \mathbf{Gen}, F \leq H\}$ only if $G \leq H$. Although this definition seems reasonable, we do not use it because it requires a special treatment of object types and some amount of fairly syntactic argumentation.

14.2.4 Properties of Joins

In this section we obtain some necessary results about joins of relations. A join is defined with a closure operation C . The definition does not give a very explicit description of the elements of the join. In particular, it is not true that if $(x,y) \in S \sqcup T$ then either $(x,y) \in S$ or $(x,y) \in T$, and the definition of $S \sqcup T$ does not help much in pinning down what else (x,y) could be.

Here we analyze the closure operation. We obtain results that enable us to reason about all elements of a join by reasoning about the elements of the components of the join. Some of the other results, though more technical, are also important in the next section. The casual reader may wish to look only at Propositions 14.2-9 and 14.2-11.

We use some further notation for binary relations and sets of binary relations:

- P° : The restriction of P to finite elements.
- $T(P)$: The transitive closure of P .
- $U(P)$: The completion of P : $(x,y) \in U(P)$ if and only if either $(x,y) = (\perp,\perp)$ or $(x,y) = (\sqcup_i x_i, \sqcup_i y_i)$ for $\langle (x_i, y_i) \rangle$ an increasing sequence in P .
- **NUSR**: The set of nonempty, uniform, symmetric binary relations.
- **NUPER**: The set of nonempty, uniform pers.

Proposition 14.2-4

If $P \in \mathbf{NUSR}$, then $T(P) \in \mathbf{NUPER}$.

Proof

We check that $T(P) \in \mathbf{NUSR}$, and that in addition $T(P)$ is transitive.

- Transitive closure preserves nonemptiness.
- Transitive closure preserves symmetry.
- Transitive closure preserves uniformity: if $(x,y) \in T(P)$, then $x = z_0 P z_1 \dots z_{n-1} P z_n = y$ for some z_0, \dots, z_n , hence $p_i(x) = p_i(z_0) P p_i(z_1) \dots p_i(z_{n-1}) P p_i(z_n) = p_i(y)$ for all i , by the uniformity of P , so $(p_i(x), p_i(y)) \in T(P)$.
- Transitive closure establishes transitivity.

□

Proposition 14.2-5

If (x,y) is finite and $(x,y) \in U(P)$, then $(x,y) \in P$ or $(x,y) = (\perp,\perp)$.

Proof

Assume $(x,y) \in U(P)$ and $(x,y) \neq (\perp,\perp)$. Let $x = \sqcup_k x_k$ and $y = \sqcup_k y_k$, where $\langle (x_k, y_k) \rangle$ is an increasing sequence in P . Since (x,y) is finite, there exists k such that $(x,y) = (x_k, y_k)$.

□

Proposition 14.2-6

If P is a uniform binary relation, then so is $U(P)$.

Proof

Consider $(x,y) \in U(P)$. We wish to show that $(p_i(x), p_i(y)) \in U(P)$ for all i . In case $(x,y) = (\perp,\perp)$, this is trivial, as $(p_i(x), p_i(y)) = (\perp,\perp)$. Otherwise, let $x = \sqcup_k x_k$ and $y = \sqcup_k y_k$, where $\langle (x_k, y_k) \rangle$ is an increasing sequence in P . Clearly $(p_i(x), p_i(y)) \sqsubseteq (\sqcup_k x_k, \sqcup_k y_k)$, and since $(p_i(x), p_i(y))$ is finite there exists j such that $(p_i(x), p_i(y)) \sqsubseteq (x_j, y_j)$. By monotonicity and idempotence of p_i , $(p_i(x), p_i(y)) \sqsubseteq (p_i(x_j), p_i(y_j))$. On the other hand, $(x_j, y_j) \sqsubseteq (x,y)$, and hence $(p_i(x_j), p_i(y_j)) \sqsubseteq (p_i(x), p_i(y))$. Hence for all i there exists j such that $p_i(x) = p_i(x_j)$ and $p_i(y) = p_i(y_j)$. Since P is uniform and $(x_j, y_j) \in P$, it follows that $(p_i(x), p_i(y)) \in P$ for all i , and hence that $(p_i(x), p_i(y)) \in U(P)$ for all i .

□

Proposition 14.2-7

If P is a uniform binary relation, then $U(P)$ is complete.

Proof

Clearly, $(\perp,\perp) \in U(P)$. In addition, consider an increasing sequence in $U(P)$, $\langle (x_i, y_i) \rangle$, with limit (x,y) . We show that $(x,y) \in U(P)$ by finding an increasing sequence $\langle (x'_i, y'_i) \rangle$ in P with limit (x,y) .

The case of P empty is trivial. In case P is nonempty, let $(x'_i, y'_i) = (p_i(x_i), p_i(y_i))$. If $\langle (x_i, y_i) \rangle$ is increasing, then so is $\langle (p_i(x_i), p_i(y_i)) \rangle$. Since completion preserves uniformity by Proposition 14.2-6, $(x'_i, y'_i) \in U(P)$. Moreover, by Proposition 14.2-5, $(x'_i, y'_i) \in P$ (since $(\perp,\perp) \in P$ as P is nonempty).

Finally, we check that the limit of this constructed sequence is (x,y) . Clearly, $(\sqcup_i x'_i, \sqcup_i y'_i) \sqsubseteq (x,y)$, since $(x'_i, y'_i) \sqsubseteq (x_i, y_i)$ for all i and $(x,y) = (\sqcup_i x_i, \sqcup_i y_i)$. To show that $(x,y) \sqsubseteq (\sqcup_i x'_i, \sqcup_i y'_i)$, we prove that for all j there exists i such that $(p_j(x), p_j(y)) \sqsubseteq (x'_i, y'_i)$. By finiteness considerations, for all j there exists k such that $(p_j(x), p_j(y)) \sqsubseteq (x_k, y_k)$. Letting $i = \max(j, k)$, we obtain $(p_j(x), p_j(y)) \sqsubseteq (x_i, y_i)$, and then $(p_j(x), p_j(y)) \sqsubseteq (p_i(x_i), p_i(y_i)) = (x'_i, y'_i)$, the desired result.

□

Proposition 14.2-8

If $P \in \text{NUPER}$, then $\mathbf{U}(P) \in \text{CUPER}$.

Proof

We check that $\mathbf{U}(P) \in \text{NUPER}$, and that in addition $\mathbf{U}(P)$ is complete.

- Completion preserves nonemptiness.
- Completion preserves symmetry.
- Completion preserves uniformity by Proposition 14.2-6.
- Completion preserves transitivity in the presence of uniformity. If $(x,y), (y,z) \in \mathbf{U}(P)$, then $(p_i(x),p_i(y)), (p_i(y),p_i(z)) \in P$ for all i , by uniformity, and $(p_i(x),p_i(y)), (p_i(y),p_i(z)) \in P$ for all i by Proposition 14.2-5. By transitivity of P , $(p_i(x),p_i(z)) \in P$ for all i , and hence $(x,z) \in \mathbf{U}(P)$.
- Completion establishes completeness in the presence of uniformity by Proposition 14.2-7.

□

Proposition 14.2-9

Let $P \in \text{NUSR}$. Then $C(P) = \mathbf{U}(T(P^\circ))$.

Proof

Let $Q = P^\circ$; note that $Q \in \text{NUSR}$. First, $P \subseteq \mathbf{U}(T(Q))$, and in fact $P \subseteq \mathbf{U}(Q)$: if $(x,y) \in P$, then $(p_i(x),p_i(y)) \in P$ for all i by uniformity, and hence $(p_i(x),p_i(y)) \in Q$ for all i ; therefore $(x,y) \in \mathbf{U}(Q)$. Next, $\mathbf{U}(T(Q)) \in \text{CUPER}$ by Propositions 14.2-4 and 14.2-8. Finally, consider any other cuper R such that $P \subseteq R$. We show that $\mathbf{U}(T(Q)) \subseteq R$. Since U and T are monotonic and $\mathbf{U}(T(R)) = R$, we have that $\mathbf{U}(T(P)) \subseteq R$. So it suffices to remark that $\mathbf{U}(T(Q)) \subseteq \mathbf{U}(T(P))$.

□

Proposition 14.2-10

Let $P, Q \in \text{NUSR}$. Assume that f and g are continuous functions such that $(f(x),f(y)) \in Q$ and $(f(x),g(y)) \in Q$ for all $(x,y) \in P$. Then $(f(x),g(y)) \in C(Q)$ for all $(x,y) \in C(P)$.

Proof

Assume that (x,y) is finite. First we argue that if $(x,y) \in C(P)$, then for some z_0, \dots, z_n , we have $x = z_0 P z_1 \dots z_{n-1} P z_n = y$. By Proposition 14.2-9, $C(P) = \mathbf{U}(T(P^\circ))$; so if $(x,y) \in C(P)$, then $(x,y) \in T(P^\circ)$ by Proposition 14.2-5 (since clearly $(\perp,\perp) \in P^\circ$). So there exist z_0, \dots, z_n such that $x = z_0 P^\circ z_1 \dots z_{n-1} P^\circ z_n = y$, and hence $x = z_0 P z_1 \dots z_{n-1} P z_n = y$. By hypothesis, $f(x) = f(z_0) Q f(z_1) \dots f(z_{n-1}) Q g(z_n) = g(y)$. Since $Q \subseteq C(Q)$ and $C(Q)$ is transitive, it follows that $(f(x),g(y)) \in C(Q)$.

Now consider (x,y) not necessarily finite. If $(x,y) \in C(P)$, then for all i , $(p_i(x),p_i(y)) \in C(P)$, by uniformity. Now the argument for finite elements yields $(f(p_i(x)),g(p_i(y)))$

$\in C(Q)$. Since f and g are continuous, $(f(x), g(y)) = (\sqcup_i f(p_i(x)), \sqcup_i g(p_i(y)))$. By completeness it follows that $(f(x), g(y)) \in C(Q)$.

□

The following proposition entails that, if $R_i \in \mathbf{CUPER}$ for all $i \in I$, and $S \in \mathbf{CUPER}$, and f and g are continuous functions such that $(x, y) \in R_i$ implies $(f(x), g(y)) \in S$, then $(x, y) \in \sqcup_{i \in I} R_i$ implies $(f(x), g(y)) \in S$.

Proposition 14.2-11

If $R_i \in \mathbf{CUPER}$ for all $i \in I$ and $S \in \mathbf{CUPER}$, then $(\sqcup_{i \in I} R_i) \rightarrow S = \cap_{i \in I} (R_i \rightarrow S)$.

Proof

Since $(\sqcup_{i \in I} R_i) \rightarrow S \subseteq R_i \rightarrow S$ for each $i \in I$, $(\sqcup_{i \in I} R_i) \rightarrow S \subseteq \cap_{i \in I} (R_i \rightarrow S)$. In order to prove the converse, consider two functions f, g such that $(f, g) \in \cap_{i \in I} (R_i \rightarrow S)$. The assumption that $(f, g) \in \cap_{i \in I} (R_i \rightarrow S)$ implies that if $(x, y) \in \sqcup_{i \in I} R_i$ then $(f(x), f(y)) \in S$ and $(g(x), g(y)) \in S$. Since $\sqcup_{i \in I} R_i \in \mathbf{NUSR}$, by applying Proposition 14.2-10 we obtain that $(x, y) \in C(\sqcup_{i \in I} R_i)$ implies $(f(x), g(y)) \in C(S)$; that is, $(x, y) \in \sqcup_{i \in I} R_i$ implies $(f(x), g(y)) \in S$. Therefore $(f, g) \in (\sqcup_{i \in I} R_i) \rightarrow S$.

□

As a special case of this proposition, we obtain:

Proposition 14.2-12

For all n , if $R_i \in \mathbf{CUPER}$ for all $i \in I$, $S_j \in \mathbf{CUPER}$ for $j \in J$, and for all $i \in I$ there exists $j \in J$ such that $p_n(R_i) \subseteq p_n(S_j)$, then $p_n(\sqcup_{i \in I} R_i) \subseteq p_n(\sqcup_{j \in J} S_j)$.

Proof

If for all $i \in I$ there exists $j \in J$ such that $p_n(R_i) \subseteq p_n(S_j)$, then $(p_n, p_n) \in \cap_{i \in I} (R_i \rightarrow (\sqcup_{j \in J} S_j))$. By Proposition 14.2-11, $(p_n, p_n) \in (\sqcup_{i \in I} R_i) \rightarrow (\sqcup_{j \in J} S_j)$, hence $p_n(\sqcup_{i \in I} R_i) \subseteq (\sqcup_{j \in J} S_j)$, and immediately $p_n(\sqcup_{i \in I} R_i) \subseteq p_n(\sqcup_{j \in J} S_j)$.

□

14.2.5 Metric Properties

As mentioned above, Amadio has studied the metric properties of \rightarrow and μ . In summary, \rightarrow is contractive; if $H(R_1, \dots, R_{k+1})$ is nonexpansive in R_1, \dots, R_k and contractive in R_{k+1} , then $\mu(R_{k+1})H(R_1, \dots, R_{k+1})$ is nonexpansive in R_1, \dots, R_k ; and if $H(R_1, \dots, R_{k+1})$ is contractive in R_1, \dots, R_{k+1} , then $\mu(R_{k+1})H(R_1, \dots, R_{k+1})$ is contractive in R_1, \dots, R_k .

Now we can prove that the object-type construction is contractive, and that bounded join and intersection are nonexpansive. As a preliminary step, we deal with the record-type construction.

Proposition 14.2-13

If $G_i(R_1, \dots, R_k)$ is nonexpansive in R_1, \dots, R_k for all $i \in I$, then $\langle \langle m_i; G_i(R_1, \dots, R_k) \rangle \rangle^{i \in I}$ is contractive in R_1, \dots, R_k .

Proof

We give the proof for $k = 1$; the general case is similar to this one. Let R and R' be equal up to rank n (at least). Since G_i is nonexpansive for all $i \in I$, we obtain that $G_i(R)$ and $G_i(R')$ are equal up to rank n . We show that $\langle\langle m_i; G_i(R)^{i \in I} \rangle\rangle$ and $\langle\langle m_i; G_i(R')^{i \in I} \rangle\rangle$ are equal up to rank $n+1$. Reasoning by contradiction, suppose that (o, o') has rank $n+1$ and that $(o, o') \in \langle\langle m_i; G_i(R)^{i \in I} \rangle\rangle$ but $(o, o') \notin \langle\langle m_i; G_i(R')^{i \in I} \rangle\rangle$. Since $(o, o') \in \langle\langle m_i; G_i(R)^{i \in I} \rangle\rangle$, either $(o, o') = (\perp, \perp)$ or $(o, o') \in (L \rightarrow D) \times (L \rightarrow D)$. Since $(o, o') \notin \langle\langle m_i; G_i(R')^{i \in I} \rangle\rangle$, it must be that $(o, o') \in (L \rightarrow D) \times (L \rightarrow D)$, and $(o(m_i), o'(m_i)) \notin G_i(R')$ for some $i \in I$; but $(o(m_i), o'(m_i)) \in G_i(R)$ because $(o, o') \in \langle\langle m_i; G_i(R)^{i \in I} \rangle\rangle$. Because (o, o') has rank $n+1$, $(o(m_i), o'(m_i))$ is either $(p_n(o(m_i)), p_n(o'(m_i)))$ or (\perp, \perp) , depending on whether $n \leq i$. In either case, this means that $(o(m_i), o'(m_i))$ has rank at most n . In short, we have found an index $i \in I$ and a pair $(o(m_i), o'(m_i))$ of rank at most n such that $(o(m_i), o'(m_i)) \in G_i(R)$ and $(o(m_i), o'(m_i)) \notin G_i(R')$; this contradicts the fact that $G_i(R)$ and $G_i(R')$ are equal up to rank n .

□

Proposition 14.2-14

If $G_i(R_1, \dots, R_k)$ is nonexpansive in R_1, \dots, R_k for all $i \in I$, then $\langle\langle [m_i; G_i(R_1, \dots, R_k)]^{i \in I} \rangle\rangle$ is contractive in R_1, \dots, R_k .

Proof

We give the proof for $k = 1$. Let R and R' be equal up to rank n . We show that for every extension F of $\lambda(S)\langle\langle m_i; S \rightarrow G_i(R)^{i \in I} \rangle\rangle$ there exists an extension F' of $\lambda(S)\langle\langle m_i; S \rightarrow G_i(R')^{i \in I} \rangle\rangle$ equal to F up to rank $n+1$. Consider $F \in \mathbf{Gen}$ such that $F \preccurlyeq \lambda(S)\langle\langle m_i; S \rightarrow G_i(R)^{i \in I} \rangle\rangle$. If $F = \lambda(S)\langle\langle m_i; S \rightarrow G_i(R)^{i \in I}, m_j; S \rightarrow T_j^{j \in J} \rangle\rangle$, let $F' = \lambda(S)\langle\langle m_i; S \rightarrow G_i(R')^{i \in I}, m_j; S \rightarrow T_j^{j \in J} \rangle\rangle$. Now, $F' \in \mathbf{Gen}$ and $F' \preccurlyeq \lambda(S)\langle\langle m_i; S \rightarrow G_i(R')^{i \in I} \rangle\rangle$. Furthermore, we obtain that F and F' are equal up to rank $n+1$, because each G_i is nonexpansive, \rightarrow is contractive, and $\langle\langle \dots \rangle\rangle$ is contractive by Proposition 14.2-13. Then $\mu(S)F(S)$ and $\mu(S)F'(S)$ coincide up to rank $n+1$, and hence $\langle\langle [m_i; G_i(R)]^{i \in I} \rangle\rangle$ is included in $\langle\langle [m_i; G_i(R')]^{i \in I} \rangle\rangle$ up to rank $n+1$ by Proposition 14.2-12. We obtain the converse analogously. Hence $\langle\langle [m_i; G_i(R)]^{i \in I} \rangle\rangle$ and $\langle\langle [m_i; G_i(R')]^{i \in I} \rangle\rangle$ coincide up to rank $n+1$.

□

Proposition 14.2-15

If $G(R_1, \dots, R_{k+1})$ is contractive (nonexpansive) in R_1, \dots, R_{k+1} and $H(R_1, \dots, R_k)$ is nonexpansive in R_1, \dots, R_k , then $\sqcup_{R_{k+1} \in \mathbf{CUPER}, R_{k+1} \subseteq H(R_1, \dots, R_k)} G(R_1, \dots, R_{k+1})$ is contractive (nonexpansive) in R_1, \dots, R_k .

Proof

Again we give the proof for $k = 1$, and we prove only the contractiveness claim as the other one is similar. Suppose that R and R' are equal up to rank n . We want to prove that $\sqcup_{R_2 \in \mathbf{CUPER}, R_2 \subseteq H(R)} G(R, R_2)$ and $\sqcup_{R_2 \in \mathbf{CUPER}, R_2 \subseteq H(R')} G(R', R_2)$ are equal up to

rank $n+1$. Let $(x, x') \in \sqcup_{R_2 \in \text{CUPER}, R_2 \subseteq H(R)} G(R, R_2)$ be a pair of rank $n+1$. We wish to show that $(x, x') \in \sqcup_{R_2 \in \text{CUPER}, R_2 \subseteq H(R')} G(R', R_2)$. By Proposition 14.2-12 it suffices to prove that if $(x, x') \in G(R, R_2)$ for some $R_2 \subseteq H(R)$ then $(x, x') \in \sqcup_{R_2 \in \text{CUPER}, R_2 \subseteq H(R')} G(R', R_2)$, by proving that $(x, x') \in G(R', R_2')$ for some $R_2' \in \text{CUPER}$ such that $R_2' \subseteq H(R')$. So assume that $(x, x') \in G(R, R_2)$ and $R_2 \subseteq H(R)$. Since G is contractive in R_2 and (x, x') is of rank $n+1$, we have $(x, x') \in G(R, p_n(R_2))$. Similarly, since G is contractive in R , and R and R' are equal up to rank n , we have $(x, x') \in G(R', p_n(R_2))$. In addition, $p_n(R_2) \in \text{CUPER}$ and $p_n(R_2) \subseteq H(R)$. Since H is nonexpansive in R , and R and R' are equal up to rank n , we have also $p_n(R_2) \subseteq H(R')$. It follows that $(x, x') \in \sqcup_{R_2 \in \text{CUPER}, R_2 \subseteq H(R')} G(R', R_2)$.

□

Proposition 14.2-16

If $G(R_1, \dots, R_{k+1})$ is contractive (nonexpansive) in R_1, \dots, R_{k+1} and $H(R_1, \dots, R_k)$ is nonexpansive in R_1, \dots, R_k , then $\sqcap_{R_{k+1} \in \text{CUPER}, R_{k+1} \subseteq H(R_1, \dots, R_k)} G(R_1, \dots, R_{k+1})$ is contractive (nonexpansive) in R_1, \dots, R_k .

Proof

We give the contractiveness proof for $k = 1$. Suppose that R and R' are equal up to rank n . We want to prove that $\sqcap_{R_2 \in \text{CUPER}, R_2 \subseteq H(R)} G(R, R_2)$ and $\sqcap_{R_2 \in \text{CUPER}, R_2 \subseteq H(R')} G(R', R_2)$ are equal up to rank $n+1$. Let $(x, x') \in \sqcap_{R_2 \in \text{CUPER}, R_2 \subseteq H(R)} G(R, R_2)$ be a pair of rank $n+1$. Pick R_2 such that $R_2 \in \text{CUPER}$ and $R_2 \subseteq H(R')$, to show that $(x, x') \in G(R', R_2)$. It will follow that $(x, x') \in \sqcap_{R_2 \in \text{CUPER}, R_2 \subseteq H(R')} G(R', R_2)$. Since R and R' agree up to rank n and H is nonexpansive, $R_2 \subseteq H(R')$ implies $p_n(R_2) \subseteq H(R)$. In addition, $p_n(R_2) \in \text{CUPER}$. Hence $(x, x') \in G(R, p_n(R_2))$. Since G is contractive in R , it follows that $(x, x') \in G(R', p_n(R_2))$. Since G is contractive in R_2 , it follows that $(x, x') \in G(R', R_2)$.

□

14.3 The Interpretation of Types and Typed Terms

Finally in this section we give an interpretation of types and terms, and check the soundness of $\text{FOb}_{<\mu}$ under this interpretation. We consider not only the typing rules of $\text{FOb}_{<\mu}$ but also its equational rules (given by the fragments $\Delta_=\Delta_x, \Delta_{= \rightarrow}, \Delta_{= \text{Ob}}, \Delta_{= \text{c}}, \Delta_{= \text{c}; \text{Ob}}, \Delta_{= \text{c}; \mu}, \Delta_{= \text{c}; \forall}$, and $\Delta_{= \text{c}; \exists}$). The proofs are long but mostly straightforward because of the previous semantic development.

14.3.1 Interpreting Types

We define the semantics function for types:

$$\llbracket \cdot \rrbracket : (TV \rightarrow \text{CUPER}) \rightarrow (TE \rightarrow \text{CUPER})$$

where TV is the set of type variables and TE the set of type expressions. A mapping η in $TV \rightarrow \text{CUPER}$ is a *type environment*. We write $\llbracket A \rrbracket_\eta$ for the semantics of a type A with the environment η . We set:

$$\begin{aligned}\llbracket X \rrbracket_\eta &\triangleq \eta(X) \\ \llbracket \text{Top} \rrbracket_\eta &\triangleq \text{Univ} \\ \llbracket [m_i; C_i]_{i \in 1..n} \rrbracket_\eta &\triangleq ([m_i; \llbracket C_i \rrbracket_\eta]_{i \in 1..n}) \\ \llbracket A \rightarrow B \rrbracket_\eta &\triangleq \llbracket A \rrbracket_\eta \rightarrow \llbracket B \rrbracket_\eta \\ \llbracket \forall(X <: B)A \rrbracket_\eta &\triangleq \sqcap_{R \in \text{CUPER}, R \subseteq B} \llbracket A \rrbracket_{\eta(X \leftarrow R)} \\ \llbracket \exists(X <: B)A \rrbracket_\eta &\triangleq \sqcup_{R \in \text{CUPER}, R \subseteq B} \llbracket A \rrbracket_{\eta(X \leftarrow R)} \\ \llbracket \mu(X)A \rrbracket_\eta &\triangleq \mu(T)(\text{Univ} \rightarrow \llbracket A \rrbracket_{\eta(X \leftarrow T)})\end{aligned}$$

The definition for recursive types is the most unusual one. A simpler alternative would have been: $\llbracket \mu(X)A \rrbracket_\eta \triangleq \mu(T)\llbracket A \rrbracket_{\eta(X \leftarrow T)}$. The simpler definition would have yielded the attractive property $\llbracket \mu(X)A\{X\} \rrbracket_\eta = \llbracket A\{\mu(X)A\} \rrbracket_\eta$, but we do not need that property. Moreover, the simpler definition requires special treatment for some types, such as $\mu(X)X$; technically, it works well only for types that are formally contractive [17, 82]. The definition that we have given does not depend on formal contractiveness; it yields $\llbracket \mu(X)A\{X\} \rrbracket_\eta = \llbracket \mu(X)A\{\mu(X)A\} \rrbracket_\eta$ but not $\llbracket \mu(X)A\{X\} \rrbracket_\eta = \llbracket A\{\mu(X)A\} \rrbracket_\eta$.

The following proposition implies that the definition is proper in all cases:

Proposition 14.3-1

If A is a well-formed type expression, then $\llbracket A \rrbracket_{\eta(X \leftarrow R)}$ is nonexpansive in R , and hence $\text{Univ} \rightarrow \llbracket A \rrbracket_{\eta(X \leftarrow R)}$ is contractive in R .

Proof

The proof is by induction on the structure of A . The cases of type variables and \rightarrow are well known. It is also well known that metric fixpoints preserve nonexpansiveness, and hence if $\llbracket A \rrbracket_{\eta(Y \leftarrow T)}$ is nonexpansive in T then so are both $\text{Univ} \rightarrow \llbracket A \rrbracket_{\eta(Y \leftarrow T)}$ and $\mu(T)(\text{Univ} \rightarrow \llbracket A \rrbracket_{\eta(Y \leftarrow T)})$. In fact, $\text{Univ} \rightarrow \llbracket A \rrbracket_{\eta(Y \leftarrow T)}$ is contractive in T , so when $X \equiv Y$ we have that $\mu(T)(\text{Univ} \rightarrow \llbracket A \rrbracket_{\eta(X \leftarrow T)})$ is well defined. The case of Top is trivial. Propositions 14.2-15 and 14.2-16 deal with bounded quantification. Proposition 14.2-14 deals with object types.

□

It follows that every well-formed type expression denotes a nonexpansive function on CUPER (as a function of its free variables).

14.3.2 Interpreting Typed Terms

We define the semantics function for terms:

$$\llbracket \cdot \rrbracket : (V \rightarrow D) \rightarrow (E \rightarrow D)$$

where V is the set of variables and E the set of expressions. A mapping ρ in $V \rightarrow D$ is an *environment*. We write $\llbracket a \rrbracket_\rho$ for the semantics of a term a with an environment ρ . The

semantics of terms does not depend on the semantics of types; all type information is erased in the course of the interpretation of terms. Therefore we do not need a type environment.

$\llbracket x \rrbracket_p$	$\triangleq p(x)$
$\llbracket [m_i = \zeta(x:A)c_i]^{i \in I} \rrbracket_p$	$\triangleq \langle \langle m_i = \llbracket \lambda(x_i:A)c_i \rrbracket_p \rangle^{i \in I} \rangle$
$\llbracket a.m \rrbracket_p$	$\triangleq \text{if } \llbracket a \rrbracket_p \in (L \rightarrow D) \text{ and } \llbracket a \rrbracket_p(m) \in (D \rightarrow D)$ $\text{then } \llbracket a \rrbracket_p(m)(\llbracket a \rrbracket_p) \text{ else } *$
$\llbracket a.m = \zeta(x:A)c \rrbracket_p$	$\triangleq \text{if } \llbracket a \rrbracket_p \in (L \rightarrow D)$ $\text{then } \llbracket a \rrbracket_p(m \leftarrow \llbracket \lambda(x:A)c \rrbracket_p) \text{ else } *$
$\llbracket \lambda(x:A)b \rrbracket_p$	$\triangleq \lambda(v)\llbracket b \rrbracket_{p(x \leftarrow v)}$
$\llbracket b(a) \rrbracket_p$	$\triangleq \text{if } \llbracket b \rrbracket_p \in (D \rightarrow D) \text{ then } \llbracket b \rrbracket_p(\llbracket a \rrbracket_p) \text{ else } *$
$\llbracket \lambda(X <: A)b \rrbracket_p$	$\triangleq \llbracket b \rrbracket_p$
$\llbracket b(A) \rrbracket_p$	$\triangleq \llbracket b \rrbracket_p$
$\llbracket \text{pack } X <: A = C \text{ with } b(X):B(X) \rrbracket_p$	$\triangleq \llbracket b \rrbracket_p$
$\llbracket \text{open } c \text{ as } X <: A, x:B \text{ in } d(x):D \rrbracket_p$	$\triangleq \llbracket d \rrbracket_{p(x \leftarrow \llbracket c \rrbracket_p)}$
$\llbracket \text{fold}(A,a) \rrbracket_p$	$\triangleq \lambda(v)\llbracket a \rrbracket_p$
$\llbracket \text{unfold}(a) \rrbracket_p$	$\triangleq \llbracket a \rrbracket_p(\perp)$

This definition is given in a metalanguage where $v \in V$ is a strict membership test, and where conditionals and conjunctions are strict and evaluated left to right, for example $\llbracket a.m = \zeta(x:A)c \rrbracket_p = \perp$ if $\llbracket a \rrbracket_p = \perp$. Thus the interpretation of each term is a continuous function of the interpretation of its free variables.

Note how the semantics turns ζ 's into λ 's and objects into records. In a sense, then, it is easy to encode an object calculus in a record calculus (essentially using the ideas of the self-application semantics). The catch is that the denotations of object types are not standard record types.

Note also that, in the semantics, updating a method of an object does not produce an error if the object does not have the method in the first place. This definition of update simplifies our technical treatment.

14.3.3 Soundness

We say that E and η are consistent in the usual sense: if $X <: A$ appears in E , then $\eta(X) \subseteq \llbracket A \rrbracket_\eta$. We say that E , η , and (ρ, ρ') are consistent when, in addition, if $x:A$ appears in E , then $(\rho(x), \rho'(x)) \in \llbracket A \rrbracket_\eta$. Next we check that, if η and (ρ, ρ') are consistent with E , then $E \vdash B <: A$ implies $\llbracket B \rrbracket_\eta \subseteq \llbracket A \rrbracket_\eta$, $E \vdash a : A$ implies $(\llbracket a \rrbracket_\rho, \llbracket a \rrbracket_{\rho'}) \in \llbracket A \rrbracket_\eta$, and $E \vdash a \leftrightarrow a' : A$ implies $(\llbracket a \rrbracket_\rho, \llbracket a' \rrbracket_{\rho'}) \in \llbracket A \rrbracket_\eta$. An immediate corollary of these verifications is that no well-typed term has $*$ as its denotation.

The proof of soundness is by induction on derivations. We start with the subtyping rules.

Lemma 14.3-2 (Soundness of subtyping rules)

The subtyping rules are sound.

Proof

We treat only two interesting rules. The arguments for other rules are standard.

$$\begin{array}{c} (\text{Sub Object}) \quad (l_i \text{ distinct}) \\ E \vdash B_i \quad \forall i \in 1..n+m \\ \hline E \vdash [l_i; B_i]_{i \in 1..n+m} <: [l_i; B_i]_{i \in 1..n} \end{array}$$

This follows immediately from the definition of the semantics and from Proposition 14.2-3.

$$\begin{array}{c} (\text{Sub Rec}) \\ E \vdash \mu(X)A \quad E \vdash \mu(Y)B \quad E, Y <: \text{Top}, X <: Y \vdash A <: B \\ \hline E \vdash \mu(X)A <: \mu(Y)B \end{array}$$

Consider η and (ρ, ρ') consistent with E , and $R, T \in \mathbf{CUPER}$ such that $R \subseteq T$. Let $\vartheta = \eta(Y \leftarrow T)(X \leftarrow R)$. Since ϑ and (ρ, ρ') are consistent with E , $Y <: \text{Top}$, $X <: Y$ the third hypothesis implies that $\llbracket A \rrbracket_\vartheta \subseteq \llbracket B \rrbracket_\vartheta$. According to the other hypotheses, $X \notin FV(B)$ and $Y \notin FV(A)$; so $\llbracket A \rrbracket_{\eta(X \leftarrow R)} \subseteq \llbracket B \rrbracket_{\eta(Y \leftarrow T)}$. Hence $\text{Univ} \rightarrow \llbracket A \rrbracket_{\eta(X \leftarrow R)} \subseteq \text{Univ} \rightarrow \llbracket B \rrbracket_{\eta(Y \leftarrow T)}$. Let F be the function that maps a cuper R to $\text{Univ} \rightarrow \llbracket A \rrbracket_{\eta(X \leftarrow R)}$ and G be the function that maps a cuper T to $\text{Univ} \rightarrow \llbracket B \rrbracket_{\eta(Y \leftarrow T)}$. We have just obtained that $R \subseteq T$ implies $F(R) \subseteq G(T)$. Since these functions are contractive, Proposition 14.2-2 yields that $\mu(S)F(S) \subseteq \mu(S)G(S)$, that is, that $\llbracket \mu(X)A \rrbracket_\eta \subseteq \llbracket \mu(Y)B \rrbracket_\eta$.

□

The proof of the soundness of the equational rules is given in Appendix C.4, as Lemma C.4-7. The arguments for the value typing rules are special cases of those for the equational rules.

We obtain the following soundness results:

Theorem 14.3-3

Assume that η and (ρ, ρ') are consistent with E . Then, for derivations in $\mathbf{FOb}_{<: \mu}$:

If $E \vdash A$, then $\llbracket A \rrbracket_\eta \in \mathbf{CUPER}$.

If $E \vdash A <: B$, then $\llbracket A \rrbracket_\eta \subseteq \llbracket B \rrbracket_\eta$.

If $E \vdash a : A$, then $(\llbracket a \rrbracket_\rho, \llbracket a \rrbracket_{\rho'}) \in \llbracket A \rrbracket_\eta$.

If $E \vdash a \leftrightarrow a' : A$, then $(\llbracket a \rrbracket_\rho, \llbracket a' \rrbracket_{\rho'})$ in $\llbracket A \rrbracket_\eta$.

□

By reversing the last two implications, it follows that:

- If $\llbracket a \rrbracket_\rho = *$, then $E \not\vdash a : A$, so the type theory is sound.
- If $\llbracket a \rrbracket_\rho \neq \llbracket a' \rrbracket_{\rho'}$, then $E \not\vdash a \leftrightarrow a' : A$, so the equational theory is consistent.

15 DEFINABLE COVARIANT SELF TYPES

In this chapter, we resume the study of Self types. We work entirely within the calculus $\mathbf{Ob}_{\leq\mu}$ of Chapter 13 (namely, the second-order calculus with bounded quantifiers, recursion, and simple object types). Within this calculus, we derive rules for the combination of object types and the Self quantifier.

We show how, using those derived rules, we can easily provide typings for the examples of Section 6.5. In particular, we can give satisfactory types to objects with methods that return self. However, we find that updating methods that return self is difficult, and so is the definition of sufficiently parametric pre-methods for use in classes. We explore two solutions to these problems. The first solution simply relies on adding a trivial auxiliary method to objects. The second solution consists in assuming stronger invariants about objects, and in building those invariants into our rules. The second solution leads us to a reformulation of the rules for Self types that we develop in Chapter 16.

Self types, as we define them in this chapter, are not intended for use in the types of binary methods. We discuss binary methods briefly, and postpone a fuller discussion to Part III.

15.1 ζ -Objects

The payoff of the Self quantifier, and particularly of the (Sub Self) rule derived in Section 13.5, comes when it is used in conjunction with object types. *Object types with Self* are obtained by the combination of the simple object types of Chapters 7 and 8 with the Self quantifier of Section 13.5. These new types allow subsumption between objects containing methods that return self.

15.1.1 Defining ζ -Objects

We examine types of the form:

$$\zeta(X)[l; B_i\{X^+\}^{i \in 1..n}] \quad \text{where } B_i\{X^+\} \text{ indicates that each } B_i\{X\} \text{ is covariant in } X$$

We call these structures ζ -object types, and ζ -objects their corresponding values. The parameter X in $\zeta(X)[l; B_i\{X^+\}^{i \in 1..n}]$ is intended as the type Self hypothesized in Section 9.6. We routinely call the parameter of a Self quantifier a Self type, or simply Self. The B_i are result types of methods; they may depend on Self. If Self occurs in a B_i , it must occur covariantly there.

Note that we do not require that $[l_i:B_i\{X\}^{i \in 1..n}]$ be covariant in X , only that each $B_i\{X\}$ be covariant in X . (In fact, $[l_i:B_i\{X\}^{i \in 1..n}]$ is invariant in X as soon as X occurs in some $B_i\{X\}$.) Although the Self quantifier of Chapter 13 has no covariance restrictions, we shall see that the covariance requirement is necessary when selecting components of ζ -objects.

The covariance requirement implies that X_i must not occur within any object type within B_i , since object types are invariant in their components. For example, the type $\zeta(X)[l:\zeta(Y)[m:X, n:Y]]$ violates the covariance requirement. Hence, informally, we may say that Self types do not nest: there is a single meaningful Self type within each pair of object brackets.

It is often useful to consider an unfolding $A(C) \equiv [l_i:B_i\{C\}^{i \in 1..n}]$ of a ζ -object type $A \equiv \zeta(X)[l_i:B_i\{X^+\}^{i \in 1..n}]$, for $C <: A$. We frequently consider $A(X)$, for a variable X , and the *self-unfolding* $A(A)$ of A . (When building an element of type A it is common to build first an element of type $A(A)$.) Together, a type $C <: A$ and an element of $A(C)$ can be used for building an element of A ; we say that C is the *representation type* for this element.

15.1.2 Derived Rules for ζ -Objects

Building on the rules for the Self quantifier (Chapter 13), we consider special rules for ζ -objects. We rely on the following notation. We use the syntax $\text{wrap}(Y <: A = C)b[Y]$ from Chapter 13 for ζ -object values, where b now has an object type. Operations on ζ -objects are indicated by special syntax that encapsulates occurrences of the *use* construct:

ζ -Object operations

Let $A \equiv \zeta(X)[l_i:B_i\{X^+\}^{i \in 1..n}]$, $A(C) \equiv [l_i:B_i\{C\}^{i \in 1..n}]$, and $j \in 1..n$.

$a_{A,l_j} \triangleq$

use a as $Z <: A$, $y:A(Z)$ in $y.l_j : B_j\{A\}$

$a.l_j \triangleq (Y <: A, y:A(Y))\zeta(x:A(Y))b[Y, y, x] \triangleq$

use a as $Z <: A$, $y:A(Z)$ in $\text{wrap}(Y <: A = Z)(y.l_j \triangleq \zeta(x:A(Y))b[Y, y, x]) : A$

Essentially, a_{A,l_j} is a form of selection and $a.l_j \triangleq (Y <: A, y:A(Y))\zeta(x:A(Y))b$ is a form of update. In the selection operation, we include a subscript A in order to make the definition unambiguous; this subscript is a type for the term a , and is often omitted. In the update operation, the updating method b can take advantage of three variables: (1) $Y <: A$, the unknown subtype of A that was used to construct a ; (2) $y:A(Y)$, the raw object inside a , which can be thought of as the old self; y is the value of self at the time the update takes place, containing the old version of method l_j ; (3) $x:A(Y)$, the regular self of the updating method b .

Combining the rules for objects and for Self quantification we derive the following rules. (Some of the derivations are in Appendix C.3.)

$\Delta_{\zeta+}$

(Type ζ Object) (where $A \equiv \zeta(X)[l_i; B_i\{X^+\}_{i \in 1..n}]$) $E, X \vdash B_i\{X^+\} \quad \forall i \in 1..n$ $E \vdash \zeta(X)[l_i; B_i\{X\}_{i \in 1..n}]$	(Sub ζ Object) (where $A' \equiv \zeta(X)[l_i; B_i\{X^+\}_{i \in 1..n+m}]$) $E, X \vdash B_i\{X^+\} \quad \forall i \in 1..n+m$ $E \vdash \zeta(X)[l_i; B_i\{X\}_{i \in 1..n+m}] <: \zeta(X)[l_i; B_i\{X\}_{i \in 1..n}]$
(Val ζ Object) (where $A \equiv \zeta(X)[l_i; B_i\{X^+\}_{i \in 1..n}]$) $E \vdash C <: A \quad E \vdash b\{C\} : A(C)$	
$E \vdash \text{wrap}(Y <: A = C) b\{Y\} : A$	
(Val ζ Select) (where $A \equiv \zeta(X)[l_i; B_i\{X^+\}_{i \in 1..n}]$) $E \vdash a : A \quad j \in 1..n$	
$E \vdash a_A l_j : B_j\{A\}$	
(Val ζ Update) (where $A \equiv \zeta(X)[l_i; B_i\{X^+\}_{i \in 1..n}]$) $E \vdash a : A \quad E, Y <: A, y : A(Y), x : A(Y) \vdash b : B_j\{Y\} \quad j \in 1..n$	
$E \vdash a.l_j = (Y <: A, y : A(Y)) \zeta(x : A(Y)) b : A$	

 $\Delta_{\zeta+}$

(Eq ζ Object) (where $A \equiv \zeta(X)[l_i; B_i\{X^+\}_{i \in 1..n}]$, $A' \equiv \zeta(X)[l_i; B_i\{X^+\}_{i \in 1..n+m}]$) $E \vdash C <: A' \quad E \vdash b\{C\} \leftrightarrow b'\{C\} : A'(C)$ $E \vdash \text{wrap}(Y <: A = C) b\{Y\} \leftrightarrow \text{wrap}(Y <: A' = C) b'\{Y\} : A$
(Eq Sub ζ Object) (where $A \equiv \zeta(X)[l_i; B_i\{X^+\}_{i \in 1..n}]$, $A' \equiv \zeta(X)[l_i; B_i\{X^+\}_{i \in 1..n+m}]$) $E \vdash C <: A' \quad E, x_i : A(C) \vdash b_i\{C\} : B_i\{C\} \quad \forall i \in 1..n$ $E, x_j : A'(C) \vdash b_j\{C\} : B_j\{C\} \quad \forall j \in n+1..n+m$ $E \vdash \text{wrap}(Y <: A = C)[l_i = \zeta(x_i : A(Y)) b_i\{Y\}_{i \in 1..n}] \leftrightarrow$ $\text{wrap}(Y <: A' = C)[l_i = \zeta(x_i : A'(Y)) b_i\{Y\}_{i \in 1..n+m}] : A$
(Eq ζ Select) (where $A \equiv \zeta(X)[l_i; B_i\{X^+\}_{i \in 1..n}]$) $E \vdash a \leftrightarrow a' : A \quad j \in 1..n$ $E \vdash a_A l_j \leftrightarrow a'_{A \circ l_j} : B_j\{A\}$
(Eq ζ Update) (where $A \equiv \zeta(X)[l_i; B_i\{X^+\}_{i \in 1..n}]$) $E \vdash a \leftrightarrow a' : A \quad E, Y <: A, y : A(Y), x : A(Y) \vdash b \leftrightarrow b' : B_j\{Y\} \quad j \in 1..n$ $E \vdash a.l_j = (Y <: A, y : A(Y)) \zeta(x : A(Y)) b \leftrightarrow a'.l_j = (Y <: A, y : A(Y)) \zeta(x : A(Y)) b' : A$

(Eval ζ Select) (where $A \equiv \zeta(X)[l_i:B_i\{X^+\}_{i \in 1..n}]$, $c \equiv \text{wrap}(Z <: A = C)a\{Z\}$)

$$E \vdash c : A \quad j \in 1..n$$

$$\frac{}{E \vdash c_{A;l_j} \leftrightarrow a\{C\}.l_j : B_j\{A\}}$$

(Eval ζ Update) (where $A \equiv \zeta(X)[l_i:B_i\{X^+\}_{i \in 1..n}]$, $c \equiv \text{wrap}(Z <: A = C)a\{Z\}$)

$$E \vdash c : A \quad E, Y <: A, y:A(Y), x:A(Y) \vdash b\{Y,y,x\} : B_j\{Y\} \quad j \in 1..n$$

$$E \vdash c_{A;l_j}(Y <: A, y:A(Y))\zeta(x:A(Y))b\{Y,y,x\} \leftrightarrow$$

$$\text{wrap}(Z <: A = C)a\{Z\}.l_j \zeta(x:A(Z))b\{Z,a\{Z\},x\} : A$$

In the first two rules, (Type ζ Object) and (Sub ζ Object), the antecedents require that each $B_i\{X\}$ be covariant in X . Whenever a type of the form $\zeta(X)[l_i:B_i\{X^+\}_{i \in 1..n}]$ appears in the antecedents of the other rules, the covariance condition is thus ensured.

The most remarkable fact about ζ -object types is that the rule (Sub ζ Object) holds. In contrast, an example in Section 9.5 shows that a rule analogous to (Sub ζ Object) but where μ replaces ζ is unsound.

The rule (Val ζ Object) can be used for building a ζ -object $\text{wrap}(Y <: A = C)b\{Y\}$ from a subtype C of the desired ζ -object type A , and from a regular object $b\{C\}$. The Y variable in $b\{Y\}$ is the Self type, in case the methods of b need to refer to it. When constructing a fixed ζ -object, its methods are not required to operate on an arbitrary self: they just need to match the given representation type of the object being constructed. That is, to construct a ζ -object of type $A \equiv \zeta(X)[l_i:B_i\{X^+\}_{i \in 1..n}]$ it suffices to have a set of methods $b_j : B_j\{C\}$ with $C <: A$ (not $b_j : B_j\{X\}$ for an arbitrary $X <: A$). Moreover, each of these methods can assume the existence of a self parameter $x_j : [l_i:B_i\{C\}_{i \in 1..n}]$.

The ζ -object selection operation $a_{A;l_j}$ reduces fairly simply to a regular selection operation on the underlying object.

The ζ -object update operation $a_{A;l_j}(Y <: A, y:A(Y))\zeta(x:A(Y))b$ is more interesting, although similarly it reduces to an update operation on the underlying object. The method body b must produce a result of type $B_j\{Y\}$, parametrically in Y .

The rule (Eq ζ Object) yields a flexible congruence lemma, analogous to the one obtained from the (Eq Pack $<:$) rule for existentials:

(Eq ζ Object Lemma) (where $A \equiv \zeta(X)[l_i:B_i\{X^+\}_{i \in 1..n}]$, $A' \equiv \zeta(X)[l_i:B_i\{X^+\}_{i \in 1..n+m}]$)

$$E \vdash C <: A' \quad E \vdash b\{C\} \leftrightarrow b'\{C\} : A(C) \quad E \vdash b'\{C\} : A'(C)$$

$$\frac{}{E \vdash \text{wrap}(Y <: A = C)b\{Y\} \leftrightarrow \text{wrap}(Y <: A' = C)b'\{Y\} : A}$$

The rule (Eq Sub ζ Object) is obtained from (Eq Sub Object) and (Eq Wrap Lemma). This rule is of limited power because the same type C appears on both sides of the equation in the conclusion. We can trace back this limitation to a similar limitation in the rules for existential types.

15.2 Examples, with Typing

We are now ready to reexamine some examples. We find that these examples can be typed rather easily when seen in terms of ζ -objects, even when a method needs to return or to modify *self*.

15.2.1 Movable Points

This is the typed version of the example in Section 6.5.1. We begin by defining the types of one-dimensional and two-dimensional movable points using the *Self* quantifier:

$$\begin{aligned} P_1 &\triangleq \zeta(\text{Self})[x:\text{Int}, mv_x:\text{Int}\rightarrow\text{Self}] \\ P_2 &\triangleq \zeta(\text{Self})[x,y:\text{Int}, mv_x,mv_y:\text{Int}\rightarrow\text{Self}] \end{aligned}$$

We have the inclusion $P_2 <: P_1$, by (Sub ζ Object). (Compare with Section 9.5.)

Next we define the one-dimensional origin point, where *Self* names the type of the object, P_1 . Recall that $\text{wrap}(A,c)$ abbreviates $\text{wrap}(X <: A = A)c$ for an unused X , and that $\text{wrap}(X = A)c$ abbreviates $\text{wrap}(X <: A = A)c$. A common pattern in the code below is that methods for a ζ -object type A use the construct $\text{wrap}(X,c)$ to produce a value of type X from a value c of type $A(X)$.

$$\begin{aligned} \text{origin}_1 : P_1 &\triangleq \\ &\text{wrap}(\text{Self} = P_1)[x = 0, mv_x = \zeta(s:P_1(\text{Self})) \lambda(dx:\text{Int}) \text{wrap}(\text{Self}, s.x := s.x + dx)] \end{aligned}$$

The typing $\text{origin}_1 : P_1$ can be derived as follows, with $P_1(P_1) \equiv [x:\text{Int}, mv_x:\text{Int}\rightarrow P_1]$ as the chosen representation type for P_1 .

$$\left\{ \begin{array}{ll} \left\{ \begin{array}{l} \left\{ \begin{array}{l} \emptyset, s:P_1(P_1), dx:\text{Int} \vdash s.x := s.x + dx : P_1(P_1) \\ \emptyset, s:P_1(P_1), dx:\text{Int} \vdash P_1 <: P_1 \end{array} \right. & \text{by (Val Select), (Val Update)} \\ \downarrow \emptyset, s:P_1(P_1), dx:\text{Int} \vdash \text{wrap}(P_1, s.x := s.x + dx) : P_1 & \text{by (Sub Refl)} \\ \downarrow \emptyset, s:P_1(P_1) \vdash \lambda(dx:\text{Int}) \text{wrap}(P_1, s.x := s.x + dx) : \text{Int}\rightarrow P_1 & \text{(Val } \zeta\text{Object)} \\ \emptyset \vdash [x = 0, mv_x = \zeta(s:P_1(P_1)) \lambda(dx:\text{Int}) \text{wrap}(P_1, s.x := s.x + dx)] : P_1(P_1) & \text{(Val Object)} \\ \emptyset \vdash P_1 <: P_1 & \text{by (Sub Refl)} \\ \emptyset \vdash \text{wrap}(\text{Self} = P_1)[x = 0, mv_x = \zeta(s:P_1(\text{Self})) \lambda(dx:\text{Int}) \text{wrap}(\text{Self}, s.x := s.x + dx)] : P_1 & \text{(Val } \zeta\text{Object)} \end{array} \right. & \end{array} \right.$$

The rule (Val ζ Select) allows us to invoke methods whose type involves *Self*:

$$\text{origin}_1.mv_x : \text{Int}\rightarrow P_1$$

Moreover, the equational theory allows us to derive expected equivalences, such as:

$$\text{origin}_1.mv_x(1) \leftrightarrow$$

$$\text{wrap}(\text{Self} = P_1)[x = 1, mv_x = \zeta(s:P_1(\text{Self})) \lambda(dx:\text{Int}) \text{wrap}(\text{Self}, s.x := s.x + dx)] : P_1$$

that is, the unit point equals the result of moving the origin point.

15.2.2 Backup Methods

This is the typed version of the example in Section 6.5.2. The basic type in question is:

$$Bk \triangleq \zeta(Self)[retrieve:Self, backup:Self]$$

To show a less trivial example, we define movable points with *backup* and *retrieve* methods:

$$BkP_1 \triangleq \zeta(Self)[retrieve:Self, backup:Self, x:Int, mv_x:Int \rightarrow Self]$$

Then we have both $BkP_1 <: P_1$ and $BkP_1 <: Bk$. Using the same techniques as in the previous example we obtain the following typing:

$$\begin{aligned} o : BkP_1 &\triangleq \\ &\text{wrap}(Self=BkP_1) \\ &[\text{retrieve} = \zeta(s:BkP_1(Self)) \text{wrap}(Self, s), \\ &\quad \text{backup} = \zeta(s:BkP_1(Self)) \\ &\quad \quad \text{wrap}(Self, s.\text{retrieve} \Leftarrow \zeta(s':BkP_1(Self)) \text{wrap}(Self, s')), \\ &\quad x = \dots, \\ &\quad mv_x = \dots] \end{aligned}$$

15.2.3 Booleans

In preparation for the next example, we show an encoding that uses quantified types.

As we saw in Section 7.4.2, the booleans can be typed as follows:

$$Bool_A \triangleq [if.A, then:A, else:A]$$

The definition depends on a result type A for conditional expressions.

We can achieve a more uniform typing by using quantifiers, so that a single polymorphic type can account for all uses of booleans:

$$\begin{aligned} Bool &\triangleq \forall(X)[if:X, then:X, else:X] \\ true &\triangleq \\ &\lambda(X)[if = \zeta(x:Bool_X) x.\text{then}, then = \zeta(x:Bool_X) x.\text{then}, else = \zeta(x:Bool_X) x.\text{else}] \\ false &\triangleq \\ &\lambda(X)[if = \zeta(x:Bool_X) x.\text{else}, then = \zeta(x:Bool_X) x.\text{then}, else = \zeta(x:Bool_X) x.\text{else}] \\ if_A b \text{ then } c \text{ else } d &\triangleq \\ &((b(A).\text{then} \Leftarrow \zeta(x:Bool_A)c).\text{else} \Leftarrow \zeta(x:Bool_A)d).\text{if} && x \notin FV(c) \cup FV(d) \end{aligned}$$

15.2.4 Object-Oriented Natural Numbers

This is the typed version of the example in Section 6.5.3. The type of natural numbers is quite interesting: it contains a universal quantification of the form $\forall(Z) Z \rightarrow (X \rightarrow Z) \rightarrow Z$, with a free X in covariant position.

$$N_{Ob} \triangleq \zeta(Self)[succ:Self, case:\forall(Z) Z \rightarrow (Self \rightarrow Z) \rightarrow Z]$$

The *zero* numeral can then be typed as follows:

$$\begin{aligned} \text{zero}_{\text{Ob}} : N_{\text{Ob}} &\triangleq \\ &\text{wrap}(\text{Self} = N_{\text{Ob}}) \\ &[\text{case} = \lambda(Z) \lambda(z:Z) \lambda(f:\text{Self} \rightarrow Z) z, \\ &\quad \text{succ} = \zeta(n:N_{\text{Ob}}(\text{Self})) \\ &\quad \text{wrap}(\text{Self}, n.\text{case} := \lambda(Z) \lambda(z:Z) \lambda(f:\text{Self} \rightarrow Z) f(\text{wrap}(\text{Self}, n)))] \end{aligned}$$

The operations are:

$$\begin{aligned} \text{succ} : N_{\text{Ob}} \rightarrow N_{\text{Ob}} &\triangleq \lambda(n:N_{\text{Ob}}) n.\text{succ} \\ \text{pred} : N_{\text{Ob}} \rightarrow N_{\text{Ob}} &\triangleq \lambda(n:N_{\text{Ob}}) n.\text{case}(N_{\text{Ob}})(\text{zero}_{\text{Ob}})(\lambda(p:N_{\text{Ob}}) p) \\ \text{iszero} : N_{\text{Ob}} \rightarrow \text{Bool} &\triangleq \lambda(n:N_{\text{Ob}}) n.\text{case}(\text{Bool})(\text{true})(\lambda(p:N_{\text{Ob}}) \text{false}) \end{aligned}$$

Looking back at Section 9.4, we can now explain the correspondence between our second-order typing for numerals and the typing based on sums. The type for the *case* method, *Unit+Self*, can be encoded as $\forall(Z)Z \rightarrow (\text{Self} \rightarrow Z) \rightarrow Z$, since in general $A+B$ can be encoded as $\forall(Z)(A \rightarrow Z) \rightarrow (B \rightarrow Z) \rightarrow Z$. Using this encoding, we can view the definition of Section 9.4 as a variant of the one given here.

15.2.5 A Calculator

This is the typed version of the example in Section 6.5.4. Once the code is properly annotated, the calculator has type:

$$C \triangleq \zeta(\text{Self})[\text{arg}, \text{acc}:\text{Real}, \text{enter}:\text{Real} \rightarrow \text{Self}, \text{add}, \text{sub}:\text{Self}, \text{equals}:\text{Real}]$$

The code is:

$$\begin{aligned} \text{calculator} : C &\triangleq \\ &\text{wrap}(\text{Self} = C) \\ &[\text{arg} = 0.0, \\ &\quad \text{acc} = 0.0, \\ &\quad \text{enter} = \zeta(s:C(\text{Self})) \lambda(n:\text{Real}) \text{wrap}(\text{Self}, s.\text{arg} := n), \\ &\quad \text{add} = \zeta(s:C(\text{Self})) \\ &\quad \text{wrap}(\text{Self}, (s.\text{acc} := s.\text{equals}).\text{equals} \Leftarrow \zeta(s':C(\text{Self})) s'.\text{acc} + s'.\text{arg}), \\ &\quad \text{sub} = \zeta(s:C(\text{Self})) \\ &\quad \text{wrap}(\text{Self}, (s.\text{acc} := s.\text{equals}).\text{equals} \Leftarrow \zeta(s':C(\text{Self})) s'.\text{acc} - s'.\text{arg}), \\ &\quad \text{equals} = \zeta(s:C(\text{Self})) s.\text{arg}] \end{aligned}$$

By subsumption, the calculator also has type:

$$\text{Calc} \triangleq \zeta(\text{Self})[\text{enter}:\text{Real} \rightarrow \text{Self}, \text{add}, \text{sub}:\text{Self}, \text{equals}:\text{Real}]$$

This shorter *Calc* type is the one shown to users of the calculator.

A scientific calculator could also be defined with additional state and operations. Its inner design might be quite different from that of our basic calculator, but the scientific calculator's type could still be a subtype of *Calc*.

15.2.6 Storage Cells

This is the typed version of the example in Section 6.5.5. The types for cells and restorable cells are:

$$\begin{aligned} \text{Cell} &\triangleq \zeta(\text{Self})[\text{contents:Nat}, \text{get:Nat}, \text{set:Nat} \rightarrow \text{Self}] \\ \text{ReCell} &\triangleq \zeta(\text{Self})[\text{contents:Nat}, \text{get:Nat}, \text{set:Nat} \rightarrow \text{Self}, \text{backup:Nat}, \text{restore:Self}] \\ \text{ReCell}' &\triangleq \zeta(\text{Self})[\text{contents:Nat}, \text{get:Nat}, \text{set:Nat} \rightarrow \text{Self}, \text{restore:Self}] \end{aligned}$$

where $\text{ReCell} <: \text{ReCell}' <: \text{Cell}$.

The code for cells and restorable cells is typed as follows:

$$\begin{aligned} \text{myCell : Cell} &\triangleq \\ &\quad \text{wrap}(\text{Self}=\text{Cell}) \\ &\quad [\text{contents} = 0, \\ &\quad \text{get} = \zeta(s:\text{Cell}(\text{Self})) s.\text{contents}, \\ &\quad \text{set} = \zeta(s:\text{Cell}(\text{Self})) \lambda(n:\text{Nat}) \text{wrap}(\text{Self}, s.\text{contents} := n)] \\ \text{myReCell : ReCell} &\triangleq \\ &\quad \text{wrap}(\text{Self}=\text{ReCell}) \\ &\quad [\text{contents} = 0, \\ &\quad \text{get} = \zeta(s:\text{ReCell}(\text{Self})) s.\text{contents}, \\ &\quad \text{set} = \zeta(s:\text{ReCell}(\text{Self})) \lambda(n:\text{Nat}) \\ &\quad \quad \text{wrap}(\text{Self}, (s.\text{backup} := s.\text{contents}).\text{contents} := n), \\ &\quad \text{backup} = 0, \\ &\quad \text{restore} = \zeta(s:\text{ReCell}(\text{Self})) \text{wrap}(\text{Self}, s.\text{contents} := s.\text{backup})] \\ \text{myOtherReCell : ReCell'} &\triangleq \\ &\quad \text{wrap}(\text{Self}=\text{ReCell}') \\ &\quad [\text{contents} = 0, \\ &\quad \text{get} = \zeta(s:\text{ReCell}'(\text{Self})) s.\text{contents}, \\ &\quad \text{set} = \zeta(s:\text{ReCell}'(\text{Self})) \lambda(n:\text{Nat}) \\ &\quad \quad \text{wrap}(\text{Self}, \\ &\quad \quad (s.\text{restore} \neq \zeta(z:\text{ReCell}'(\text{Self})) \text{wrap}(\text{Self}, z.\text{contents} := s.\text{contents})) \\ &\quad \quad .\text{contents} := n), \\ &\quad \text{restore} = \zeta(s:\text{ReCell}'(\text{Self})) \text{wrap}(\text{Self}, s.\text{contents} := 0)] \end{aligned}$$

15.3 Binary Methods and the Covariance Requirement

A typical example of a binary method is an equality method in a point object. Such a method has a parameter of the same type as its self variable, so we might naturally assign it the type `Self` [36]. We could write points with equality as follows:

$$P_{1eq} \triangleq \zeta(\text{Self})[x:\text{Int}, mv_x:\text{Int} \rightarrow \text{Self}, eq:\text{Self} \rightarrow \text{Bool}]$$

$$\begin{aligned} \text{origin}_{1\text{eq}} &\triangleq \\ &\text{wrap}(\text{Self} = P_{1\text{eq}})[x = 0, mv_x = \dots, eq = \zeta(s: P_{1\text{eq}}(\text{Self})) \lambda(p: \text{Self}) s.x =_{\text{Int}} p.x] \end{aligned}$$

Binary methods violate the covariance requirement of ζ -object types. We temporarily ignore this requirement, which is just a convention. We use the rules for the ζ quantifier, which do not depend on covariance, instead of the derived rules for ζ -objects.

We discover that binary methods cannot be invoked effectively because of typing restrictions. Expanding the encoding of method invocation from Section 15.1, we may try to pull out the eq method from a ζ -object p of type $P_{1\text{eq}}$:

$$\begin{aligned} &\text{use } p \text{ as } \text{Self} <: P_{1\text{eq}}, z: [x: \text{Int}, mv_x: \text{Int} \rightarrow \text{Self}, eq: \text{Self} \rightarrow \text{Bool}] \\ &\quad \text{in } z.eq : P_{1\text{eq}} \rightarrow \text{Bool} \quad \cdot \end{aligned} \quad (\text{not typable})$$

The rule (Val Use) requires that $z.eq$ be given a type independent of Self . This is not possible because Self is in contravariant position in $\text{Self} \rightarrow \text{Bool}$. (In contrast, we can pull out $z.mv_x : \text{Int} \rightarrow P_{1\text{eq}}$ because $\text{Int} \rightarrow \text{Self} <: \text{Int} \rightarrow P_{1\text{eq}}$.)

Although we cannot pull out the eq method from p , we can still apply it within the scope of the *use* construct if we can find an adequate argument. One possibility is:

$$\begin{aligned} &\text{use } p \text{ as } \text{Self} <: P_{1\text{eq}}, z: [x: \text{Int}, mv_x: \text{Int} \rightarrow \text{Self}, eq: \text{Self} \rightarrow \text{Bool}] \\ &\quad \text{in } z.eq(z.mv_x(1)) : \text{Bool} \end{aligned}$$

which compares p with its translation by 1, returning *false*. However, most advantageous uses of eq would involve comparisons with independently obtained points, and these comparisons are not possible. This situation arises, essentially, because independent instances of the same existential type do not intermix.

To avoid the contravariant occurrence of Self , we may define an alternative to $P_{1\text{eq}}$:

$$\begin{aligned} P_1 &\triangleq \zeta(\text{Self})[x: \text{Int}, mv_x: \text{Int} \rightarrow \text{Self}] \\ P_{1\text{eq}'} &\triangleq \zeta(\text{Self})[x: \text{Int}, mv_x: \text{Int} \rightarrow \text{Self}, eq: P_1 \rightarrow \text{Bool}] \\ \text{origin}_{1\text{eq}'} &\triangleq \\ &\text{wrap}(\text{Self} = P_{1\text{eq}'}')[x = 0, mv_x = \dots, eq = \zeta(s: P_{1\text{eq}'}(\text{Self})) \lambda(p: P_1) s.x =_{\text{Int}} p.x] \end{aligned}$$

These definitions are more useful than the previous ones. In particular, $\text{origin}_{1\text{eq}}.eq(p)$ is well-typed whenever p has type P_1 . However, eq is no longer a binary method because its argument does not have the same type as self .

Binary methods are essentially incompatible with subsumption. In contrast, the purpose of covariant Self types is to work well with subsumption. We may conclude that Self types as we have defined them are not well suited for modeling binary methods. We discuss the handling of binary methods again in Part III.

15.4 Classes and Inheritance, with Self

We now extend the techniques developed in Section 8.5 to model classes and inheritance. We take into account Self, and the need for pre-methods and classes to be sufficiently polymorphic.

If $A \equiv \zeta(\text{Self})[l_i; B_i\{\text{Self}^+\}_{i \in 1..n}]$ is an object type, then:

$$\text{Class}(A) \triangleq [\text{new}:A, l_i; \forall(\text{Self} <: A) A(\text{Self}) \rightarrow B_i\{\text{Self}\}_{i \in 1..n}]$$

can be seen as a class type. This is the type of a collection of pre-methods that can be used in objects of type A and reused in other classes. Each pre-method is required to work for an arbitrary subtype Self of A .

As before, $A' <: A$ does not imply $\text{Class}(A') <: \text{Class}(A)$. Hence we say:

$$\text{Class}(A') \text{ may inherit from } \text{Class}(A) \text{ iff } A' <: A$$

If $\text{Class}(A')$ may inherit from $\text{Class}(A)$, then A' and A have the forms $A' \equiv \zeta(\text{Self})[l_i; B_i\{\text{Self}^+\}_{i \in 1..n+m}]$ and $A \equiv \zeta(\text{Self})[l_i; B_i\{\text{Self}^+\}_{i \in 1..n}]$. Thus, since $A' <: A$, we obtain $\forall(\text{Self} <: A) A(\text{Self}) \rightarrow B_i\{\text{Self}\}_{i \in 1..n} <: \forall(\text{Self} <: A') A'(\text{Self}) \rightarrow B_i\{\text{Self}\}_{i \in 1..n}$. Hence pre-methods of members of $\text{Class}(A)$ may be reused for members of $\text{Class}(A')$. Note that this reuse property would not hold if we took $A \rightarrow B_i\{A\}$ and $A' \rightarrow B_i\{A'\}$ as the types of the pre-methods. The use of polymorphic types for pre-methods is essential.

For a class of type $\text{Class}(A)$, we can write the body of the method new as follows:

$$\zeta(z:\text{Class}(A)) \text{wrap}(\text{Self}=A)[l_i = \zeta(s:A(\text{Self})) z.l_i(\text{Self})(s)_{i \in 1..n}]$$

There remains a serious problem in defining classes: writing pre-methods of types $\forall(\text{Self} <: A) A(\text{Self}) \rightarrow B_i\{\text{Self}\}$. For example, for the type of points P_1 from Section 15.2.1, we would have to write pre-methods for the class type:

$$\begin{aligned} \text{Class}(P_1) \triangleq \\ [\text{new}:P_1, \\ x:\forall(\text{Self} <: P_1) P_1(\text{Self}) \rightarrow \text{Int}, \\ \text{mv_}x:\forall(\text{Self} <: P_1) P_1(\text{Self}) \rightarrow \text{Int} \rightarrow \text{Self}] \end{aligned}$$

Writing a pre-method of type $\forall(\text{Self} <: P_1) P_1(\text{Self}) \rightarrow \text{Int}$ is easy, but writing a useful pre-method of type $\forall(\text{Self} <: P_1) P_1(\text{Self}) \rightarrow \text{Int} \rightarrow \text{Self}$ is impossible in the present system. Not even a pre-method that returns self has this type. This difficulty is related to the problem of updating methods that return self, discussed next. Solutions are described in Sections 15.6 and 16.6.

15.5 Updating from the Outside

In Section 15.1 we remark that it is easy to create a ζ -object, because its initial methods need to work only for the actual type of the object being constructed. In particular, methods that update self present no difficulties; see the examples of Section 15.2.

In contrast, updating methods from the outside is challenging. If we want to update a method of an existing ζ -object c of type A , the new method must work for any possible subtype B of A , because c might have been built as an element of B . We know neither the “true type” of c , nor the “true type” of the self parameters of its methods. When updating a method of c , the updating method can assume only that the object has been constructed from an unknown subtype of A .

If the type of the updated method is the *Self* type, it is critical that the new method return a result of the “true type” of c , because one of the other methods may be invoked on the result. Therefore the new method must return *self* or a modified *self*. As in Section 9.6, we say that the new method must be parametric in *Self*.

This is where we need the complex rule for updating ζ -objects, (Val ζ Update), which was not used in Section 15.2 even when typing update operations. Consider, for example, the type:

$$A \triangleq \zeta(\text{Self})[n,f:\text{Int}, m:\text{Self}] \quad \text{with } A(X) \equiv [n,f:\text{Int}, m:X]$$

An updating method for $a \equiv \text{wrap}(Y <: A = C)b[Y]$ can use in its body the variables $\text{Self} <: A$, $x' : A(\text{Self})$, and $x : A(\text{Self})$, where x' is in fact $b[\mathbb{C}]$, according to (Eval ζ Update), and x is the self of the new method. We can therefore update the f method of A with methods that produce integers, in any of the following ways.

$$a.f \triangleq (\text{Self} <: A, x' : A(\text{Self})) \zeta(x : A(\text{Self})) 3 \\ \text{setting } a.f \text{ to produce 3 (constantly)}$$

$$a.f \triangleq (\text{Self} <: A, x' : A(\text{Self})) \zeta(x : A(\text{Self})) x'.n + 1 \\ \text{setting } a.f \text{ to produce } b.n + 1 \text{ (constantly)}$$

$$a.f \triangleq (\text{Self} <: A, x' : A(\text{Self})) \zeta(x : A(\text{Self})) x.n + 1 \\ \text{setting } a.f \text{ to produce } 1 + \text{the value of } n \text{ when } f \text{ is invoked}$$

Let us now attempt to update the m method. The typing rule (Val ζ Update) requires that, from the variables $\text{Self} <: A$, $x' : A(\text{Self})$, $x : A(\text{Self})$ at its disposal, the updating method must produce a value of type *Self*. Here are some possibilities:

$$a.m \triangleq (\text{Self} <: A, x' : A(\text{Self})) \zeta(x : A(\text{Self})) x'.m \\ \text{setting } a.m \text{ to produce the current } b.m \text{ (constantly)}$$

$$a.m \triangleq (\text{Self} <: A, x' : A(\text{Self})) \zeta(x : A(\text{Self})) x.m \\ \text{setting } a.m \text{ to diverge}$$

However, we cannot update $a.m$ with anything useful. Note, first, that we cannot synthesize a value of type *Self* from scratch. Second, we cannot return x' or x , nor $\text{wrap}(A, x')$ or $\text{wrap}(A, x)$, because none of these can be given type *Self*. Third, any update to x' , x , $\text{wrap}(A, x')$, or $\text{wrap}(A, x)$ preserves their original type, so we cannot return the updated terms either. Finally, $\text{wrap}(\text{Self}, x) : \text{Self}$ is not derivable, for an unknown $\text{Self} <: A$; see (Val ζ Object).

Moreover, it would be unsound to ignore these typing problems and return, say, $\text{wrap}(\text{Self}, x)$ or $\text{wrap}(\text{Self}, x')$. The reason can be seen in the following example, which builds a ζ -object r_2 from a proper subtype R_1 of its own type R_2 :

$$\begin{aligned} R_2 &\triangleq \zeta(\text{Self})[p : \text{Self}, q : \text{Int}] \\ R_1 <: R_2 &\triangleq \zeta(\text{Self})[p : \text{Self}, q : \text{Int}, t : \text{Int}] \\ r_1 : R_1 &\triangleq \text{wrap}(Y <: R_1 = R_1)[p = \zeta(s : R_1(Y)) \text{wrap}(Y, s), q = 0, t = 0] \\ r_2 : R_2 &\triangleq \text{wrap}(Y <: R_2 = R_1)[p = r_1, q = \zeta(s : R_2(Y)) s.p.t] \end{aligned}$$

$$\begin{aligned}
 r_2.p &\Leftarrow (\text{Self} <: R_2, x' : R_2(\text{Self})) \zeta(x : R_2(\text{Self})) \text{wrap}(\text{Self}, x) \\
 &= \text{wrap}(Y <: R_2 = R_1)[p = \zeta(x : R_2(Y)) \text{wrap}(Y, x), q = \zeta(s : R_2(Y)) s.p.t] \quad (\text{unsound}) \\
 r_2.p &\Leftarrow (\text{Self} <: R_2, x' : R_2(\text{Self})) \zeta(x : R_2(\text{Self})) \text{wrap}(\text{Self}, x') \\
 &= \text{wrap}(Y <: R_2 = R_1) \\
 &[p = \text{wrap}(Y, [p = r_1, q = \zeta(s : R_2(Y)) s.p.t]), q = \zeta(s : R_2(Y)) s.p.t] \quad (\text{unsound})
 \end{aligned}$$

Unsound behavior can be observed by invoking q after either of these two updates, because the field t is missing from $s.p$.

The reason for this unsound behavior can be traced back to the rule for constructing ζ -objects. Because of the flexibility we have in constructing ζ -objects out of proper subtypes of themselves, the parameters x and x' at our disposal when updating may be shorter than the subtype used originally when constructing the object. For example, r_2 is constructed with a component p that is longer than the body of r_2 , to which x or x' would be bound. Therefore returning x or x' would not be safe because they would be too short.

In conclusion, we discover that the rule (Val ζ Update), although very powerful for updating simple methods and fields, is not sufficient to allow us to update methods that return a value of type Self. We are in the strange but apparently inevitable position of being able to construct a point with a mv_x method, but unable then to update mv_x with an identical method. One solution to this problem is discussed in the next section.

15.6 Recoup

In this section we introduce a special method called *recoup* with an associated run-time invariant. A *recoup* method is a method that returns self immediately. The invariant asserts that the result of a *recoup* method is its host object. These simple definitions have a number of surprising consequences.

15.6.1 The Recoup Method

Let us consider the type:

$$R \triangleq \zeta(\text{Self})[r:\text{Self}]$$

We can build an element of R as follows:

$$\left\{
 \begin{array}{ll}
 \begin{array}{l}
 \not E, s:[r:R] \vdash s : [r:R] \\
 E, s:[r:R] \vdash \text{wrap}(Z <: R = R)s : R \\
 \Downarrow \equiv E, s:[r:R] \vdash \text{wrap}(R,s) : R
 \end{array} & \text{by (Val } x\text{)} \\
 E \vdash [r = \zeta(s:[r:R])\text{wrap}(R,s)] : [r:R] & \text{(standard abbreviation)} \\
 E \vdash \text{wrap}(\text{Self} <: R = R)[r = \zeta(s:[r:\text{Self}])\text{wrap}(\text{Self},s)] : R & \text{(Val Wrap)} \\
 \equiv E \vdash \text{wrap}(\text{Self} = R)[r = \zeta(s:R(\text{Self}))\text{wrap}(\text{Self},s)] : R & \text{(Val Object)}
 \end{array}
 \right.$$

(Val Wrap)

(Val Object)

(Val Wrap)

(standard abbreviation)

We say that a method of the form $\zeta(s : A(\text{Self}))\text{wrap}(\text{Self},s)$, in the context of a ζ -object of the form $\text{wrap}(\text{Self} = A)[\dots]$, is a *recoup method*. The presence of a *recoup* method in a

ζ -object implies that the representation type of the object is the same as the true type of the object, rather than a proper subtype of the true type.

The backup example from Section 15.2.2 defined the type:

$$Bk \triangleq \zeta(Self)[retrieve:Self, backup:Self]$$

As we have just discussed, we cannot usefully update the *retrieve* or *backup* methods of *Bk*, or any method with result type *Self*. This is because we are not able to produce a useful element of type *Self* from an element $y : [retrieve:Self, backup:Self]$ as would be required by the (Val ζ Update) rule, other than $y.retrieve$ and $y.backup$. We now introduce a technique that solves this problem.

We add a new component, called *recoup*, to the type *Bk*, and consider objects where *recoup* is set to $\zeta(s:Bk(Bk))wrap(Bk,s)$. We redefine:

$$Bk \triangleq \zeta(Self)[recoup:Self, retrieve:Self, backup:Self]$$

$$a : Bk \triangleq wrap(Self=Bk) b$$

where $b \equiv$

$$[recoup = \zeta(s:Bk(Self)) wrap(Self, s),$$

$$retrieve = \zeta(s:Bk(Self)) wrap(Self, s),$$

$$backup = \zeta(s:Bk(Self)) wrap(Self, s).retrieve := wrap(Self, s)]$$

Then the following method update typechecks, since $s'.recoup$ has type *Self*:

$$a.retrieve \Leftarrow (Self <: Bk, s':Bk(Self)) \zeta(s:Bk(Self)) s'.recoup$$

Moreover, the behavior obtained is a useful one, corresponding to performing a backup “by hand” instead of calling the *backup* method to do it. We calculate:

$$(wrap(Self=Bk) b).retrieve \Leftarrow (Self <: Bk, s':Bk(Self)) \zeta(s:Bk(Self)) s'.recoup$$

$$= wrap(Self=Bk) b.retrieve \Leftarrow \zeta(s:Bk(Self)) b.recoup$$

$$= wrap(Self=Bk) b.retrieve \Leftarrow \zeta(s:Bk(Self)) a$$

$$= wrap(Self=Bk)$$

$$[recoup = \zeta(s:Bk(Self)) wrap(Self, s), retrieve = a, backup = \dots]$$

Similarly, the following update clears the backup log:

$$a.retrieve \Leftarrow (Self <: Bk, s':Bk(Self)) \zeta(s:Bk(Self)) s.recoup$$

Intuitively, a *recoup* method allows us to recover a “parametric self” $b.recoup$, which equals a but has type *Self* <: *Bk* and not just type *Bk* like a . This technique is particularly useful after an update on a value of type *Bk(Self)*, because the result of the update only has type *Bk(Self)*, but then *recoup* can extract an appropriate result of type *Self*.

15.6.2 The Recoup Invariant

The technique just described gives the correct result only as long as *recoup* is bound to $\zeta(s:A(Self))wrap(Self,s)$. Otherwise the operational behavior is not the expected one. We

must assume that *recoup* is bound to $\zeta(s:A(\text{Self}))\text{wrap}(\text{Self}, s)$ as an invariant. We have no elegant way of enforcing this invariant because *recoup* is a component like any other.

Our ability to reason about programs is dependent on the *recoup* invariant. In the following example, we use the *recoup* invariant to argue that the result of updating an object from the outside is equivalent to the original object. We use the object *a* from Section 15.6.1:

$$\begin{aligned}
 a.\text{backup} &\Leftarrow (\text{Self} <: \text{Bk}, s' : \text{Bk}(\text{Self})) \zeta(s : \text{Bk}(\text{Self})) (s.\text{retrieve} := s.\text{recoup}).\text{recoup} \\
 &= \text{wrap}(\text{Self} = \text{Bk}) b.\text{backup} \Leftarrow \zeta(s : \text{Bk}(\text{Self})) (s.\text{retrieve} := s.\text{recoup}).\text{recoup} \\
 &= \text{wrap}(\text{Self} = \text{Bk}) b.\text{backup} \Leftarrow \zeta(s : \text{Bk}(\text{Self})) (s.\text{retrieve} := \text{wrap}(\text{Self}, s)).\text{recoup} \\
 &\quad (\text{since, by the } \text{recoup} \text{ invariant, we have } s.\text{recoup} = \text{wrap}(\text{Self}, s)) \\
 &= \text{wrap}(\text{Self} = \text{Bk}) b.\text{backup} \Leftarrow \zeta(s : \text{Bk}(\text{Self})) \text{wrap}(\text{Self}, s).s.\text{retrieve} := \text{wrap}(\text{Self}, s) \\
 &\quad (\text{since, by the } \text{recoup} \text{ invariant, we have} \\
 &\quad \quad (s.\text{retrieve} := \text{wrap}(\text{Self}, s)).\text{recoup} = \text{wrap}(\text{Self}, s.s.\text{retrieve} := \text{wrap}(\text{Self}, s))) \\
 &= a
 \end{aligned}$$

The correctness of typing, on the other hand, does not depend on the *recoup* invariant.

An invariant of this kind may be acceptable for a programming language: *recoup* would be a distinguished component that is appropriately initialized and that cannot be updated. Even without language support, we may be disciplined enough to preserve the *recoup* invariant and thus we may solve the problem of updating methods with result type *Self*. If we initialize *recoup* correctly and do not update it, then the operational semantics works out as expected. However, it may be hard to incorporate the *recoup* invariant in an equational theory for the programming language.

15.6.3 The Recoup Method and Classes

If a type *A* has the form $\zeta(\text{Self})[\text{recoup}:\text{Self}, \dots]$, then we can write useful polymorphic functions of type:

$$\forall(\text{Self} <: \text{A})\text{A}(\text{Self}) \rightarrow \text{Self}$$

that are not available without *recoup*, for example:

$$\begin{aligned}
 f : \forall(\text{Self} <: \text{Bk})\text{Bk}(\text{Self}) \rightarrow \text{Self} &\triangleq \\
 \lambda(\text{Self} <: \text{Bk}) \lambda(s : \text{Bk}(\text{Self})) (s.\text{retrieve} := s.\text{recoup}).\text{recoup}
 \end{aligned}$$

Such functions are parametric enough to be used in updates from the outside, as in:

$$a.\text{retrieve} \Leftarrow (\text{Self} <: \text{Bk}, s' : \text{Bk}(\text{Self})) \zeta(s : \text{Bk}(\text{Self})) f(\text{Self})(s)$$

and can also be used for defining classes. Extending the techniques of Section 15.4, we can take advantage of *recoup* to write a class for points and, in particular, we can write a pre-method for *mv_x*:

$$P_1 \triangleq \zeta(\text{Self})[\text{recoup}:\text{Self}, x:\text{Int}, \text{mv}_x:\text{Int} \rightarrow \text{Self}]$$

```


$$\begin{aligned}
Class(P_1) &\triangleq \\
&[new:P_1, \\
&\quad x:\forall(Self <: P_1) P_1(Self) \rightarrow Int, \\
&\quad mv\_x:\forall(Self <: P_1) P_1(Self) \rightarrow Int \rightarrow Self] \\
p_1\text{-}class : Class(P_1) &\triangleq \\
&[new = \zeta(z:Class(P_1)) wrap(Self=P_1) \\
&\quad [recoup = \zeta(s:P_1(Self)) wrap(Self, s), \\
&\quad \quad x = \zeta(s:P_1(Self)) z.x(Self)(s), \\
&\quad \quad mv\_x = \zeta(s:P_1(Self)) z.mv\_x(Self)(s)], \\
&\quad x = \lambda(Self <: P_1) \lambda(s:P_1(Self)) 0, \\
&\quad mv\_x = \lambda(Self <: P_1) \lambda(s:P_1(Self)) \lambda(dx:Int) (s.x := s.x+dx).recoup]
\end{aligned}$$


```

Note that $p_1\text{-}class$ does not include a pre-method for *recoup*, although it could. Instead, the *recoup* method is simply built into objects by *new*.

This code achieves the desired effect, but is rather complex and may seem ad hoc. Therefore, starting in the next section, we investigate alternatives to the *recoup* technique.

15.7 Objects with Structural Invariants

So far we have worked with type and equational theories that are valid in standard denotational models, with a standard interpretation of subtyping (see Chapter 14). However, if we imagine a programming language including our objects, we see that additional important properties may be satisfied by the operational semantics of the language that are not satisfied in standard models. These additional properties give us an alternative way to handle methods that produce results of type *Self*.

In this section we provide plausibility arguments as to why these properties may be operationally sound, based on our current understanding. In Chapter 16 we incorporate these properties into rules, and provide an operational semantics and a proof of subject reduction for these rules.

15.7.1 Towards Structural Rules

The additional properties we wish to capture are:

- A *structural subtyping invariant*.
- An *object shape invariant*.

Informally, the structural subtyping invariant is that every subtype of an object type is a longer object type. The object shape invariant is that all elements of an object type have a certain shape. An example of a shape invariant, from Section 15.6, is that all objects have a *recoup* method of a given form. In this section we study a different shape invariant, along with its typing and equational consequences.

Both structural subtyping and shape invariants must be assumed to achieve the desired effect. Collectively we refer to both invariants as *structural invariants*.

Structural invariants have important consequences for typing, as we already discussed at the end of Section 13.1. In particular, the structural invariants discussed in this section support stronger type rules for objects with Self. We call *structural* the rules that exploit the structural invariants. Most of the examples discussed so far can already be typed with the standard rules, but the stronger structural rules provide more natural typings. Other examples that could be typed only using *recoup* can now be typed with the structural rules, again in a more natural way. In particular, it becomes easy to update methods that return self.

Our goal is to define a new calculus with “structural object types” and “structural objects”. The calculus supports the stronger structural rules. To justify these rules, we provide an interpretation of structural objects into our ζ -objects that establishes a particular shape invariant, namely that the interpretation of every structural object has a certain shape. We provide an interpretation of the operations on structural objects that preserves the invariant under reduction. We show, within our standard formal system, that the structural rules are derivable in the presence of side conditions about the shape of terms and types.

15.7.2 Structural Objects

We define a new syntax for object types, to indicate those types satisfying the structural subtyping invariant. We also define new syntax for objects, to indicate those objects satisfying a particular shape invariant. We give new syntax for object operations as well; the new operations take advantage of the shape invariant.

$Obj(X)[l_i:B_i\{X\}^{i \in 1..n}]$	structural object type
$obj(X=C)[l_i=\zeta(x_i:X)b_i\{X,x_i\}^{i \in 1..n}]$	structural object
$a.l_j$	structural method invocation
$a.l_i \Leftarrow (Y <: A, y: Y)\zeta(x: Y)b_i(Y, y, x)$	structural method update

We define:

$$\begin{aligned} \text{For } C &\equiv Obj(X)[l_i:B_i\{X\}^{i \in 1..n}], a \equiv obj(X=C)[l_i=\zeta(x_i:X)b_i\{X,x_i\}^{i \in 1..n}], \text{ and fresh } x'_i: \\ C(D) &\triangleq [l_i:B_i\{D\}^{i \in 1..n}] \\ a(D) &\triangleq [l_i=\zeta(x_i':C(D))b_i\{D, \text{wrap}(D, x_i')\}^{i \in 1..n}] \end{aligned}$$

Our intended interpretation of this new syntax is in terms of ζ -objects:

$$\begin{aligned} Obj(X)[l_i:B_i\{X\}^{i \in 1..n}] &\triangleq \zeta(X)[l_i:B_i\{X^+\}^{i \in 1..n}] \\ obj(X=C)[l_i=\zeta(x_i:X)b_i\{X,x_i\}^{i \in 1..n}] &\triangleq \\ &\quad \text{wrap}(X <: C = C)[l_i=\zeta(x_i':C(X))b_i\{X, \text{wrap}(X, x_i')\}^{i \in 1..n}] \\ &\quad \text{for } C \equiv Obj(X)[l_i:B_i\{X\}^{i \in 1..n}] \end{aligned}$$

$$\begin{aligned}
 a.l_j &\triangleq \\
 &a(C).l_j \\
 &\text{for } C \equiv Obj(X)[l_i:B_i\{X\}^{i \in 1..n}], a \equiv obj(X=C)[l_i=\zeta(x_i:X)b_i\{X,x_i\}^{i \in 1..n}], j \in 1..n \\
 a.l_i \Leftarrow (Y<:A,y:Y)\zeta(x:Y)b\{Y,y,x\} &\triangleq \\
 &wrap(Y<:C=C)a(Y).l_i \Leftarrow \zeta(x':C(Y))b\{Y,a.wrap(Y,x')\} \\
 &\text{for } C \equiv Obj(X)[l_i:B_i\{X\}^{i \in 1..n}], a \equiv obj(X=C)[l_i=\zeta(x_i:X)b_i\{X,x_i\}^{i \in 1..n}], j \in 1..n
 \end{aligned}$$

Note the special properties of ζ -objects that correspond to structural objects: the external and internal types of $wrap(X<:C=C)[l_i=\zeta(x_i:C(X))b_i\{X,wrap(X,x_i)\}^{i \in 1..n}]$ are the same (C), and all occurrences of self (x_i') in the body of methods are in the context $wrap(C,x_i')$.

Thus, intuitively, the shape invariant is that every object has the form:

$$wrap(X<:C=C)[l_i=\zeta(x_i:C(X))b_i\{X,wrap(X,x_i)\}^{i \in 1..n}]$$

and the structural subtyping invariant is that every subtype of $Obj(X)[l_i:B_i\{X\}^{i \in 1..n}]$ is a longer object type, of the form $Obj(X)[l_i:B_i\{X\}^{i \in 1..n+m}]$.

15.7.3 Rules with Preliminary Structural Assumptions

Given these definitions, we can derive the following rules. We use the fact that if $obj(X=C)[l_i=\zeta(x_i:X)b_i\{X\}^{i \in 1..n}]$ has some type A , then it also has type C , and C is a subtype of A .

Objects, with preliminary structural assumptions

(Type Object')

$$E, X \vdash B_i\{X^+\} \quad \forall i \in 1..n$$

$$\frac{}{E \vdash Obj(X)[l_i:B_i\{X\}^{i \in 1..n}]}$$

(Sub Object')

$$E, X \vdash B_i\{X^+\} \quad \forall i \in 1..n+m$$

$$\frac{}{E \vdash Obj(X)[l_i:B_i\{X\}^{i \in 1..n+m}] \ll Obj(X)[l_i:B_i\{X\}^{i \in 1..n}]}$$

(Val Object') (where $A \equiv Obj(X)[l_i:B_i\{X\}^{i \in 1..n}]$)

$$E, x_i:A \vdash b_i\{A\} : B_i\{A\} \quad \forall i \in 1..n$$

$$\frac{}{E \vdash obj(X=A)[l_i=\zeta(x_i:X)b_i\{X\}^{i \in 1..n}] : A}$$

(Val Select') (where $A \equiv Obj(X)[l_i:B_i\{X\}^{i \in 1..n}]$, $a \equiv obj(X=C)[l_i=\zeta(x_i:X)b_i\{X,x_i\}^{i \in 1..n+m}]$)

$$E \vdash a : A \quad j \in 1..n$$

$$\frac{}{E \vdash a.l_j : B_j\{A\}}$$

(Val Update') (where $A \equiv Obj(X)[l_i; B_i\{X\}^{i \in 1..n}]$, $a \equiv obj(X=C)[l_i=\zeta(x_i; X)b_i\{X, x_i\}^{i \in 1..n+m}]$)

$$\frac{E \vdash a : A \quad E, Y <: A, y:Y, x:Y \vdash b : B_j\{Y\} \quad j \in 1..n}{E \vdash a.l_j \not=(Y <: A, y:Y)\zeta(x:Y)b : A}$$

(Eval Select') (where $A \equiv Obj(X)[l_i; B_i\{X\}^{i \in 1..n}]$, $a \equiv obj(X=C)[l_i=\zeta(x_i; X)b_i\{X, x_i\}^{i \in 1..n+m}]$)

$$\frac{E \vdash a : A \quad j \in 1..n}{E \vdash a.l_j \leftrightarrow b_j\{C, a\} : B_j\{A\}}$$

(Eval Update') (where $A \equiv Obj(X)[l_i; B_i\{X\}^{i \in 1..n}]$, $a \equiv obj(X=C)[l_i=\zeta(x_i; X)b_i\{X, x_i\}^{i \in 1..n+m}]$)

$$\frac{E \vdash a : A \quad E, Y <: A, y:Y, x:Y \vdash b\{Y, y, x\} : B_j\{Y\} \quad j \in 1..n}{\frac{E \vdash a.l_j \not=(Y <: A, y:Y)\zeta(x:Y)b\{Y, y, x\} \leftrightarrow obj(X=C)[l_i=\zeta(x: X)b\{X, a, x\}, l_i=\zeta(x_i: X)b_i\{X, x_i\}^{i \in (1..n+m)-\{j\}}] : A}{}}$$

The shape invariant is established by (Val Object'), assuming it is true of subcomponents. The invariant is assumed explicitly by (Val Select') for a particular term; and preserved if already true of subcomponents. The invariant is also assumed explicitly by (Val Update'), and preserved for the updated object and its subcomponents. The invariant is preserved by reduction, as seen in (Eval Select') and (Eval Update').

15.7.4 Rules with Structural Assumptions

So far we have dealt only with the shape invariant. To exploit the structural subtyping invariant, we consider the following version of the rules, which is trivially derivable from the previous one.

Objects, with structural assumptions

(Val Select'') (where $A \equiv Obj(X)[l_i; B_i\{X\}^{i \in 1..n}]$, $A' \equiv Obj(X)[l_i; B_i\{X\}^{i \in 1..n'}]$,
 $a \equiv obj(X=C)[l_i=\zeta(x_i; X)b_i\{X, x_i\}^{i \in 1..n+m}]$)

$$\frac{E \vdash a : A \quad E \vdash A <: A' \quad j \in 1..n'}{E \vdash a.l_j : B_j\{A\}}$$

(Val Update'') (where $A \equiv Obj(X)[l_i; B_i\{X\}^{i \in 1..n}]$, $A' \equiv Obj(X)[l_i; B_i\{X\}^{i \in 1..n'}]$,
 $a \equiv obj(X=C)[l_i=\zeta(x_i; X)b_i\{X, x_i\}^{i \in 1..n+m}]$)

$$\frac{E \vdash a : A \quad E \vdash A <: A' \quad E, Y <: A, y:Y, x:Y \vdash b : B_j\{Y\} \quad j \in 1..n'}{E \vdash a.l_j \not=(Y <: A, y:Y)\zeta(x:Y)b : A}$$

(Eval Select'') (where $A \equiv Obj(X)[l_i; B_i\{X\}^{i \in 1..n}]$, $A' \equiv Obj(X)[l_i; B_i\{X\}^{i \in 1..n'}]$,
 $a \equiv obj(X=C)[l_i=\zeta(x_i; X)b_i\{X, x_i\}^{i \in 1..n+m}]$)

$$\frac{E \vdash a : A \quad E \vdash A <: A' \quad j \in 1..n}{E \vdash a.l_j \leftrightarrow b_j\{C, a\} : B_j\{A\}}$$

$$\begin{array}{c}
 (\text{Eval Update"}) \quad (\text{where } A \equiv \text{Obj}(X)[l_i; B_i\{X\}^{i \in 1..n}], \quad A' \equiv \text{Obj}(X)[l_i; B_i\{X\}^{i \in 1..n'}], \\
 \quad a \equiv \text{obj}(X=C)[l_i=\zeta(x_i; X) b_i\{X, x_i\}^{i \in 1..n+m}]) \\
 \frac{E \vdash a : A \quad E \vdash A <: A' \quad E, Y <: A, y:Y, x:Y \vdash b\{Y, y, x\} : B_j\{Y\} \quad j \in 1..n}{E \vdash a.l_j \in (Y <: A, y:Y) \zeta(x:Y) b\{Y, y, x\} \leftrightarrow \\
 \text{obj}(X=C)[l_j=\zeta(x_i; X) b_i\{X, a, x\}, l_i=\zeta(x_i; X) b_i\{X, x_i\}^{i \in (1..n+m)-\{j\}}] : A}
 \end{array}$$

These rules differ from the previous ones by a redundant assumption $E \vdash A <: A'$ and by the restriction of l_j to the labels of A' .

The rules we have derived so far are valid, but are not fully general: the typing rule (Val Select") for $a.l_j$, for example, is applicable only to an a of a specific form and not to a variable x such that $E \vdash x : A$. Similarly, that rule is applicable only to an A of a specific form and not to a type variable X such that $E \vdash X <: A'$. A stronger rule would be:

$$\begin{array}{c}
 (\text{where } A' \equiv \text{Obj}(X)[l_i; B_i\{X\}^{i \in 1..n'}]) \\
 \frac{E \vdash a : A \quad E \vdash A <: A' \quad j \in 1..n'}{E \vdash a.l_j : B_j\{A\}}
 \end{array}$$

Unfortunately, this stronger rule is not derivable. It is, however, reasonable because the structural invariants say that (at run-time) a and A can be only of the form given in (Val Select"). Moreover, as we show in Chapter 16, rules such as this one would solve the difficulties that we have found in constructing pre-methods for classes and in updating methods from the outside.

We cannot proceed further in our exploration of structural rules within $\mathbf{Ob}_{<\mu}$. In Chapter 16, we develop an alternative type system. There we remove the side conditions about the form of a subtype of an object type, and the side conditions about the shape of an object (except for evaluation rules). The removal of the side conditions has the following effects: (1) where a structural object was previously required, we allow a variable of a certain object type to occur; (2) where an object type with a certain shape and bound was previously required, we allow a type variable with the same bound to occur.

16 PRIMITIVE COVARIANT SELF TYPES

In Chapter 15 we have investigated an encoding of Self obtained via standard constructions. We have derived typing rules and equational theories for objects with Self that, as shown in Chapter 14, are sound in a standard denotational model. We have also devised the recoup technique to remedy a limitation of the derived typing rules, and discovered some desirable typing rules that are not derivable.

Building on that experience, we now axiomatize Self directly, taking Self as primitive. The new rules are similar to the ones in Section 15.7, and allow us to update methods with result type Self. These rules are not syntactically derivable, and are not sound for standard interpretations of subtyping as inclusion. Still, they are more natural (perhaps even required) from the point of view of programming. In absence of models, we establish their consistency by means of an operational semantics and a subject reduction theorem. We do not consider equational theories beyond those implicit in reduction systems.

As an illustration of the use of Self, we revisit some of our standard examples. Extrapolating from the examples, we exploit Self and bounded universal quantifiers in a new representation for classes and inheritance.

16.1 Primitive Self Types and Structural Rules

The most remarkable new aspect of the type system presented in this chapter is that it is based on structural assumptions about the universe of types. These assumptions are operationally valid, but would not hold in natural semantic models.

Rules based on structural assumptions (structural rules, for short) are not peculiar to object types. For example, the following rule for pairing enables us to mix two pairs a and b of type C into a new pair of the same type. The only assumption on C is that it is a subtype of a product type $B_1 \times B_2$.

$$\frac{E \vdash C <: B_1 \times B_2 \quad E \vdash a : C \quad E \vdash b : C}{E \vdash (fst(a), snd(b)) : C}$$

The soundness of this rule depends on the property that every subtype of a product type $B_1 \times B_2$ is itself a product type $C_1 \times C_2$. This property is true operationally for particular systems, but fails in any semantic model where subtyping is interpreted as the subset relation. Such a model would allow the set $\{a, b\}$ as a subtype of $B_1 \times B_2$ whenever a and b are elements of $B_1 \times B_2$. Then $(fst(a), snd(b))$ is not necessarily an element of $\{a, b\}$.

Structural rules do not seem to be very useful for standard type constructions such as products, but they are important in the treatment of object types with Self; they are required for typing programs satisfactorily and simply. Structural rules allow methods to act parametrically over any $X <: A$, where X is the Self type and A is a given object type, perhaps producing results of type X . We demonstrate the power of structural rules by examples, and by our representation of classes and inheritance. We prove that structural rules are sound for our operational semantics.

We adopt structural rules only for object types. We saw an example of such a structural rule in Section 13.1.3:

$$\frac{(\text{Val Structural Update}) \quad (\text{where } A \equiv [l_i : B_i]_{i \in 1..n})}{\begin{array}{c} E \vdash a : C \quad E \vdash C <: A \quad E, x:C \vdash b : B_j \quad j \in 1..n \\ \hline E \vdash a.l_j = \varsigma(x:C)b : C \end{array}}$$

This rule implies that, if $y : Y$ and $Y <: A$ where A is a given object type with an invariant component $l : B$, then we may update l in y with a term $b : B$ yielding an updated object of type Y , and not just A . This rule is based on the assumption that any $Y <: A$ is closed under updating of l with elements of B . The closure property holds in a model only if any subtype of A allows the result of l to be any element of B . Intuitively, this condition may fail because a subtype of A may be a subset with $l : B'$ for B' strictly included in B . Operationally, the closure property holds because any possible instance of Y in the course of a computation is a closed object type that, being a subtype of A , has a component l of type exactly B . The rule for update that we adopt in this chapter is analogous to (Val Structural Update), but deals in addition with Self.

16.2 Objects with Structural Rules

In this section, we define a minimal object calculus with Self and variance annotations; we name it **S**. Later we add quantification to **S** and we prove a subject reduction theorem for the extended system.

16.2.1 Syntax of Types and Variance

In order to obtain a flexible type system, we need constructions that provide both covariance and contravariance. For example, both variances are necessary to define function types. There are several possible choices at this point. One choice would be to take invariant object types plus the two bounded second-order quantifiers, and to proceed as in Chapter 15. Instead, we prefer to embed variance annotations directly into object types. This choice is technically sensible because it increases expressiveness, delays the need to use quantifiers, and yet does not significantly complicate the operational semantics and the related proofs.

We consider object types with Self of the form:

$\text{Obj}(X)[l_i v_i : B_i \{X^+\}_{i \in 1..n}]$

where $B\{X^+\}$ indicates that X occurs only covariantly in B

The binder Obj binds a Self type named X that can occur covariantly in the result types B_i . The covariance requirement is necessary for the soundness of our rules. As in Section 8.7, each v_i (a variance annotation) is one of the symbols $-$, o , and $+$, for contravariance, invariance, and covariance, respectively. Invariant components are the familiar ones. Covariant components allow covariant subtyping, but prevent updating. Symmetrically, contravariant components allow contravariant subtyping, but prevent invocation. Invariant components can be regarded, by subtyping, as either covariant or contravariant.

Syntax of S types

$A, B ::=$	types
X	type variable
Top	the biggest type
$\text{Obj}(X)[l_i v_i : B_i \{X^+\}_{i \in 1..n}]$	object type (l_i distinct, $v_i \in \{-, o, +\}$)

Formally, $B\{X^+\}$ indicates that X occurs at most positively in B ; similarly, $B\{X^-\}$ indicates that X occurs at most negatively in B .

Variant occurrences

$Y\{X^+\}$	whether $X = Y$ or $X \neq Y$
$\text{Top}\{X^+\}$	always
$\text{Obj}(Y)[l_i v_i : B_i \{X^+\}_{i \in 1..n}]\{X^+\}$	if $X = Y$ or for all $i \in 1..n$: if $v_i \equiv +$, then $B_i\{X^+\}$ if $v_i \equiv -$, then $B_i\{X^-\}$ if $v_i \equiv o$, then $X \notin FV(B_i)$
$Y\{X^-\}$	if $X \neq Y$
$\text{Top}\{X^-\}$	always
$\text{Obj}(Y)[l_i v_i : B_i \{X^-\}_{i \in 1..n}]\{X^-\}$	if $X = Y$ or for all $i \in 1..n$: if $v_i \equiv +$, then $B_i\{X^-\}$ if $v_i \equiv -$, then $B_i\{X^+\}$ if $v_i \equiv o$, then $X \notin FV(B_i)$
$A\{X^o\}$	if neither $A\{X^+\}$ nor $A\{X^-\}$

With this definition of covariant occurrences, more than one Self may be active in a given context, as long as the covariance requirement is respected, as in the type $\text{Obj}(X)[l^0 : \text{Obj}(Y)[m^+ : X, n^o : Y]]$.

16.2.2 Syntax of Terms and Operational Semantics

We introduce a new syntax for objects and object operations, inspired by the considerations of Section 15.7 about structural invariants.

Syntax of S terms

$a, b ::=$	terms
x	variable
$obj(X=A)[l_i=\zeta(x_i; X)b_i]^{i \in 1..n}$	object (l_i distinct)
$a.l$	method invocation
$a.l \Leftarrow (Y <: A, y: Y)\zeta(x: Y)b$	method update

An object has the form $obj(X=A)[l_i=\zeta(x_i; X)b_i]^{i \in 1..n}$, where A is the chosen implementation of the Self type. Variance information for this object is given as part of the type A . All the variables x_i have type X (so the syntax is somewhat redundant).

Method update is written $a.l \Leftarrow (Y <: A, y: Y)\zeta(x: Y)b$, where Y denotes the unknown Self type, y denotes the *old self* (the object a), and x denotes self (the self at the time the updating method is invoked). We saw the parameter y arise from the encoding of Self types in Chapter 15; it has the same role here: it denotes the object a , but with type Y instead of A .

To understand the necessity of the parameter y , consider the case where the method body b has result type Y . This method body cannot soundly return an arbitrary object of type A , because the type A may be shorter than the true Self type of the object a . Since the object a itself has the true Self type, the method could soundly return it, but the typing does not work out because a has type A rather than Y . To allow the object a to be returned, it is bound to y (in the operational semantics of update) with type Y . This mechanism is critical in certain examples discussed in Section 16.5.

The operational semantics is given in terms of a reduction judgment, $\vdash v \rightsquigarrow v'$. The results of reductions are objects of the form $obj(X=A)[l_i=\zeta(x_i; X)b_i]^{i \in 1..n}$. The reductions are essentially untyped, but they propagate the type information contained in terms in such a way that a subject reduction theorem can hold.

Operational semantics

(Red Object) (where $v \equiv obj(X=A)[l_i=\zeta(x_i; X)b_i]^{i \in 1..n}$)

$$\vdash v \rightsquigarrow v$$

(Red Select) (where $v' \equiv obj(X=A)[l_i=\zeta(x_i; X)b_i]^{i \in 1..n}$)

$$\frac{\vdash a \rightsquigarrow v' \quad \vdash b_j[A, v'] \rightsquigarrow v \quad j \in 1..n}{\vdash a.l_j \rightsquigarrow v}$$

(Red Update) (where $v \equiv obj(X=A)[l_i=\zeta(x;X)b_i^{i \in 1..n}]$)

$$\vdash a \rightsquigarrow v \quad j \in 1..n$$

$$\vdash a.l_j=(Y<:A',y:Y)\zeta(x;Y)b\{Y,y\} \rightsquigarrow obj(X=A)[l_j=\zeta(x;X)b\{X,v\}, l_i=\zeta(x_i;X)b_i^{i \in 1..n-\{j\}}]$$

These rules involve substitutions for the self parameters: (Red Select) replaces self and (Red Update) replaces old self. They involve also type substitutions to account for the presence of type variables in objects, replacing the formal Self with the actual Self.

16.2.3 Judgments and Rules for Types

The judgments used in the typing rules are familiar ones:

Judgments

$E \vdash \diamond$	well-formed environment judgment
$E \vdash A$	type judgment
$E \vdash A <: B$	subtyping judgment
$E \vdash vA <: v'B$	subtyping judgment with variance
$E \vdash a : A$	value typing judgment

The rules for the judgments $E \vdash \diamond$, $E \vdash A$, and $E \vdash A <: B$ are standard, except of course for the new rules for object types.

Environments, types, and subtypes

(Env \emptyset)	(Env x)	(Env $X <:$)
$\emptyset \vdash \diamond$	$E \vdash A \quad x \notin \text{dom}(E)$	$E \vdash A \quad X \notin \text{dom}(E)$
<hr/>		
$E, X <: A, E'' \vdash \diamond$	$E, x:A \vdash \diamond$	$E, X <: A \vdash \diamond$
<hr/>		
(Type $X <:$)	(Type Top)	(Type Object) (l_i distinct, $v_i \in \{^o, ^-, ^+\}$)
$E', X <: A, E'' \vdash \diamond$	$E \vdash \diamond$	$E, X <: \text{Top} \vdash B_i\{X^+\} \quad \forall i \in 1..n$
$E', X <: A, E'' \vdash X$	$E \vdash \text{Top}$	$E \vdash Obj(X)[l_i v_i; B_i\{X\}^{i \in 1..n}]$
<hr/>		
(Sub Refl)	(Sub Trans)	(Sub Top)
$E \vdash A$	$E \vdash A <: B \quad E \vdash B <: C$	$E \vdash A$
$E \vdash A <: A$	$E \vdash A <: C$	$E \vdash A <: \text{Top}$
<hr/>		
(Sub X)		
		$E', X <: A, E'' \vdash \diamond$
		$E', X <: A, E'' \vdash X <: A$
<hr/>		
(Sub Object) (where $A \equiv Obj(X)[l_i v_i; B_i\{X\}^{i \in 1..n+m}]$, $A' \equiv Obj(X)[l_i v'_i; B'_i\{X\}^{i \in 1..n}]$)		
$E \vdash A \quad E \vdash A' \quad E, Y <: A \vdash v_i B_i\{Y\} <: v'_i B'_i\{Y\} \quad \forall i \in 1..n$		
	$E \vdash A <: A'$	

(Sub Invariant)	(Sub Covariant)	(Sub Contravariant)
$E \vdash B$	$E \vdash B <: B' \quad v \in \{^o, ^+\}$	$E \vdash B' <: B \quad v \in \{^o, ^-\}$
$E \vdash {}^o B <: {}^o B$	$E \vdash v B <: {}^+ B'$	$E \vdash v B <: {}^- B'$

The formation rule for object types (Type Object) requires that all the component types be covariant in Self. This condition, which has already been discussed in Chapter 15, will be essential in the subject reduction proof.

The subtyping rule for object types (Sub Object) says, to a first approximation, that a longer object type A on the left is a subtype of a shorter object type A' on the right. Because of the variance annotations, as in Section 8.7, we use an auxiliary judgment for inclusion of attributes with variance, and three auxiliary rules. According to the rules, we may match an invariant component of A against a component of arbitrary variance of A' ; in this sense, an invariant component of A can be regarded as covariant or as contravariant. A covariant component of A requires a covariant component of A' , and a contravariant component of A requires a contravariant component of A' . The components are considered under the assumption that Self is a subtype A .

The type $\text{Obj}(X)[\dots]$ can be viewed as an alternative to the recursive type $\mu(X)[\dots]$, but with differences in subtyping that are crucial for object-oriented applications. The subtyping rule for object types (Sub Object), with all components invariant, reads:

$$\frac{E, X <: \text{Top} \vdash B_i[X^+] \quad \forall i \in 1..n+m}{E \vdash \text{Obj}(X)[l_i; B_i[X]^{i \in 1..n+m}] <: \text{Obj}(X)[l_i; B_i[X]^{i \in 1..n}]} \quad \text{(Sub Object)}$$

An analogous property fails with μ instead of Obj .

16.2.4 Rules for Terms

The rules for term typing are suggested by the ones in Section 15.7.

Terms with typing annotations

(Val Subsumption)	(Val x)
$E \vdash a : A \quad E \vdash A <: B$	$E', x:A, E'' \vdash \diamond$

(Val Object) (where $A \equiv \text{Obj}(X)[l_i; v_i; B_i[X]^{i \in 1..n}]$)
$E, x_i:A \vdash b_i[\{A\}] : B_i[\{A\}] \quad \forall i \in 1..n$

$$E \vdash obj(X=A)[l_i = \zeta(x_i; X) b_i[X]^{i \in 1..n}] : A$$

(Val Select) (where $A' \equiv \text{Obj}(X)[l_i; v_j; B_i[X]^{j \in 1..n}]$)
$E \vdash a : A \quad E \vdash A <: A' \quad v_j \in \{{}^o, {}^+\} \quad j \in 1..n$

$$E \vdash a.l_j : B_j[\{A\}]$$

$$\begin{array}{c}
 (\text{Val Update}) \quad (\text{where } A' \equiv \text{Obj}(X)[l; v_i; B_i[X]^{i \in 1..n}]) \\
 E \vdash a : A \quad E \vdash A <: A' \quad E, Y <: A, y:Y, x:Y \vdash b : B_j[Y] \quad v_j \in \{^o, ^-\} \quad j \in 1..n \\
 \hline
 E \vdash a.l_j \not\equiv (Y <: A, y:Y) \zeta(x:Y) b : A
 \end{array}$$

The rule (Val Object) can be used for building an object of a type A from code for its methods. In that code, the variable X refers to the Self type; in checking the code, X is replaced with A , and self is assumed of type A . Thus the object is built with knowledge that Self is A . The methods can exploit that knowledge. For example, a method with result type Self may be defined to return a fixed object of type A , rather than self or a modification of self. Such a method is not parametric in Self.

The rule (Val Select) treats method invocation, replacing the Self type X with a known type A for the object a whose method is invoked. The type A might not be the true type of a . The result type is obtained by examining a supertype A' of A .

The rule (Val Update) requires that an updating method work with a partially unknown Self type Y , which is assumed to be a subtype of a type A of the object a being modified. Because of this requirement, the updating method must be parametric in Self: it must return self, the old self, or a modification of these. Again, the result type is obtained by examining a supertype A' of A .

The rules (Val Select) and (Val Update) rely on the structural assumption that every subtype of an object type is an object type. In order to understand them, it may be useful to compare them with the following more obvious alternatives:

$$\begin{array}{c}
 (\text{Val Non-Structural Select}) \quad (\text{where } A \equiv \text{Obj}(X)[l; v_i; B_i[X]^{i \in 1..n}]) \\
 E \vdash a : A \quad v_j \in \{^o, ^+\} \quad j \in 1..n \\
 \hline
 E \vdash a.l_j : B_j[A]
 \end{array}$$

$$\begin{array}{c}
 (\text{Val Non-Structural Update}) \quad (\text{where } A \equiv \text{Obj}(X)[l; v_i; B_i[X]^{i \in 1..n}]) \\
 E \vdash a : A \quad E, Y <: A, y:Y, x:Y \vdash b : B_j[Y] \quad v_j \in \{^o, ^-\} \quad j \in 1..n \\
 \hline
 E \vdash a.l_j \not\equiv (Y <: A, y:Y) \zeta(x:Y) b : A
 \end{array}$$

These rules are easy special cases of (Val Select) and (Val Update) for $A \equiv A'$. Those rules are more general in that they allow A to be a type variable. For example, if $E \vdash x : X$ and $E \vdash X <: \text{Obj}(Y)[l:Y]$, then (Val Select) yields $E \vdash x.l : X$, while (Val Non-Structural Select), together with subsumption, yields only $E \vdash x.l : \text{Obj}(Y)[l:Y]$. The difference is essential to satisfy the assumptions of either (Val Update) or (Val Non-Structural Update).

We now have a complete system. Examples of its use appear in Section 16.5.

16.3 Quantifiers

Our system with Self types, S , is sufficient for expressing many examples. However, this system still lacks facilities for type parameterization. Type parameterization has

useful interactions with objects; in particular, it supports method reuse and inheritance. To allow for parameterization, we add standard bounded universal quantifiers. We do so without perturbing the existing definitions. We call S_V the resulting system.

First we extend the syntax of types and terms:

Syntax of type parameterization

$A, B ::=$	types
...	(as before)
$\forall(X<:A)B$	bounded universal type
$a, b ::=$	terms
...	(as before)
$\lambda(X<:A)b$	type abstraction
$a(A)$	type application

We add two rules to the operational semantics. According to these rules, evaluation stops at type abstractions and is triggered again by type applications. We let a type abstraction $\lambda(X<:A)b$ be a result.

Operational semantics for type parameterization

(Red Fun2) (where $v \equiv \lambda(X<:A)b$)

$$\vdash v \rightsquigarrow v$$

(Red Appl2)

$$\frac{\vdash b \rightsquigarrow \lambda(X<:A)c(X) \quad \vdash c[A'] \rightsquigarrow v}{\vdash b(A') \rightsquigarrow v}$$

The type rules for bounded universal quantifiers are:

Quantifier rules

(Type All $<:$)

$$E, X<:A \vdash B$$

$$E \vdash \forall(X<:A)B$$

(Sub All)

$$E \vdash A' <: A \quad E, X<:A' \vdash B <: B'$$

$$E \vdash \forall(X<:A)B <: \forall(X<:A')B'$$

(Val Fun2 $<:$)

$$E, X<:A \vdash b : B$$

$$E \vdash \lambda(X<:A)b : \forall(X<:A)B$$

(Val Appl2 $<:$)

$$E \vdash b : \forall(X<:A)B[X] \quad E \vdash A' <: A$$

$$E \vdash b(A') : B[A']$$

We also extend the definition of positive and negative occurrences:

Variant occurrences for quantifiers

$(\forall(Y <: A)B)\{X^+\}$	if $X = Y$ or both $A\{X^-\}$ and $B\{X^+\}$
$(\forall(Y <: A)B)\{X^-\}$	if $X = Y$ or both $A\{X^+\}$ and $B\{X^-\}$

16.4 Subject Reduction

The consistency of the operational semantics of S_V with its typing rules is established by a subject reduction theorem. We start by stating a bound weakening lemma, a substitution lemma (both without proof), three simple lemmas about subtyping, and a lemma about variance. The main theorem then follows.

Lemma 16.4-1 (Bound weakening)

If $E', X <: A, E'' \vdash \mathfrak{S}$ and $E' \vdash A' <: A$, then $E', X <: A', E'' \vdash \mathfrak{S}$.

□

Lemma 16.4-2 (Substitution)

- (1) If $E', X <: A, E''\{X\} \vdash \mathfrak{S}\{X\}$ and $E' \vdash A' <: A$, then $E', E''\{A'\} \vdash \mathfrak{S}\{A'\}$.
- (2) If $E', x:A, E'' \vdash \mathfrak{S}\{x\}$ and $E' \vdash a : A$, then $E', E'' \vdash \mathfrak{S}\{a\}$.

□

The next three lemmas analyze the possible forms of a type C that is a supertype of a given type. If the given type is Top , then C is Top . If the given type is an object type C' , then either C is Top or C is itself an object type, and in the latter case the component types of C are subtypes or supertypes of the corresponding component types of C' . Similarly, if the given type is a bounded universal type, then either C is Top or C is itself a bounded universal type.

Lemma 16.4-3

If $E \vdash \text{Top} <: C$, then $C \equiv \text{Top}$.

Proof

By induction on the derivation of $E \vdash \text{Top} <: C$.

The (Sub Refl) and (Sub Top) cases are immediate. The (Sub X), (Sub Object), and (Sub All $<:$) cases are vacuous. For (Sub Trans) we have $E \vdash \text{Top} <: C'$ and $E \vdash C' <: C$. By induction hypothesis, $C' \equiv \text{Top}$. Then again by induction hypothesis, $C \equiv \text{Top}$.

□

Lemma 16.4-4

If $C' \equiv \text{Obj}(X)[l_i v_i : B_i\{X\}]^{i \in I}$ and $E \vdash C' <: C$, then either $C \equiv \text{Top}$, or $C \equiv \text{Obj}(X)[l_i v_i : B_i\{X\}]^{i \in J}$ with $J \subseteq I$, and, for $j \in J$ we have:

- (1) if $v_j \in \{\circ, +\}$, then $v_j \in \{\circ, +\}$ and $E, X <: C' \vdash B_j\{X\} <: B_j\{X\}$,
- (2) if $v_j \in \{\circ, -\}$, then $v_j \in \{\circ, -\}$ and $E, X <: C' \vdash B_j\{X\} <: B_j\{X\}$.

Proof

By induction on the derivation of $E \vdash C' <: C$:

In the (Sub Refl) case we have $C' \equiv C$ and from (Type Object) $E, X <: Top \vdash B_i\{X\}$. We can then use Lemma 16.4-1 to replace Top with C' , and apply (Sub Refl). The (Sub Object) case is also easy, using reflexivity in the case $v_j' \equiv \circ$. The (Sub Top) case is immediate. The (Sub X) and (Sub All) cases are vacuous.

For (Sub Trans) we have $E \vdash C' <: C''$ and $E \vdash C'' <: C$. By induction hypothesis (on $E \vdash C' <: C''$), either $C'' \equiv Top$ or C'' has the form $Obj(X)[l; v_i'': B_i''\{X\}^{i \in K}]$. In the former case, $C \equiv Top$ by Lemma 16.4-3, and the proof is complete. In the latter case we obtain, in addition: $K \subseteq I$ and, for $j \in K$, if $v_j'' \in \{\circ, +\}$, then $v_j' \in \{\circ, +\}$ and $E, X <: C' \vdash B_j'\{X\} <: B_j''\{X\}$, and if $v_j'' \in \{\circ, -\}$, then $v_j' \in \{\circ, -\}$ and $E, X <: C' \vdash B_j''\{X\} <: B_j'\{X\}$. By induction hypothesis (on $E \vdash C'' <: C$), either $C \equiv Top$ or C has the form $Obj(X)[l; v_i; B_i\{X\}^{i \in J}]$. In the former case, the proof is complete. In the latter case we obtain, in addition: $J \subseteq K$ and, for $j \in J$, if $v_j \in \{\circ, +\}$, then $v_j'' \in \{\circ, +\}$ and $E, X <: C'' \vdash B_j''\{X\} <: B_j\{X\}$, and if $v_j \in \{\circ, -\}$, then $v_j'' \in \{\circ, -\}$ and $E, X <: C'' \vdash B_j\{X\} <: B_j''\{X\}$. Since $E \vdash C' <: C''$, Lemma 16.4-1 yields that, for $j \in J$, if $v_j \in \{\circ, +\}$, then $v_j'' \in \{\circ, +\}$ and $E, X <: C' \vdash B_j''\{X\} <: B_j\{X\}$, and if $v_j \in \{\circ, -\}$, then $v_j'' \in \{\circ, -\}$ and $E, X <: C' \vdash B_j\{X\} <: B_j''\{X\}$. By transitivity, $J \subseteq I$ and, for $j \in J$, if $v_j \in \{\circ, +\}$, then $v_j' \in \{\circ, +\}$ and $E, X <: C' \vdash B_j''\{X\} <: B_j\{X\}$, and if $v_j \in \{\circ, -\}$, then $v_j' \in \{\circ, -\}$ and $E, X <: C' \vdash B_j\{X\} <: B_j''\{X\}$.

□

Lemma 16.4-5

If $C' \equiv \forall(X <: A')B'$ and $E \vdash C' <: C$, then either $C \equiv Top$, or $C \equiv \forall(X <: A)B$ with $E \vdash A <: A'$ and $E, X <: A \vdash B' <: B$.

Proof

By induction on the derivation of $E \vdash C' <: C$.

In the (Sub Refl) case we have $C' \equiv C$; the derivation must have come from (Type All $<:$) with $E, X <: A \vdash B$, and therefore we have $E \vdash A$ by an easy lemma. By (Sub Refl), we obtain $E \vdash A <: A$ and $E, X <: A \vdash B <: B$. The (Sub Top) and (Sub All $<:$) cases are immediate. The (Sub X) and (Sub Object) cases are vacuous. For (Sub Trans) we have $E \vdash C' <: C''$ and $E \vdash C'' <: C$. By induction hypothesis, either $C'' \equiv Top$, and then $C \equiv Top$ by Lemma 16.4-3; or $C'' \equiv \forall(X <: A'')B''$ with $E \vdash A'' <: A'$ and $E, X <: A'' \vdash B' <: B''$. In the latter case, again by induction hypothesis, we have that either $C \equiv Top$, or $C \equiv \forall(X <: A)B$ with $E \vdash A <: A''$ and $E, X <: A \vdash B'' <: B$. By (Sub Trans), $E \vdash A <: A'$. By Lemma 16.4-1, $E \vdash A <: A''$ implies $E, X <: A \vdash B' <: B''$. By (Sub Trans), $E, X <: A \vdash B' <: B$.

□

The final lemma establishes that the syntactic criterion for variance is a sufficient condition for variant behavior:

Lemma 16.4-6

Assume $B\{Y^+, X^-\}$.

If $E, X <: A \vdash B\{X, X\}$ and $E \vdash C_1 <: C_2$ and $E \vdash C_2 <: A$, then $E \vdash B\{C_1, C_2\} <: B\{C_2, C_1\}$.

□

The proof of this lemma is given in Appendix C.2.

We can now prove the subject reduction property.

Theorem 16.4-7 (Subject reduction)

If $\emptyset \vdash a : A$ and $\vdash a \rightsquigarrow v$, then $\emptyset \vdash v : A$.

Proof

By induction on the derivation of $\vdash a \rightsquigarrow v$.

Case (Red Object) (where $v \equiv obj(X=A)[l_i=\zeta(x_i; X)b_i\{X, x_i\}^{i \in 1..n}]$)

$$\overline{\vdash v \rightsquigarrow v}$$

Immediate, since $a \equiv v$.

Case (Red Select) (where $v' \equiv obj(X=C')[l_i=\zeta(x_i; X)b_i\{X, x_i\}^{i \in 1..n}]$)

$$\frac{\vdash c \rightsquigarrow v' \quad \vdash b_j\{C', v'\} \rightsquigarrow v \quad j \in 1..n}{\vdash c.l_j \rightsquigarrow v}$$

By hypothesis $\emptyset \vdash c.l_j : A$. This must have come from (1) an application of (Val Select) with assumptions $\emptyset \vdash c : C$ and $\emptyset \vdash C <: D$ where D has the form $Obj(X)[l_j; v_j; B_j\{X\}, \dots]$ and $v_j \in \{^0, ^+\}$, and with conclusion $\emptyset \vdash c.l_j : B_j\{C\}$, followed by (2) a number of subsumption steps implying $\emptyset \vdash B_j\{C\} <: A$ by transitivity.

By induction hypothesis, since $\emptyset \vdash c : C$ and $\vdash c \rightsquigarrow v'$, we have $\emptyset \vdash v' : C$.

Now $\emptyset \vdash v' : C$ must have come from (1) an application of (Val Object) with assumptions $\emptyset, x_i; C' \vdash b_j\{C'\} : B_j\{C'\}$ and $C' \equiv Obj(X)[l_i; v'_i; B_i\{X\}^{i \in 1..n}]$, and with conclusion $\emptyset \vdash v' : C'$, followed by (2) a number of subsumption steps implying $\emptyset \vdash C' <: C$ by transitivity. By transitivity, $\emptyset \vdash C' <: D$. Hence by Lemma 16.4-4 we must have $v'_i \in \{^0, ^+\}$ and $\emptyset, X <: C' \vdash B_j\{X\} <: B_j\{X\}$. By Lemma 16.4-2, $\emptyset \vdash B_j\{C'\} <: B_j\{C\}$. From $\emptyset, x_j; C' \vdash b_j\{C', x_j\} : B_j\{C'\}$ we obtain $\emptyset \vdash b_j\{C', v'\} : B_j\{C'\}$ by Lemma 16.4-2. By Lemma 16.4-6 and the covariance of $B_j\{X\}$, we have $\emptyset \vdash B_j\{C'\} <: B_j\{C\}$, and so $\emptyset \vdash B_j\{C'\} <: A$ by transitivity. By subsumption, $\emptyset \vdash b_j\{C', v'\} : A$.

By induction hypothesis, since $\emptyset \vdash b_j\{C', v'\} : A$ and $\vdash b_j\{C', v'\} \rightsquigarrow v$, we have $\emptyset \vdash v : A$.

Case (Red Update) (where $v \equiv obj(X=C')[l_i=\zeta(x_i;X)b_i^{i \in 1..n}]$)

$$\frac{\vdash c \rightsquigarrow v \quad j \in 1..n}{\vdash c.l_j=(Y<:C,y:Y)\zeta(x:Y)b\{Y,y\} \rightsquigarrow obj(X=C')[l_i=\zeta(x_i;X)b\{X,v\}, l_i=\zeta(x_i;X)b_i^{i \in 1..n-j}]}$$

By hypothesis $\emptyset \vdash c.l_j=(Y<:C,y:Y)\zeta(x:Y)b\{Y,y,x\} : A$. This must have come from (1) an application of (Val Update) with assumptions $\emptyset \vdash c : C$, and $\emptyset \vdash C <: D$ where D has the form $Obj(X)[l_jv_j:B_j\{X\}, \dots]$, and $\emptyset, Y<:C, y:Y, x:Y \vdash b\{Y,y,x\} : B_j\{Y\}$, and $v_j \in \{^0, ^-\}$, and with conclusion $\emptyset \vdash c.l_j=(Y<:C,y:Y)\zeta(x:Y)b\{Y,y,x\} : C$, followed by (2) a number of subsumption steps implying $\emptyset \vdash C <: A$ by transitivity.

By induction hypothesis, since $\emptyset \vdash c : C$ and $\vdash c \rightsquigarrow v$, we have $\emptyset \vdash v : C$.

Now $\emptyset \vdash v : C$ must have come from (1) an application of (Val Object) with assumptions $\emptyset, x_i:C' \vdash b_i\{C',x_i\} : B_i\{C'\}$ and $C' \equiv Obj(X)[l'_i v'_i : B_i\{X\} \text{ } i \in 1..n]$, and with conclusion $\emptyset \vdash v : C'$, followed by (2) a number of subsumption steps implying $\emptyset \vdash C' <: C$ by transitivity. By transitivity, $\emptyset \vdash C' <: D$. Hence by Lemma 16.4-4 we must have $v'_j \in \{^0, ^-\}$, and $\emptyset, X <: C' \vdash B_j\{X\} <: B_j\{X\}$, and by Lemma 16.4-2 $\emptyset \vdash B_j\{C'\} <: B_j\{C'\}$. From $\emptyset, Y <: C, y:Y, x:Y \vdash b\{Y,y,x\} : B_j\{Y\}$ by Lemma 16.4-2 we get $\emptyset, x:C' \vdash b\{C',v,x\} : B_j\{C'\}$, and by subsumption $\emptyset, x:C' \vdash b\{C',v,x\} : B_j\{C'\}$. Then, by (Val Object), we obtain $\emptyset \vdash obj(X=C')[l_i=\zeta(x_i;X)b\{X,v,x\}, l_i=\zeta(x_i;X)b_i^{i \in 1..n-j}] : C'$. Since $\emptyset \vdash C' <: A$ by transitivity, we have $\emptyset \vdash obj(X=C')[l_i=\zeta(x_i;X)b\{X,v,x\}, l_i=\zeta(x_i;X)b_i^{i \in 1..n-j}] : A$ by subsumption.

Case (Red Fun2) (where $v \equiv \lambda(X <: D)b$)

$$\frac{}{\vdash v \rightsquigarrow v}$$

Immediate, since $a \equiv v$.

Case (Red Appl2)

$$\frac{\vdash b \rightsquigarrow \lambda(X <: D')c\{X\} \quad \vdash c\{D''\} \rightsquigarrow v}{\vdash b(D'') \rightsquigarrow v}$$

By hypothesis $\emptyset \vdash b(D'') : A$. This must have come from (1) an application of (Val Appl2 $<:$) with assumptions $\emptyset \vdash b : \forall(X <: D)C\{X\}$ and $\emptyset \vdash D'' <: D$ and conclusion $\emptyset \vdash b(D'') : C\{D''\}$, followed by (2) a number of subsumption steps implying $\emptyset \vdash C\{D''\} <: A$ by transitivity.

By induction hypothesis, since $\emptyset \vdash b : \forall(X <: D)C\{X\}$ and $\vdash b \rightsquigarrow \lambda(X <: D')c\{X\}$, we have $\emptyset \vdash \lambda(X <: D')c\{X\} : \forall(X <: D)C\{X\}$.

Now, $\emptyset \vdash \lambda(X <: D')c\{X\} : \forall(X <: D)C\{X\}$ must have come from (1) an application of (Val Fun2 $<:$) with assumption $\emptyset, X <: D' \vdash c\{X\} : C'\{X\}$ and conclusion $\emptyset \vdash \lambda(X <: D')c\{X\} : \forall(X <: D')C'\{X\}$, followed by (2) a number of subsumption steps

implying $\emptyset \vdash \forall(X <: D')C\{X\} <: \forall(X <: D)C\{X\}$. Hence $\emptyset \vdash D <: D'$ and $\emptyset, X <: D \vdash C\{X\} <: C\{X\}$ by Lemma 16.4-5.

We have $\emptyset \vdash D'' <: D$, and also $\emptyset \vdash D'' <: D'$ by transitivity. Therefore $\emptyset \vdash C'\{D''\} <: C\{D'\}$, and $\emptyset \vdash c\{D''\} : C'\{D''\}$ by Lemma 16.4-2.

By induction hypothesis, since $\emptyset \vdash c\{D''\} : C'\{D''\}$ and $\vdash c\{D''\} \rightsquigarrow v$, we have $\emptyset \vdash v : C'\{D''\}$. By transitivity and subsumption, $\emptyset \vdash v : A$.

□

16.5 Examples

We now revisit some examples in order to demonstrate the power of the rules with primitive Self. In other chapters we have seen how to program these examples, sometimes with fairly elaborate techniques. In contrast, the code in this section is almost as straightforward as could be expected. The examples show that much can be done without quantifiers, but that quantifiers are needed to factor out methods as generic procedures. Later we describe how collections of generic procedures are organized into classes and subclasses.

For simplicity we use only invariant components. It requires little extrapolation to see how variance annotations could be used in examples.

In the examples to follow we frequently use the notations of Chapters 7 and 8 as abbreviations:

$$\begin{array}{lll} [l_i \cup; B_i]^{i \in 1..n} & \triangleq & Obj(X)[l_i \cup; B_i]^{i \in 1..n} & \text{for } X \notin FV(B_i), i \in 1..n \\ [l_i; B_i]^{i \in 1..n} & \triangleq & Obj(X)[l_i^0; B_i]^{i \in 1..n} & \text{for } X \notin FV(B_i), i \in 1..n \\ [l_i = \zeta(x_i; A) b_i]^{i \in 1..n} & \triangleq & obj(X=A)[l_i = \zeta(x_i; X) b_i]^{i \in 1..n} & \text{for } X \notin FV(b_i), i \in 1..n \\ a.l_j = \zeta(x; A)b & \triangleq & a.l_j \Leftrightarrow (Y <: A, y: Y) \zeta(x: Y) b & \text{for } Y, y \notin FV(b) \end{array}$$

We also assume basic types and function types.

16.5.1 Storage Cells

Our first example illustrates some critical uses of the structural assumptions in (Val Update). We define the type of storage cells with a *get* field and with a *set* method that, given a number, returns a storage cell storing that number.

$$\begin{aligned} St &\triangleq Obj(X)[get:Nat, set:Nat \rightarrow X] \\ st : St &\triangleq [get = 0, set = \zeta(x:St) \lambda(n:Nat) x.get := n] \\ &\equiv [get = \zeta(x:St) 0, \\ &\quad set = \zeta(x:St) \lambda(n:Nat) x.get \Leftarrow (Y <: St, y: Y) \zeta(z: Y) n] \end{aligned}$$

The claimed typing for the storage cell *st* can be derived as follows, assuming introduction rules for *Nat* and \rightarrow . For compactness, all type annotations are omitted in terms appearing in derivations.

$\emptyset, x:St \vdash 0 : Nat$	<i>Nat introduction</i>
$\emptyset, x:St, n:Nat \vdash x : St$	by (Val x)
$\emptyset, x:St, n:Nat \vdash St <: St$	by (Sub Refl)
$\emptyset, x:St, n:Nat, Y <: St, y:Y, z:Y \vdash n : Nat$	by (Val x)
$\emptyset, x:St, n:Nat \vdash x.get\#(y)\zeta(z)n : St$	(Val Update)
$\emptyset, x:St \vdash \lambda(n)x.get\#(y)\zeta(z)n : Nat \rightarrow St$	\rightarrow introduction
$\emptyset \vdash [get\#(x)0, set\#(x)\lambda(n)x.get\#(y)\zeta(z)n] : St$	(Val Object)

Note, in particular, the application of the (Val Update) rule, whose assumptions are easy to satisfy.

More interesting uses of (Val Update) are required when updating methods that return a value of type Self. We update the method *set* with a parametric method that always sets *get* to 0. We prove:

$$st.set\#(Y <: St, y:Y)\zeta(x:Y)\lambda(n:Nat)x.get:=0 : St$$

as follows:

$\emptyset \vdash st : St$	<i>by previous derivation</i>
$\emptyset \vdash St <: St$	by (Sub Refl)
$\emptyset, Y <: St, y:Y, x:Y, n:Nat \vdash x : Y$	by (Val x)
$\emptyset, Y <: St, y:Y, x:Y, n:Nat \vdash Y <: St$	by (Sub X)
$\emptyset, Y <: St, y:Y, x:Y, n:Nat, Z <: Y, z:Z, w:Z \vdash 0 : Nat$	<i>Nat introduction</i>
$\emptyset, Y <: St, y:Y, x:Y, n:Nat \vdash x.get\#(z)\zeta(w)0 : Y$	(Val Update)
$\emptyset, Y <: St, y:Y, x:Y \vdash \lambda(n)x.get\#(z)\zeta(w)0 : Nat \rightarrow Y$	\rightarrow introduction
$\emptyset \vdash st.set\#(y)\zeta(x)\lambda(n)x.get\#(z)\zeta(w)0 : St$	(Val Update)

The parametricity of the updating method is established by the first application of (Val Update), which gives type Y to $x.get:=0$ for an arbitrary $Y <: St$.

The treatment of other kinds of cells is completely analogous. For example, we can redefine the types *Cell* and *ReCell* of Section 15.2.6:

$$Cell \triangleq$$

$$Obj(Self)[contents:Nat, get:Nat, set:Nat \rightarrow Self]$$

$$ReCell \triangleq$$

$$Obj(Self)[contents:Nat, get:Nat, set:Nat \rightarrow Self, backup:Nat, restore:Self]$$

We obtain the subtyping $ReCell <: Cell <: St$.

In the presence of bounded quantifiers, we can define “parametric pre-methods” as polymorphic functions that can later be used in updating. For example, the method used above for updating *st.set* can be isolated as a function of type $\forall(X <: St)X \rightarrow X$. We derive:

$$\lambda(X <: St)\lambda(st:X)st.set\#(Y <: X, y:Y)\zeta(x:Y)\lambda(n:Nat)x.get:=0 : \forall(X <: St)X \rightarrow X$$

as follows:

$\emptyset, X <: St, st:X \vdash st : X$	by (Val x)
$\emptyset, X <: St, st:X \vdash X <: St$	by (Sub X)
$\emptyset, X <: St, st:X, Y <: X, y:Y, x:Y, n:Nat \vdash x : Y$	by (Val x)
$\emptyset, X <: St, st:X, Y <: X, y:Y, x:Y, n:Nat \vdash Y <: St$	by (Sub X , Sub Trans)
$\emptyset, X <: St, st:X, Y <: X, y:Y, x:Y, n:Nat, Z <: Y, z:Z, w:Z \vdash 0 : Nat$	Nat introduction
$\emptyset, X <: St, st:X, Y <: X, y:Y, x:Y, n:Nat \vdash x.get\Leftarrow(z)\zeta(w)0 : Y$	(Val Update)
$\emptyset, X <: St, st:X, Y <: X, y:Y, x:Y \vdash \lambda(n)x.get\Leftarrow(z)\zeta(w)0 : Nat \rightarrow Y$	\rightarrow introduction
$\emptyset, X <: St, st:X \vdash st.set\Leftarrow(y)\zeta(x)\lambda(n)x.get\Leftarrow(z)\zeta(w)0 : X$	(Val Update)
$\emptyset, X <: St \vdash \lambda(st)st.set\Leftarrow(y)\zeta(x)\lambda(n)x.get\Leftarrow(z)\zeta(w)0 : X \rightarrow X$	\rightarrow introduction
$\emptyset \vdash \lambda(X <: St)\lambda(st)st.set\Leftarrow(y)\zeta(x)\lambda(n)x.get\Leftarrow(z)\zeta(w)0 : \forall(X <: St)X \rightarrow X$	(Val Fun2 $<$)

Note, in particular, the essential use of the structural subtyping assumption in the occurrences of (Val Update) in this derivation; without the structural subtyping assumption, we would obtain at best the type $\forall(X <: St)X \rightarrow St$ instead of the type $\forall(X <: St)X \rightarrow X$.

Polymorphic functions such as the one above are the foundation on which classes can be built, as shown in Section 16.6.

16.5.2 Backup Methods

An interesting difficulty arises when trying to update fields of type *Self*; this difficulty is avoided by using the old-self parameter of the update operator. Consider again objects with backup:

$$\begin{aligned} Bk &\triangleq Obj(Self)[retrieve:Self, backup:Self, \dots] \\ o : Bk &\triangleq \\ &\quad obj(Self=Bk) \\ &\quad [retrieve = \zeta(s:Self)s, \\ &\quad backup = \zeta(s:Self)s.retrieve \Leftarrow (Y <: Self, y:Y) \zeta(x:Y)y, \\ &\quad \dots] \end{aligned}$$

Note the way the *backup* method is written. We might have expected a simpler coding of this method, namely:

$$backup = \zeta(s:Self)s.retrieve := s,$$

These two definitions of *backup* are in fact operationally equivalent, because during reduction the parameter y is replaced by the old self, s . The simpler definition of *backup*, however, does not typecheck, as can be understood by verifying the typing of the correct version.

The crucial point in the derivation of $o : Bk$ is that the rule (Val Update) requires the type Y for the body of the *backup* method. The parameter y has this type, but s does not. The derivation is as follows:

$$\begin{array}{l}
 \text{... (retrieve, and other attributes)} \\
 \left\{ \begin{array}{ll}
 \emptyset, s:Bk \vdash s : Bk & \text{by (Val } x\text{)} \\
 \emptyset, s:Bk \vdash Bk <: Bk & \text{by (Sub Refl)} \\
 \downarrow \\
 \emptyset, s:Bk, Y <: Bk, y:Y, x:Y \vdash y : Y & \text{by (Val } x\text{)} \\
 \emptyset, s:Bk \vdash s.retrieve \Leftarrow (Y <: Self, y:Y) \zeta(x:Y) y : Bk & \text{(Val Update)} \\
 \emptyset, s:Bk \vdash obj(Self=Bk)[\text{backup} = \zeta(s:Self)] s.retrieve \Leftarrow (Y <: Self, y:Y) \zeta(x:Y) y, \dots : Bk & \\
 & \text{(Val Object)}
 \end{array} \right.
 \end{array}$$

16.5.3 Object-Oriented Natural Numbers

We adapt the example of the object-oriented natural numbers to our new object types. Compare this formulation with the one in Section 15.2.4, and note the absence of coercions. The structural subtyping assumptions in our new rules are responsible for the code simplification. Note, however, that the code of the *succ* method needs to use the old-self parameter:

$$\begin{aligned}
 N_{Ob} &\triangleq Obj(Self)[succ:Self, case:\forall(Z) Z \rightarrow (Self \rightarrow Z) \rightarrow Z] \\
 zero_{Ob} : N_{Ob} &\triangleq \\
 &obj(Self=N_{Ob}) \\
 &[case = \lambda(Z) \lambda(z:Z) \lambda(f:Self \rightarrow Z) z, \\
 &\quad succ = \zeta(n:Self) \\
 &\quad n.case \Leftarrow (Y <: Self, y:Y) \zeta(x:Y) \lambda(Z) \lambda(z:Z) \lambda(f:Y \rightarrow Z) f(y)]
 \end{aligned}$$

As shown in Section 16.5.1, we can now update even methods that return values of type *Self*, given a sufficiently parametric new method. For example, we can typecheck:

$$\begin{aligned}
 &zero_{Ob}.succ \\
 &\Leftarrow (Self <: N_{Ob}, n':Self) \zeta(n:Self) \\
 &n.case \Leftarrow (Y <: Self, y:Y) \zeta(x:Y) \lambda(Z) \lambda(z:Z) \lambda(f:Self \rightarrow Z) f(y) \\
 &: N_{Ob}
 \end{aligned}$$

The crucial intermediate step is to obtain the typing $n.case \Leftarrow \dots : Self$.

16.5.4 Movable Points

We define types of movable points P_1 and P_2 , with the subtyping $P_2 <: P_1$. A polymorphic function *inc_x* is used for building origin points *origin*₁ and *origin*₂ of P_1 and P_2 . We can view *origin*₂ as inheriting the definition of *inc_x* originally intended for *origin*₁. The code of *mv_x* is identical in the two origin points although *Self* refers to different types.

$$\begin{aligned}
 P_1 &\triangleq Obj(Self)[x:Int, mv_x:Int \rightarrow Self] \\
 inc_x : \forall(Self <: P_1) Self \rightarrow Int \rightarrow Self &\triangleq \\
 &\lambda(Self <: P_1) \lambda(s:Self) \lambda(dx:Int) s.x := s.x + dx \\
 origin_1 &\triangleq obj(Self=P_1)[x = 0, mv_x = \zeta(s:Self) inc_x(Self)(s)]
 \end{aligned}$$

$$\begin{aligned} P_2 &\triangleq \text{Obj}(\text{Self})[x,y:\text{Int}, \text{mv_x}, \text{mv_y}:\text{Int} \rightarrow \text{Self}] \\ \text{origin}_2 &\triangleq \text{obj}(\text{Self} = P_2)[x = 0, \text{mv_x} = \zeta(s:\text{Self}) \text{inc_x}(\text{Self})(s), y = 0, \text{mv_y} = \dots] \end{aligned}$$

Moreover, we can define other polymorphic functions of type $\forall(\text{Self} \subset P_1) \text{ Self} \rightarrow \text{Int} \rightarrow \text{Self}$, and use them for updating mv_x in any object of a subtype of P_1 .

$$\begin{aligned} \text{inc2_x} : \forall(\text{Self} \subset P_1) \text{ Self} \rightarrow \text{Int} \rightarrow \text{Self} &\triangleq \\ \lambda(\text{Self} \subset P_1) \lambda(s:\text{Self}) \lambda(dx:\text{Int}) s.x &:= s.x + 2^* dx \\ \text{origin}_1.\text{mv_x} &\Leftarrow (\text{Self} \subset P_1, s' : \text{Self}) \zeta(s:\text{Self}) \text{inc2_x}(\text{Self})(s) \\ \text{origin}_2.\text{mv_x} &\Leftarrow (\text{Self} \subset P_2, s' : \text{Self}) \zeta(s:\text{Self}) \text{inc2_x}(\text{Self})(s) \end{aligned}$$

16.6 Classes and Inheritance, with Primitive Self

In this section we develop a presentation of classes based on types of the form $\forall(X \subset A)X \rightarrow B\{X\}$. We obtain a satisfactory conclusion to the encoding of classes for object types with **Self**, first attempted in Section 15.4.

16.6.1 Classes

Several types of the form $\forall(X \subset A)X \rightarrow B\{X\}$ were used in the previous examples for updating methods that return values of type **Self**. These types can be used also for defining classes as collections of pre-methods. Each pre-method must work for all possible subclasses, parametrically in **Self**, so that it can be inherited and instantiated to any of these subclasses. This is precisely what a type of the form $\forall(X \subset A)X \rightarrow B\{X\}$ expresses. In the special case where $B\{X\}$ is simply X , the type $\forall(X \subset A)X \rightarrow X$ captures exactly that the pre-method must return its self argument, possibly with some updates. Therefore the type $\forall(X \subset A)X \rightarrow X$ is a formal version of the condition of being parametric in **Self** (first discussed in Section 9.6), and the type $\forall(X \subset A)X \rightarrow B\{X\}$ is a generalization of that condition.

As in previous chapters, we can associate a class type $\text{Class}(A)$ with each object type A . We make the components of $\text{Class}(A)$ invariant, for simplicity:

$$\begin{aligned} A &\equiv \text{Obj}(X)[l_i \nu_i; B_i\{X\}^{i \in 1..n}] \\ \text{Class}(A) &\triangleq [\text{new}:A, l_i: \forall(X \subset A)X \rightarrow B_i\{X\}^{i \in 1..n}] \\ c : \text{Class}(A) &\triangleq [\text{new} = \zeta(z:\text{Class}(A))\text{obj}(X=A)[l_i = \zeta(s:X)z.l_i(X)(s)^{i \in 1..n}], \dots] \end{aligned}$$

For example, we may rewrite the class of one-dimensional points from Section 15.6.3:

$$\begin{aligned} P_1 &\triangleq \text{Obj}(\text{Self})[x:\text{Int}, \text{mv_x}:\text{Int} \rightarrow \text{Self}] \\ \text{Class}(P_1) &\triangleq \\ &[\text{new}:P_1, \\ &x: \forall(\text{Self} \subset P_1) \text{ Self} \rightarrow \text{Int}, \\ &\text{mv_x}: \forall(\text{Self} \subset P_1) \text{ Self} \rightarrow \text{Int} \rightarrow \text{Self}] \end{aligned}$$

$$\begin{aligned}
p_1\text{-class} : Class(P_1) &\triangleq \\
&[new = \zeta(z:Class(P_1)) obj(Self=P_1)] \\
&[x = \zeta(s:Self) z.x(Self)(s),] \\
&\quad mv_x = \zeta(s:Self) z.mv_x(Self)(s)], \\
&x = \lambda(Self <: P_1) \lambda(s:Self) 0, \\
&mv_x = \lambda(Self <: P_1) \lambda(s:Self) \lambda(dx:Int) s.x := s.x + dx]
\end{aligned}$$

We can now consider the inheritance relation between classes. Suppose that we have another type $A' <: A$, and a corresponding class type $Class(A')$:

$$\begin{aligned}
A' &\equiv Obj(X)[l_i \cup_i' B_i'[X] \ i \in 1..n+m] \\
Class(A') &\equiv [new:A', l_i: \forall(X <: A') X \rightarrow B_i'[X] \ i \in 1..n+m]
\end{aligned}$$

We say that:

$$\begin{aligned}
l_i \text{ is } \text{inheritable} \text{ from } Class(A) \text{ into } Class(A') \\
\text{if and only if } X <: A' \text{ implies } B_i[X] <: B_i'[X], \text{ for all } i \in 1..n
\end{aligned}$$

When l_i is inheritable, we have:

$$\forall(X <: A) X \rightarrow B_i[X] <: \forall(X <: A') X \rightarrow B_i'[X]$$

So, if $c : Class(A)$ and l_i is inheritable, we obtain $c.l_i : \forall(X <: A') X \rightarrow B_i'[X]$. Then $c.l_i$ can be reused when building a class $c' : Class(A')$. That is, it can be inherited.

For example, mv_x is inheritable from $Class(P_1)$ to $Class(P_2)$:

$$\begin{aligned}
P_2 &\triangleq Obj(Self)[x,y:Int, mv_x, mv_y:Int \rightarrow Self] \\
Class(P_2) &\triangleq \\
&[new:P_2] \\
&x,y:\forall(Self <: P_2) Self \rightarrow Int, \\
&\quad mv_x, mv_y:\forall(Self <: P_2) Self \rightarrow Int \rightarrow Self] \\
p_2\text{-class} : Class(P_2) &\triangleq \\
&[new = \zeta(z:Class(P_2)) obj(Self=P_2)[\dots],] \\
&x = \lambda(Self <: P_2) \lambda(s:Self) 0, \\
&y = \lambda(Self <: P_2) \lambda(s:Self) 0, \\
&\quad mv_x = p_1\text{-class}.mv_x, \\
&\quad mv_y = \lambda(Self <: P_2) \lambda(s:Self) \lambda(dy:Int) s.y := s.y + dy]
\end{aligned}$$

The inheritability condition holds for any invariant component l_i , since in this case $B_i[X] = B_i'[X]$.

For contravariant components, if $A' \equiv Obj(X)[l^-:B'[X], \dots]$ and $A \equiv Obj(X)[l^-:B[X], \dots]$ with $A' <: A$, then by Lemma 16.4-4 we have that $X <: A'$ implies $B[X] <: B'[X]$. Now, if $c : Class(A)$, then $c.l$ can always be inherited into $Class(A')$ because $\forall(X <: A) X \rightarrow B[X] <: \forall(X <: A') X \rightarrow B'[X]$. Thus inheritability of contravariant components is guaranteed.

Covariant components do not necessarily correspond to inheritable pre-methods. For example, if $A' \equiv [l^+:Nat]$ and $A \equiv [l^+:Int]$, and $c : [new:A, l:\forall(X <: A) X \rightarrow Int]$, then $c.l$

cannot be inherited into $\text{Class}(A') \equiv [\text{new}:A', l:\forall(X <: A')X \rightarrow \text{Nat}]$, because it would produce a bad result. However, a class $c' : \text{Class}(A')$ can include a different method for l with result type Nat ; this corresponds to method specialization on override, discussed in the Review.

In summary, covariant components induce mild restrictions in subclassing. Invariant and contravariant components induce no restrictions. In practice, inheritability is expected between a class type C and another class type C' obtained as an extension of C (so that C' and C have identical common components). In this case, inheritability trivially holds for components of any variance.

16.6.2 Variations on Class Types

The variations on class types explored in Section 8.5.2 apply to the more general class types studied here. For an object type $A \equiv \text{Obj}(X)[l_i; v_i; B_i\{X\}^{i \in I}]$ with methods $l_i^{i \in I}$ we consider a restricted *instance interface*, determined by a set $\text{Ins} \subseteq I$, and a restricted *subclass interface*, determined by a set $\text{Sub} \subseteq I$. We define, for $\text{Ins}, \text{Sub} \subseteq I$:

$$\text{Class}(A)_{\text{Ins}, \text{Sub}} \triangleq [\text{new}: \text{Obj}(X)[l_i; v_i; B_i\{X\}^{i \in \text{Ins}}], l_i; \forall(X <: A)X \rightarrow B_i\{X\}^{i \in \text{Sub}}]$$

16.6.3 Accessing Classes from Objects

We can take advantage of Self types to increase the expressiveness of our classes.

We let an instance a of a class c generate new objects of its own class. To obtain an object a with this capability, we include a method named *fresh* in a . When $a.\text{fresh}$ is invoked, it produces a fresh instance of c by executing $c.\text{new}$. Therefore the object a should have the form:

$$a \equiv \text{obj}(X=A)[\text{fresh}=\zeta(s:X)c.\text{new}, l_i=\zeta(s:X)b_i^{i \in 1..n}]$$

Each method body b_i may contain invocations of $s.\text{fresh}$. Correspondingly, the type of a should have the form:

$$A \equiv \text{Obj}(X)[\text{fresh}: X, l_i; v_i; B_i\{X\}^{i \in 1..n}]$$

The result type of *fresh* is the Self type X , because the new object that *fresh* generates has the same type as a .

In order to generate objects of type A with the expected behavior, we need to extend our representation of classes. We give a special treatment to *fresh*: in defining classes for A , we do not include a pre-method for *fresh*, but instead insert the method *fresh* directly into the body of *new*. We obtain the class type:

$$C \triangleq [\text{new}:A, l_i; \forall(X <: A)X \rightarrow B_i\{X\}^{i \in 1..n}]$$

A typical class of this type has the form:

$$\begin{aligned} c : C &\triangleq \\ &[\text{new}=\zeta(z:C)\text{obj}(X=A)[\text{fresh}=\zeta(s:X)z.\text{new}, l_i=\zeta(s:X)z.l_i(X)(s)^{i \in 1..n}], \\ &l_i=\lambda(X <: A)\lambda(s:X)b_i^{i \in 1..n}] \end{aligned}$$

The code for *fresh* uses the self of the class, and is well-typed because in the context of the definition of the *new* method it is known that the Self type of A is A itself. The pre-methods of c take as input a type $X <: A$ and a self variable of type X , and hence they are allowed to call *fresh*.

If we have a type A' with more methods than A , then we obtain $A' <: A$ by virtue of the subtyping properties of Self types. Hence pre-methods can be inherited in subclasses, by the usual mechanisms. A pre-method that contains an invocation of $s.fresh$ generates in each subclass an object of that subclass.

It is instructive to consider implementing *fresh* with a regular pre-method, and to see why this idea does not work. The obvious code for the *fresh* pre-method is:

$$\text{fresh} = \zeta(z:\text{Class}(A))\lambda(X <: A)\lambda(s:X)z.\text{new}$$

Given the type $A \equiv \text{Obj}(X)[\text{fresh}:X, l_i\forall_i:B_i[X]^{i \in 1..n}]$, our standard definition of $\text{Class}(A)$ says that the pre-method should be of type $\forall(X <: A)X \rightarrow X$. The code for *fresh*, on the other hand, yields only the type $\forall(X <: A)X \rightarrow A$. This difference explains why we have resorted to a special treatment and extended our representation of classes.

17 IMPERATIVE CALCULI WITH SELF TYPES

In this chapter we study an imperative object calculus with primitive covariant Self types. It is an extension of the first-order calculus of Chapter 11, enriched with the Self types, the variance annotations, and the quantifiers of Chapter 16. A subject reduction result covers subtyping and polymorphism in the presence of side-effects.

This chapter combines the ideas of Chapters 11 and 16 in a mostly straightforward manner, but we do encounter some interesting new issues. The proof of subject reduction is harder than that of Chapter 11, essentially because of Self types and polymorphism. In addition, even the choice of syntax is somewhat delicate: we find that the interaction of Self types and imperative semantics calls for a generalized form of method update.

17.1 Syntax of Terms and Operational Semantics

We adopt untyped terms, as in our treatment of imperative calculi in Chapters 10 and 11. The terms are almost the same as the ones described in Section 10.1, but we adopt a more complex method update construct that subsumes the *let* construct. This change is due to the eager evaluation requirements of the imperative semantics, combined with typing requirements for Self types.

17.1.1 Syntax of Terms

The syntax of the $\text{imp}\zeta$ -calculus is revised as follows:

Syntax of the $\text{imp}\zeta$ -calculus

$a, b ::=$	terms
x	variable
$[l_i = \zeta(x_i)b_i \mid i \in 1..n]$	object (l_i distinct)
$a.l \Leftarrow (y, z = c)\zeta(x)b$	method invocation
$\text{clone}(a)$	method update
	cloning

The construct for method update is an (untyped) extension of that of Chapter 16. In the term $a.l \Leftarrow (y, z = c\{y\})\zeta(x)b\{x, y, z\}$, the variable y still stands for the old self (that is, a), and may occur free in c and b ; the variable x stands for self. In addition, method update

evaluates the term c and binds the result to the variable z , which may occur free in b . Thus in the untyped imperative calculus of Chapter 10, we may paraphrase:

$$a.l\Leftarrow(y,z=c)\zeta(x)b \quad \text{as} \quad let\ y=a\ in\ let\ z=c\ in\ y.l\Leftarrow\zeta(x)b$$

These terms have the same operational behavior, but the new update construct has a more powerful typing rule than that which can be obtained from this encoding.

The usefulness of this form of method update will not become clear until an example in Section 17.4. Until then, one may think of the new update construct as a combination of the old update and *let* constructs.

The new syntax does not include *let*; as shown later, *let* is definable in terms of the extended form of method update.

17.1.2 Reduction Rules

The operational semantics is a variant of that of Chapter 10. It is given in terms of a reduction relation between two stores (σ , σ'), a stack (S), a term (b), and a result (v).

$$\sigma.S \vdash b \rightsquigarrow v.\sigma'$$

The only changes in the operational semantics are the new rule for method update and the absence of *let*. The new rule is:

Operational semantics of method update

(Red Update)

$$\begin{aligned} \sigma.S \vdash a \rightsquigarrow [l_i=l_i^{i \in 1..n}].\sigma' \quad l_i \in \text{dom}(\sigma) \quad j \in 1..n \\ \sigma'.(S, y \mapsto [l_i=l_i^{i \in 1..n}]) \vdash c \rightsquigarrow v.\sigma'' \end{aligned}$$

$$\sigma.S \vdash a.l\Leftarrow(y,z=c)\zeta(x)b \rightsquigarrow [l_i=l_i^{i \in 1..n}].(\sigma''.l_j\Leftarrow(\zeta(x)b, (S, y \mapsto [l_i=l_i^{i \in 1..n}], z \mapsto v)))$$

An update operation $a.l\Leftarrow(y,z=c)\zeta(x)b$ first reduces the object a to the final result $[l_i=l_i^{i \in 1..n}]$. Next, with y bound to $[l_i=l_i^{i \in 1..n}]$, it reduces the term c to a result v . Finally, it updates the appropriate store location with a closure. The closure consists of the new method $\zeta(x)b$ and a stack binding y and z .

17.2 Typing with Self

The types in this chapter are the same as the ones of system S_V of Chapter 16. In this section, we use those of the subsystem S , which include type variables, a *Top* type, and object types of the form $Obj(X)[l_i\nu_i; B_i\{X\}^{i \in 1..n}]$; the relevant rules are given in Section 16.2.3.

The type rules for terms borrow from both those of the first-order imperative calculus and those of the functional calculus with primitive covariant Self types of Chapters 11 and 16, respectively.

Terms

$\begin{array}{c} (\text{Val Subsumption}) \\ \dfrac{E \vdash a : A \quad E \vdash A <: B}{E \vdash a : B} \end{array}$	$\begin{array}{c} (\text{Val } x) \\ \dfrac{E', x:A, E'' \vdash \diamond}{E', x:A, E'' \vdash x : A} \end{array}$
$\begin{array}{c} (\text{Val Object}) \quad (\text{where } A \equiv Obj(X)[l_i v_i; B_i[X]^{i \in 1..n}]) \\ \dfrac{E, x_i:A \vdash b_i : B_i[A] \quad \forall i \in 1..n}{E \vdash [l_i = \zeta(x_i) b_i]^{i \in 1..n} : A} \end{array}$	
$\begin{array}{c} (\text{Val Select}) \quad (\text{where } A' \equiv Obj(X)[l_i v_i; B_i[X]^{i \in 1..n}]) \\ \dfrac{E \vdash a : A \quad E \vdash A <: A' \quad v_j \in \{^o, ^+ \} \quad j \in 1..n}{E \vdash a.l_j : B_j[A]} \end{array}$	
$\begin{array}{c} (\text{Val Update}) \quad (\text{where } A' \equiv Obj(X)[l_i v_i; B_i[X]^{i \in 1..n}]) \\ \dfrac{E \vdash a : A \quad E \vdash A <: A' \quad E, Y <: A, y:Y \vdash c : C \\ E, Y <: A, y:Y, z:C, x:Y \vdash b : B_j[Y] \quad v_j \in \{^o, ^- \} \quad j \in 1..n}{E \vdash a.l_j \Leftarrow (y, z=c) \zeta(x) b : A} \end{array}$	
$\begin{array}{c} (\text{Val Clone}) \quad (\text{where } A' \equiv Obj(X)[l_i v_i; B_i[X]^{i \in 1..n}]) \\ \dfrac{E \vdash a : A \quad E \vdash A <: A'}{E \vdash \text{clone}(a) : A} \end{array}$	

The new rule for update, (Val Update), requires the term c to be typed in an environment where y has type Y and where Y is a subtype of the type of the object being updated. The type C obtained for c is used as the type of z in typing the new method body b .

The rules (Val Select) and (Val Update) are structural, as in Chapter 16, and so is the rule (Val Clone).

17.3 Quantifiers

We now extend the $\text{imp}\zeta$ -calculus with quantifiers. We add two new forms to the syntax of terms: we write $\lambda()b$ for a type abstraction and $a()$ for a type application. In typed calculi, it is common to find $\lambda(X <: A)b$ and $a(A)$ instead. However, we are already committed to an untyped operational semantics, so we strip the types from those terms. Technically, we adopt $\lambda()b$, instead of dropping $\lambda()$ altogether, in order to distinguish the elements of quantified types from those of object types. The distinction greatly simplifies case analysis in proofs.

Syntax of the $\text{imp}\zeta\forall$ -calculus

$a, b ::=$	terms
...	(as for $\text{imp}\zeta$)
$\lambda()b$	type abstraction
$a()$	type application

This choice of syntax is reflected in the operational semantics. We let a closure $(\lambda()b, S)$ be a result and we add two rules to the operational semantics. According to these rules, evaluation stops at type abstractions and is triggered again by type applications. This is a sensible semantics of polymorphism, particularly in the presence of side-effects.

Operational semantics for polymorphism

$v ::=$	results
...	(as for $\text{imp}\zeta$)
$(\lambda()b, S)$	type abstraction result

$$\begin{array}{l} (\text{Red Fun2}) \\ \sigma \cdot S \vdash \diamond \\ \sigma \cdot S \vdash \lambda()b \rightsquigarrow (\lambda()b, S) \cdot \sigma \end{array}$$

$$\begin{array}{l} (\text{Red Appl2}) \\ \sigma \cdot S \vdash a \rightsquigarrow (\lambda()b, S') \cdot \sigma' \quad \sigma' \cdot S' \vdash b \rightsquigarrow v \cdot \sigma'' \\ \sigma \cdot S \vdash a() \rightsquigarrow v \cdot \sigma'' \end{array}$$

The type rules for polymorphism are the same as in functional calculi, except for the change in term syntax.

Quantifier rules

$$\begin{array}{ll} (\text{Type All} <:) & (\text{Sub All}) \\ E, X <: A \vdash B & E \vdash A' <: A \quad E, X <: A' \vdash B <: B' \\ \hline E \vdash \forall(X <: A)B & E \vdash \forall(X <: A)B <: \forall(X <: A')B' \end{array}$$

$$\begin{array}{ll} (\text{Val Fun2} <:) & (\text{Val Appl2} <:) \\ E, X <: A \vdash b : B & E \vdash b : \forall(X <: A)B[X] \quad E \vdash A' <: A \\ \hline E \vdash \lambda()b : \forall(X <: A)B & E \vdash b() : B[A'] \end{array}$$

17.4 Examples

Many of our standard examples carry over to this calculus without surprises. Here we discuss some peculiarities of imperative semantics and Self types.

17.4.1 Encoding *let*

We begin by showing how the constructs of Chapters 10 and 11 can be represented in the current calculus. The new method update construct can express both plain method update and *let*:

$$\begin{aligned} [l_i; B_i \ i \in 1..n] &\triangleq Obj(X)[l_i^o; B_i \ i \in 1..n] \\ a.l \Leftarrow \zeta(x)b &\triangleq a.l \Leftarrow (y, z = [])\zeta(x)b \\ let \ x = a \ in \ b\{x\} &\triangleq ([val = \zeta(y)y.val].val \Leftarrow (z, x = a)\zeta(w)b\{x\}).val \\ &\text{where } y, z, w \notin FV(b) \text{ and } z \notin FV(a), \text{ with } x, y, z, w \text{ distinct} \\ &\text{and } X \notin FV(B_i), \text{ for } i \in 1..n \end{aligned}$$

With these definitions, the first-order typing rules of Section 11.1 are derivable from the typing rules in Sections 16.2.3 and 17.2. Moreover, the imperative operational semantics of Section 10.5 can be emulated by the operational semantics of Section 17.1.2.

Since *let* is definable, we can obtain fields as in Section 10.2. However, a more direct encoding of field update is now possible:

$$\begin{aligned} [l_i = b_i \ i \in 1..n, l_j = \zeta(x_j)b_j \ j \in n+1..n+m] &\triangleq \\ let \ y_1 = b_1 \ in \ ... \ let \ y_n = b_n \ in [l_i = \zeta(y_0)y_i \ i \in 1..n, l_j = \zeta(x_j)b_j \ j \in n+1..n+m] \\ a.l := b &\triangleq a.l \Leftarrow (y, z = b)\zeta(x)z \\ &\text{where } y_i \notin FV(b_k \ k \in 1..n+m), y_i \text{ distinct for } i \in 0..n \\ &\text{and } x, y, z \notin FV(b) \text{ with } x, y, z \text{ distinct} \end{aligned}$$

17.4.2 Alternative Forms of Method Update

In Part I we transformed a first-order functional calculus into a first-order imperative calculus simply by adding the *let* and *clone* constructs, and modifying the operational semantics. One might hope to proceed in the same way for calculi with Self types, to define an imperative version of the calculus of Chapter 16. However, that is not what we have done in this chapter: we have generalized the method update construct to $a.l \Leftarrow (y, z = c)\zeta(x)b$, and have not included a *let* construct. As we have just seen, the *let* construct is definable; as we explain next, the new update construct is crucial in some examples.

Suppose we have available only a more limited form of method update (analogous to the one in Chapter 16), defined as:

$$a.l \Leftarrow (y)\zeta(x)b \triangleq a.l \Leftarrow (y, z = [])\zeta(x)b$$

with the derived rule:

$$\begin{array}{c}
 (\text{Val Update}') \quad (\text{where } A' \equiv \text{Obj}(X)[l_i v_j; B_i\{X\}^{i \in 1..n}]) \\
 E \vdash a : A \quad E \vdash A <: A' \\
 E, Y <: A, y:Y, x:Y \vdash b : B_j\{Y\} \quad v_j \in \{\circ, \neg\} \quad j \in 1..n \\
 \hline
 E \vdash a.l_i \not= (y) \zeta(x)b : A
 \end{array}$$

We now show by an example that, using just this construct and *let*, we cannot type certain terms that we would expect to type.

Consider the type of objects with backup:

$$Bk \triangleq \text{Obj}(\text{Self})[\text{retrieve}:\text{Self}, \text{backup}:\text{Self}, \dots]$$

The functional code for the *backup* method, from Section 6.5.2, is:

$$\text{backup} = \zeta(\text{self}) \text{ self.retrieve} \not= \zeta(w) \text{ self}$$

This code does not work imperatively: the *backup* method should make a copy of *self*, so that the *backup* is not affected by side-effects on *self*. Instead we could write one of:

$$\begin{aligned}
 \text{backup} &= \zeta(\text{self}) \text{ self.retrieve} \not= (y) \zeta(w) \text{ clone}(\text{self}) \\
 \text{backup} &= \zeta(\text{self}) \text{ self.retrieve} \not= (y) \zeta(w) \text{ clone}(y)
 \end{aligned}$$

Neither achieves the desired effect because cloning is delayed by the $\zeta(w)$ binder: cloning happens when the *retrieve* method is invoked, not when the *backup* method is invoked. We should therefore pull the cloning outside the binder:

$$\text{backup} = \zeta(\text{self}) \text{ let } z = \text{clone}(\text{self}) \text{ in } \text{self.retrieve} \not= (y) \zeta(w) z$$

This code has the expected imperative semantics, but does not typecheck according to the rule (Val Update'). In typing $\text{self.retrieve} \not= \zeta(w)z$, we have $\text{self} : Bk$, and $z : Bk$ as well. The update requires that for an arbitrary $Y <: Bk$ we be able to show that $z : Y$. This cannot be achieved.

Using the more general update construct, we can write the correct code for *backup* without relying on *let*:

$$\text{backup} = \zeta(\text{self}) \text{ self.retrieve} \not= (y, z=\text{clone}(y)) \zeta(x) z$$

Moreover, this code typechecks according to the rule (Val Update) of Section 17.2. In typing $\text{self.retrieve} \not= (y, z=\text{clone}(y)) \zeta(x) z$, we have $\text{self} : Bk$. For an arbitrary $Y <: Bk$, the update rule assigns to y the type Y . Therefore we have also $\text{clone}(y)$ of type Y by (Val Clone) and hence z has the required type Y .

Note that these typing issues arise even with field update, and are not a consequence of allowing method update. They arise from the combination of Self and eager evaluation when a component of a type that depends on Self is updated.

17.4.3 Protected Storage Cells

As a final example, we continue the discussion of storage cells (see Section 16.5.1) in an imperative setting. We have:

$$\begin{aligned} St &\triangleq Obj(X)[get:Nat, set:Nat \rightarrow X] \\ st : St &\triangleq [get = 0, set = \zeta(x) \lambda(n) x.get := n] \end{aligned}$$

It is natural to expect both components of a storage cell to be protected against external update. The *get* field should be protected because it is used only for reading (since writing should be through *set*), and the *set* method should not normally be updated. For protection, we can use covariance annotations:

$$ProtectedSt \triangleq Obj(X)[get^+:Nat, set^+:Nat \rightarrow X]$$

We obtain $St <: ProtectedSt$.

Thus any storage cell can be subsumed into *ProtectedSt* and protected against updating. However, the *set* method is typed with the type *St* for self, so it can still update the *get* field.

17.5 Subject Reduction

We adapt the subject reduction proof of Section 11.4 to the calculus with Self. The resulting proof is lengthy and laborious, and many readers may want to skip it. We include this proof in part to show how our methods apply to a fairly complex calculus, and as an indicator of how subject reduction techniques may scale up to realistic programming languages.

Two main changes are needed with respect to the proof of Section 11.4. The first is a change in the definition of method type, since the Self variable occurs in the method result types. The second is the introduction of type variables in environments, with the corresponding need for type stacks in the stack typing judgment.

17.5.1 typings with Stores

A method type has now the form:

$$M \equiv Obj(X)[l_i v_i : B_i\{X\}^{i \in 1..n}] \Rightarrow j$$

The type of the self argument for a method of method type *M* is $Obj(X)[l_i v_i : B_i\{X\}^{i \in 1..n}]$, and its result type is $B_j\{Obj(X)[l_i v_i : B_i\{X\}^{i \in 1..n}]\}$. The index *j* on the right of \Rightarrow determines the choice of $B_j\{X\}$.

A type stack is an association of type variables to types, of the form:

$$T \equiv X_i \mapsto A_i^{i \in 1..n} \quad A_i \text{ closed types}$$

We introduce an operation of substitution of a type stack *T*:

Substitution of type stacks

$$\begin{aligned} \diamond \{ \leftarrow T \} &\triangleq \diamond \\ A \{ \leftarrow T \} &\triangleq A\{X_i \leftarrow A_i^{i \in 1..n}\} \quad \text{for } T \equiv X_i \mapsto A_i^{i \in 1..n} \\ (A' <: A)\{ \leftarrow T \} &\triangleq (A'\{ \leftarrow T \}) <: (A\{ \leftarrow T \}) \end{aligned}$$

$$\begin{aligned} (\vee A' <: \vee A')\{\leftarrow T\} &\triangleq \vee(A'\{\leftarrow T\}) <: \vee(A\{\leftarrow T\}) \\ (a : A)\{\leftarrow T\} &\triangleq a : (A\{\leftarrow T\}) \end{aligned}$$

$E\{\leftarrow T\}$ is defined by:

$$\begin{aligned} \emptyset\{\leftarrow T\} &\triangleq \emptyset \\ (E, x:A)\{\leftarrow T\} &\triangleq E\{\leftarrow T\}, x:A\{\leftarrow T\} \\ (E, X <: A)\{\leftarrow T\} &\triangleq E\{\leftarrow T\} && \text{if } X \in \text{dom}(T) \\ (E, X <: A)\{\leftarrow T\} &\triangleq E\{\leftarrow T\}, X <: A\{\leftarrow T\} && \text{if } X \notin \text{dom}(T) \end{aligned}$$

Given these definitions, the judgments involving store typing are a fairly straightforward generalization of the ones in Section 11.4. To preserve the untyped character of the operational semantics, type stacks are not included in closures along with stacks; in particular, (Store Typing) deals with an empty type stack. Then, substitutions $A\{\leftarrow T\}$ must be applied appropriately, as shown in (Stack x Typing) and (Stack X Typing).

Store typing

$M ::= Obj(X)[l_i; v_i; B_i]^{i \in 1..n} \Rightarrow j$	method type ($j \in 1..n$)
$\Sigma ::= \iota_i \rightarrow M_i^{i \in 1..n}$	store type (ι_i distinct)
$\Sigma_1(\iota) \triangleq Obj(X)[l_i; v_i; B_i[X]^{i \in 1..n}]$	if $\Sigma(\iota) = Obj(X)[l_i; v_i; B_i[X]^{i \in 1..n}] \Rightarrow j$
$\Sigma_2(A, \iota) \triangleq B_j[A]$	if $\Sigma(\iota) = Obj(X)[l_i; v_i; B_i[X]^{i \in 1..n}] \Rightarrow j$
$\models M \in Meth$	well-formed method type judgment (M closed)
$\Sigma \models \diamond$	well-formed store type judgment
$\Sigma \models v : A$	result typing judgment (A closed)
$\Sigma \models S \cdot T : E$	stack typing judgment ($\text{dom}(S) \cup \text{dom}(T) = \text{dom}(E)$)
$\Sigma \models \sigma$	store typing judgment

(Method Type) (ι_i distinct, $v_i \in \{^o, ^-, ^+\}$)

$$\emptyset, X <: Top \vdash B_i[X^+] \quad j \in 1..n$$

$$\models Obj(X)[l_i; v_i; B_i[X]^{i \in 1..n}] \Rightarrow j \in Meth$$

(Store Type) (ι_i distinct)

$$\models M_i \in Meth \quad \forall i \in 1..n$$

$$\iota_i \rightarrow M_i^{i \in 1..n} \models \diamond$$

(Result Object)

$$\Sigma \models \diamond \quad \Sigma_1(\iota_i) \equiv Obj(X)[l_i; v_i; \Sigma_2(X, \iota_i)_{i \in 1..n}] \quad \forall i \in 1..n$$

$$\Sigma \models [l_i = \iota_i]^{i \in 1..n} : Obj(X)[l_i; v_i; \Sigma_2(X, \iota_i)_{i \in 1..n}]$$

(Result Fun2<:)

$$\frac{\Sigma \models S \cdot \emptyset : E \quad E, X <: A \vdash b : B\{X\}}{\Sigma \models (\lambda(b)S) : \forall(X <: A)B\{X\}}$$

(Stack \emptyset Typing)

$$\frac{\Sigma \models \diamond}{\Sigma \models \emptyset \cdot \emptyset : \emptyset}$$

(Stack x Typing)

$$\frac{\Sigma \models S \cdot T : E \quad \Sigma \models v : A\{\leftarrow T\} \quad x \notin \text{dom}(E)}{\Sigma \models (S, x \mapsto v) \cdot T : (E, x:A)}$$

(Stack X Typing)

$$\frac{\Sigma \models S \cdot T : E \quad \emptyset \vdash B <: A\{\leftarrow T\} \quad X \notin \text{dom}(E)}{\Sigma \models S \cdot (T, X \mapsto B) : (E, X <: A)}$$

(Store Typing)

$$\frac{\Sigma \models S_i \cdot \emptyset : E_i \quad E_i, x_i : \Sigma_1(\iota_i) \vdash b_i : \Sigma_2(\Sigma_1(\iota_i), \iota_i) \quad \forall i \in 1..n}{\Sigma \models \iota_i \mapsto (\zeta(x_i) b_i, S_i)_{i \in 1..n}}$$

17.5.2 Proof of Subject Reduction

We state a few lemmas, without proof.

Lemma 17.5-1

- (1) If $E', X <: A, E''\{X\} \vdash \mathfrak{S}\{X\}$ and $E' \vdash A' <: A$, then $E', E''\{A'\} \vdash \mathfrak{S}\{A'\}$.
- (2) If $E', x:A, E'' \vdash \mathfrak{S}$, then $E', E'' \vdash \mathfrak{S}$, for \mathfrak{S} not of the form $a : C$.
- (3) If $E', X <: A, E'' \vdash \mathfrak{S}$ and $E \vdash A' <: A$, then $E', X <: A', E'' \vdash \mathfrak{S}$.
- (4) If $E', x:A, E'' \vdash \mathfrak{S}$ and $E \vdash A' <: A$, then $E', x:A', E'' \vdash \mathfrak{S}$.

□

Lemma 17.5-2

Assume that $\Sigma \models S \cdot T : E$, then:

- (1) $X \in \text{dom}(E)$ if and only if $X \in \text{dom}(T)$.
- (2) $\Sigma \models S \cdot \emptyset : E\{\leftarrow T\}$.
- (3) If $E, E' \vdash \mathfrak{S}$, then $(E, E')\{\leftarrow T\} \vdash \mathfrak{S}\{\leftarrow T\}$.

□

By Lemma 17.5-2(3), if $\Sigma \models S.T : E$ and $E, E' \vdash \mathfrak{S}$, then $(E, E')\{\leftarrow T\} \vdash \mathfrak{S}\{\leftarrow T\}$. By Lemma 17.5-2(1), $\Sigma \models S.T : E$ implies that $E\{\leftarrow T\}$ has no type variables; therefore if $\mathfrak{S}\{\leftarrow T\}$ is not of the form $a : C$, then $\emptyset, E'\{\leftarrow T\} \vdash \mathfrak{S}\{\leftarrow T\}$ by Lemma 17.5-1(2). The proof of the subject reduction theorem uses Lemmas 17.5-2(3) and 17.5-1(2) in this fashion several times.

The next lemma is the analogue of Lemma 11.4-1; as in Chapter 11, $\Sigma' \succcurlyeq \Sigma$ means that Σ' is an extension of Σ .

Lemma 17.5-3

If $\Sigma \models S.T : E$ and $\Sigma' \models \diamond$ with $\Sigma' \succcurlyeq \Sigma$, then $\Sigma' \models S.T : E$.

If $\Sigma \models a : A$ and $\Sigma' \models \diamond$ with $\Sigma' \succcurlyeq \Sigma$, then $\Sigma' \models a : A$.

□

The subject reduction theorem is:

Theorem 17.5-4

If $E \vdash a : A \wedge \sigma.S \vdash a \rightsquigarrow v.\sigma^\dagger \wedge \Sigma \models \sigma \wedge \text{dom}(\sigma) = \text{dom}(\Sigma) \wedge \Sigma \models S.T : E$, then there exist a closed type A^\dagger and a store type Σ^\dagger such that $\Sigma^\dagger \succcurlyeq \Sigma \wedge \Sigma^\dagger \models \sigma^\dagger \wedge \text{dom}(\sigma^\dagger) = \text{dom}(\Sigma^\dagger) \wedge \Sigma^\dagger \models v : A^\dagger \wedge \emptyset \vdash A^\dagger \lessdot A\{\leftarrow T\}$.

Proof

The proof is by induction on the derivation of $\sigma.S \vdash a \rightsquigarrow v.\sigma^\dagger$. Despite its complexity, it contains essentially no surprises.

Case (Red x)

$$\frac{\sigma.(S', x \mapsto [l_i = l_i^{i \in 1..n}], S'') \vdash \diamond}{\sigma.(S', x \mapsto [l_i = l_i^{i \in 1..n}], S'') \vdash x \rightsquigarrow [l_i = l_i^{i \in 1..n}] \cdot \sigma}$$

By hypothesis, $E \vdash x : A \wedge \Sigma \models \sigma \wedge \text{dom}(\sigma) = \text{dom}(\Sigma) \wedge \Sigma \models S.T : E$ where $S \equiv S', x \mapsto [l_i = l_i^{i \in 1..n}], S''$. Since $E \vdash x : A$, we must have $E \equiv E', x:A', E''$ for some A' such that $E' \vdash A'$ and $E \vdash A' \lessdot A$. Then, by Lemmas 17.5-2(3) and 17.5-1(2), $\emptyset \vdash A'\{\leftarrow T\} \lessdot A\{\leftarrow T\}$. Now $\Sigma \models S.T : E$ must have been derived from $\Sigma \models (S', x \mapsto [l_i = l_i^{i \in 1..n}]) \cdot T' : E', x:A'$, for some prefix T' of T , and therefore $\Sigma \models [l_i = l_i^{i \in 1..n}] : A'\{\leftarrow T\}$ by (Stack x Typing). Take $A^\dagger \equiv A'\{\leftarrow T\}$. We have $A'\{\leftarrow T\} \equiv A'\{\leftarrow T\}$, because the free variables of A' are included in $\text{dom}(E')$, and hence in $\text{dom}(T')$ by Lemma 17.5-2(1), and hence $A^\dagger \equiv A'\{\leftarrow T\}$. Take $\Sigma^\dagger \equiv \Sigma$.

We conclude $\Sigma^\dagger \models \sigma \wedge \text{dom}(\sigma) = \text{dom}(\Sigma^\dagger) \wedge \Sigma^\dagger \models [l_i = l_i^{i \in 1..n}] : A^\dagger \wedge \emptyset \vdash A^\dagger \lessdot A\{\leftarrow T\}$.

Case (Red Object) (l_i, \mathfrak{l}_i distinct)

$$\frac{\sigma.S \vdash \diamond \quad \mathfrak{l}_i \notin \text{dom}(\sigma) \quad \forall i \in 1..n}{\sigma.S \vdash [l_i = \zeta(x_i)b_i^{i \in 1..n}] \rightsquigarrow [l_i = l_i^{i \in 1..n}] \cdot (\sigma, \mathfrak{l}_i \mapsto (\zeta(x_i)b_i, S)^{i \in 1..n})}$$

By hypothesis, $E \vdash [l_i = \zeta(x_i)b_i^{i \in 1..n}] : A \wedge \Sigma \models \sigma \wedge \text{dom}(\sigma) = \text{dom}(\Sigma) \wedge \Sigma \models S.T : E$. Since $E \vdash [l_i = \zeta(x_i)b_i^{i \in 1..n}] : A$, we must have $E \vdash [l_i = \zeta(x_i)b_i^{i \in 1..n}] : \text{Obj}(X)[l_i \mathfrak{v}_i; B_i[X]^{i \in 1..n}]$ by

(Val Object), for some type $Obj(X)[l_i v_i; B_i\{X\}^{i \in 1..n}]$ such that $E \vdash Obj(X)[l_i v_i; B_i\{X\}^{i \in 1..n}] <: A$. Therefore $E, x_i : Obj(X)[l_i v_i; B_i\{X\}^{i \in 1..n}] \vdash b_i : B_i\{Obj(X)[l_i v_i; B_i\{X\}^{i \in 1..n}]\}$ for $i \in 1..n$.

Since we have $E \vdash Obj(X)[l_i v_i; B_i\{X\}^{i \in 1..n}] <: A$, we must also have $E \vdash Obj(X)[l_i v_i; B_i\{X\}^{i \in 1..n}]$, and therefore $E, X <: Top \vdash B_i\{X^+\}$ for $i \in 1..n$ and $X \notin dom(E)$. Since $X \notin dom(E)$, we have $X \notin dom(T)$ by Lemma 17.5-2(1). Hence $\emptyset, X <: Top \vdash B_i\{X\}\{\leftarrow T\}$ by Lemmas 17.5-2(3) and 17.5-1(2). Take $A^+ \equiv Obj(X)[l_i v_i; B_i\{X\}^{i \in 1..n}]\{\leftarrow T\}$. Therefore, since $\emptyset, X <: Top \vdash B_i\{X\}\{\leftarrow T\}$, (Method Type) yields $\vdash A^+ \Rightarrow j \in Meth$ for $j \in 1..n$. Take $\Sigma^+ \equiv \Sigma, \downarrow \mapsto (A^+ \Rightarrow j)^{j \in 1..n}$; by (Store Type) we have $\Sigma^+ \models \diamond$, because $\downarrow \notin dom(\sigma)$, and hence $\downarrow \notin dom(\Sigma)$, and because $\vdash A^+ \Rightarrow j \in Meth$ for $j \in 1..n$.

(1) Since Σ^+ is an extension of Σ , we also have $\Sigma^+ \models S.T : E$ by Lemma 17.5-3, so $\Sigma^+ \models S.\emptyset : E\{\leftarrow T\}$ by Lemma 17.5-2(2). From $E, x_i : Obj(X)[l_i v_i; B_i\{X\}^{i \in 1..n}] \vdash b_i : B_i\{Obj(X)[l_i v_i; B_i\{X\}^{i \in 1..n}]\}$ we have $E\{\leftarrow T\}, x_i : A^+ \vdash b_i : B_i\{\leftarrow T\}\{A^+\}$ by Lemma 17.5-2(3), that is, $E\{\leftarrow T\}, x_i : \Sigma^+_1(\downarrow_i) \vdash b_i : \Sigma^+_2(\Sigma^+_1(\downarrow_i), \downarrow_i)$.

(2) We have that σ is of the form $\epsilon_k \mapsto (\zeta(x_k)b_k, S_k)^{k \in 1..m}$. Now $\Sigma \models \sigma$ must come from the (Store Typing) rule, with $\Sigma \models S_k.\emptyset : E_k$ and $E_k, x_k : \Sigma_1(\epsilon_k) \vdash b_k : \Sigma_2(\Sigma_1(\epsilon_k), \epsilon_k)$. By Lemma 17.5-3, $\Sigma^+ \models S_k.\emptyset : E_k$; moreover $E_k, x_k : \Sigma^+_1(\epsilon_k) \vdash b_k : \Sigma^+_2(\Sigma^+_1(\epsilon_k), \epsilon_k)$, because $\Sigma^+(\epsilon_k) = \Sigma(\epsilon_k)$ for $k \in 1..m$ since $dom(\sigma) = dom(\Sigma) = \{\epsilon_k^{k \in 1..m}\}$ and Σ^+ extends Σ .

Take $\sigma^+ \equiv (\sigma, \downarrow \mapsto (\zeta(x_i)b_i, S)^{i \in 1..n})$. By (1) and (2), via the (Store Typing) rule, we have $\Sigma^+ \models \sigma^+$. Since $\Sigma^+ \models \diamond$ and $\Sigma^+ \equiv \Sigma, \downarrow \mapsto (A^+ \Rightarrow j)^{j \in 1..n}$, we have $\Sigma^+ \models [l_i = l_i^{i \in 1..n}] : A^+$ by the (Result Object) rule.

We conclude that $\Sigma^+ \geq \Sigma \wedge \Sigma^+ \models \sigma^+ \wedge dom(\sigma^+) = dom(\Sigma^+) \wedge \Sigma^+ \models [l_i = l_i^{i \in 1..n}] : A^+ \wedge \emptyset \vdash A^+ <: A\{\leftarrow T\}$.

Case (Red Select)

$$\frac{\begin{array}{c} \sigma \cdot S \vdash a \rightsquigarrow [l_i = l_i^{i \in 1..n}] \cdot \sigma' \quad \sigma'(\downarrow_j) = (\zeta(x_j)b_j, S') \quad x_j \notin dom(S') \quad j \in 1..n \\ \sigma \cdot (S', x_j \mapsto [l_i = l_i^{i \in 1..n}]) \vdash b_j \rightsquigarrow v \cdot \sigma'' \end{array}}{\sigma \cdot S \vdash a.l_j \rightsquigarrow v \cdot \sigma''}$$

By hypothesis $E \vdash a.l_j : A \wedge \Sigma \models \sigma \wedge dom(\sigma) = dom(\Sigma) \wedge \Sigma \models S.T : E$.

Since $E \vdash a.l_j : A$, we must have $E \vdash a.l_j : B_j\{C\}$ by (Val Select), with $E \vdash B_j\{C\} <: A$, and $E \vdash a : C$ and $E \vdash C <: D$, where D has the form $Obj(X)[l_i v_i; B_i\{X\}, \dots]$ and $v_i \in \{^\circ, ^+\}$.

By induction hypothesis, since $E \vdash a : C \wedge \sigma \cdot S \vdash a \rightsquigarrow [l_i = l_i^{i \in 1..n}] \cdot \sigma' \wedge \Sigma \models \sigma \wedge dom(\sigma) = dom(\Sigma) \wedge \Sigma \models S.T : E$, there exist a closed type A' and a store type Σ' such that $\Sigma' \geq \Sigma \wedge \Sigma' \models \sigma' \wedge dom(\sigma') = dom(\Sigma') \wedge \Sigma' \models [l_i = l_i^{i \in 1..n}] : A' \wedge \emptyset \vdash A' <: C\{\leftarrow T\}$.

Since $\sigma'(\downarrow_j) = (\zeta(x_j)b_j, S')$, the judgment $\Sigma' \models \sigma'$ must come via (Store Typing) from $\Sigma' \models S'.\emptyset : E_j$ and $E_j, x_j : \Sigma'_1(\downarrow_j) \vdash b_j : \Sigma'_2(\Sigma'_1(\downarrow_j), \downarrow_j)$ for some E_j . Since $\Sigma' \models [l_i = l_i^{i \in 1..n}] : A'$ must come from (Result Object), we have $A' \equiv Obj(X)[l_i v_i; \Sigma'_2(X, \downarrow_i)^{i \in 1..n}] \equiv \Sigma'_1(\downarrow_i)$. Since $E \vdash C <: D$ and $\Sigma \models S.T : E$, we obtain $\emptyset \vdash C\{\leftarrow T\} <: D\{\leftarrow T\}$ by Lemma 17.5-2, and $\emptyset \vdash$

$A' <: D\{\leftarrow T\}$ by transitivity. By Lemma 16.4-4 we must have $v_j \in \{^0, +\}$ and $\emptyset, X <: A'$ $\vdash \Sigma'_2(X, \iota_j) <: B_j\{X\}\{\leftarrow T\}$ with $X \notin \text{dom}(T)$. Hence $\emptyset \vdash \Sigma'_2(A', \iota_j) <: B_j\{A'\}\{\leftarrow T\}$ by Lemma 17.5-1. Then, from $E_j, x_j:\Sigma'_1(\iota_j) \vdash b_j : \Sigma'_2(\Sigma'_1(\iota_j), \iota_j)$, that is, from $E_j, x_j:A' \vdash b_j : \Sigma'_2(A', \iota_j)$, we obtain $E_j, x_j:A' \vdash b_j : B_j\{A'\}\{\leftarrow T\}$ by subsumption. Moreover, from $\Sigma' \models S' \cdot \emptyset : E_j$ and $\Sigma' \models [l_i = \iota_i]_{i \in 1..n} : A'$ we get $\Sigma' \models (S', x_j \mapsto [l_i = \iota_i]_{i \in 1..n}) \cdot \emptyset : (E_j, x_j:A')$ by (Stack x Typing).

Let $E' \equiv E_j, x_j:A'$. By induction hypothesis, since $E' \vdash b_j : B_j\{A'\}\{\leftarrow T\} \wedge \sigma' \cdot (S', x_j \mapsto [l_i = \iota_i]_{i \in 1..n}) \vdash b_j \rightsquigarrow v \cdot \sigma'' \wedge \Sigma' \models \sigma' \wedge \text{dom}(\sigma') = \text{dom}(\Sigma') \wedge \Sigma' \models (S', x_j \mapsto [l_i = \iota_i]_{i \in 1..n}) \cdot \emptyset : E'$, there exist a closed type A^\dagger and a store type Σ^\dagger such that $\Sigma^\dagger \succcurlyeq \Sigma' \wedge \Sigma^\dagger \models \sigma'' \wedge \text{dom}(\sigma'') = \text{dom}(\Sigma^\dagger) \wedge \Sigma^\dagger \models v : A^\dagger \wedge \emptyset \vdash A^\dagger <: B_j\{A'\}\{\leftarrow T\}$.

We conclude:

- $\Sigma^\dagger \succcurlyeq \Sigma$ by transitivity from $\Sigma^\dagger \succcurlyeq \Sigma'$ and $\Sigma' \succcurlyeq \Sigma$.
- $\Sigma^\dagger \models \sigma''$ with $\text{dom}(\sigma'') = \text{dom}(\Sigma^\dagger)$.
- $\Sigma^\dagger \models v : A^\dagger$.
- $\emptyset \vdash A^\dagger <: A\{\leftarrow T\}$. Since $E, X <: \text{Top} \vdash B_j\{X\}$ and $B_j\{X\}$ is covariant in X , we also have $\emptyset, X <: \text{Top} \vdash B_j\{X\}\{\leftarrow T\}$ by Lemmas 17.5-2(3) and 17.5-1(2), with $B_j\{X\}\{\leftarrow T\}$ covariant in X . Since $\emptyset \vdash A' <: C\{\leftarrow T\}$, we obtain $\emptyset \vdash B_j\{A'\}\{\leftarrow T\} <: B_j\{C\}\{\leftarrow T\}$, by Lemma 16.4-6, that is, $\emptyset \vdash B_j\{A'\}\{\leftarrow T\} <: B_j\{C\}\{\leftarrow T\}$. Since $E \vdash B_j\{C\} <: A$, we obtain $\emptyset \vdash B_j\{C\}\{\leftarrow T\} <: A\{\leftarrow T\}$ by Lemmas 17.5-2(3) and 17.5-1(2). We conclude that $\emptyset \vdash A^\dagger <: A\{\leftarrow T\}$ by transitivity from $\emptyset \vdash A^\dagger <: B_j\{A'\}\{\leftarrow T\}, \emptyset \vdash B_j\{A'\}\{\leftarrow T\} <: B_j\{C\}\{\leftarrow T\}$, and $\emptyset \vdash B_j\{C\}\{\leftarrow T\} <: A\{\leftarrow T\}$.

Case (Red Update)

$$\frac{\begin{array}{c} \sigma \cdot S \vdash a \rightsquigarrow [l_i = \iota_i]_{i \in 1..n} \cdot \sigma' \quad j \in 1..n \quad \iota_j \in \text{dom}(\sigma') \\ \sigma' \cdot (S, y \mapsto [l_i = \iota_i]_{i \in 1..n}) \vdash c \rightsquigarrow v \cdot \sigma'' \end{array}}{\sigma \cdot S \vdash a.l_j \#(y, z=c)\zeta(x)b \rightsquigarrow [l_i = \iota_i]_{i \in 1..n} \cdot (\sigma'' \cdot \iota_j \rightsquigarrow (\zeta(x)b, (S, y \mapsto [l_i = \iota_i]_{i \in 1..n}, z \mapsto v)))}$$

By hypothesis $E \vdash a.l_j \#(y, z=c)\zeta(x)b : A \wedge \Sigma \models \sigma \wedge \text{dom}(\sigma) = \text{dom}(\Sigma) \wedge \Sigma \models S \cdot T : E$.

Since $E \vdash a.l_j \#(y, z=c)\zeta(x)b : A$, we must have $E \vdash a.l_j \#(y, z=c)\zeta(x)b : A'$ by (Val Update), for some A' with $E \vdash A' <: A$. Hence we have $E \vdash a : A'$ and $E \vdash A' <: D$ where D has the form $\text{Obj}(X)[l_i v_i : B_j\{X\}, \dots]$ and $E, Y <: A', y : Y \vdash c : C$ and $E, Y <: A', y : Y, z : C, x : Y \vdash b : B_j\{Y\}$ and $v_i \in \{^0, +\}$. Since $Y \notin \text{dom}(E)$, we have $Y \notin \text{dom}(T)$ by Lemma 17.5-2(1).

By induction hypothesis, since $E \vdash a : A' \wedge \sigma \cdot S \vdash a \rightsquigarrow [l_i = \iota_i]_{i \in 1..n} \cdot \sigma' \wedge \Sigma \models \sigma \wedge \text{dom}(\sigma) = \text{dom}(\Sigma) \wedge \Sigma \models S \cdot T : E$, there exist a closed type A^\dagger and a store type Σ' such that $\Sigma' \succcurlyeq \Sigma \wedge \Sigma' \models \sigma' \wedge \text{dom}(\sigma') = \text{dom}(\Sigma') \wedge \Sigma' \models [l_i = \iota_i]_{i \in 1..n} : A^\dagger \wedge \emptyset \vdash A^\dagger <: A'\{\leftarrow T\}$.

Now, $\Sigma' \models [l_i = \iota_i]_{i \in 1..n} : A^\dagger$ must have been derived via (Result Object) from $\Sigma' \models \diamond$ and $\Sigma'_1(\iota_i) \equiv \text{Obj}(X)[l_i v_i : \Sigma'_2(X, \iota_i)]_{i \in 1..n} \equiv A^\dagger$ for all $i \in 1..n$. We have $\emptyset \vdash A'\{\leftarrow T\} <: D\{\leftarrow T\}$ by Lemmas 17.5-2(3) and 17.5-1(2), and then $\emptyset \vdash A^\dagger <: D\{\leftarrow T\}$ by transitivity.

Hence by Lemma 16.4-4 we must have $v_j \in \{^0, -\}$ and $\emptyset, X <: A^\dagger \vdash B_j[X]\{\leftarrow T\} <: \Sigma'_2(X, \iota_j)$, for $X \notin \text{dom}(T)$. Therefore, $\emptyset \vdash B_j[A^\dagger]\{\leftarrow T\} <: \Sigma'_2(A^\dagger, \iota_j)$ by Lemma 17.5-1.

Let $T' \equiv T$, $Y \mapsto A^\dagger$. From $\Sigma \models S.T : E$ we have $\Sigma' \models S.T : E$ by Lemma 17.5-3. Then from $\emptyset \vdash A^\dagger <: A'\{\leftarrow T\}$ we obtain $\Sigma' \models S.T' : (E, Y <: A')$ by (Stack X Typing). Then from $\Sigma' \models [l_i = l_i^{i \in 1..n}] : A^\dagger \equiv Y\{\leftarrow T'\}$ we obtain $\Sigma' \models (S, y \mapsto [l_i = l_i^{i \in 1..n}]) \bullet T' : (E, Y <: A', y:Y)$ by (Stack x Typing).

By induction hypothesis, since $E, Y <: A', y:Y \vdash c : C \wedge \sigma' \bullet (S, y \mapsto [l_i = l_i^{i \in 1..n}]) \vdash c \rightsquigarrow v \bullet \sigma'' \wedge \Sigma' \models \sigma' \wedge \text{dom}(\sigma') = \text{dom}(\Sigma') \wedge \Sigma' \models (S, y \mapsto [l_i = l_i^{i \in 1..n}]) \bullet T' : (E, Y <: A', y:Y)$, there exist a closed type C^\dagger and a store type Σ^\dagger such that $\Sigma^\dagger \geq \Sigma' \wedge \Sigma^\dagger \models \sigma'' \wedge \text{dom}(\sigma'') = \text{dom}(\Sigma^\dagger) \wedge \Sigma^\dagger \models v : C^\dagger \wedge \emptyset \vdash C^\dagger <: C\{\leftarrow T'\}$.

Take $\sigma^\dagger \equiv \sigma''.\iota_j \mapsto (\zeta(x)b, (S, y \mapsto [l_i = l_i^{i \in 1..n}], z \mapsto v))$.

- $\Sigma^\dagger \geq \Sigma$ by transitivity.
- We have $\text{dom}(\Sigma^\dagger) = \text{dom}(\sigma'')$. Moreover $\text{dom}(\sigma') \subseteq \text{dom}(\sigma'')$ by the definition of reduction, and hence $\iota_j \in \text{dom}(\sigma'')$. So σ^\dagger is well-defined, and $\text{dom}(\sigma'') = \text{dom}(\sigma^\dagger)$. Therefore $\text{dom}(\Sigma^\dagger) = \text{dom}(\sigma^\dagger)$.
- By (1), (2), and (3) below, we obtain $\Sigma^\dagger \models \sigma^\dagger$ via the (Store Typing) rule.

(1) From $\Sigma' \models S.T : E$ we have $\Sigma' \models S.\emptyset : E\{\leftarrow T\}$ by Lemma 17.5-2(2). From $\Sigma' \models [l_i = l_i^{i \in 1..n}] : A^\dagger$ we have $\Sigma' \models (S, y \mapsto [l_i = l_i^{i \in 1..n}]) \bullet \emptyset : (E\{\leftarrow T\}, y:A^\dagger)$ by (Stack x Typing). By Lemma 17.5-3 we have $\Sigma^\dagger \models (S, y \mapsto [l_i = l_i^{i \in 1..n}]) \bullet \emptyset : (E\{\leftarrow T\}, y:A^\dagger)$. From $\Sigma^\dagger \models v : C^\dagger$ we have $\Sigma^\dagger \models (S, y \mapsto [l_i = l_i^{i \in 1..n}], z \mapsto v) \bullet \emptyset : (E\{\leftarrow T\}, y:A^\dagger, z:C^\dagger)$ by (Stack x Typing).

(2) From $\Sigma \models S.T : E$ and $E, Y <: A', y:Y, z:C, x:Y \vdash b : B_j$ we obtain $E\{\leftarrow T\}, Y <: A'\{\leftarrow T\}, y:Y, z:C\{\leftarrow T\}, x:Y \vdash b : B_j\{\leftarrow T\}$ by Lemma 17.5-2(3). Since $\emptyset \vdash A^\dagger <: A'\{\leftarrow T\}$ and $\emptyset \vdash C^\dagger <: C\{\leftarrow T\}$ we have $E\{\leftarrow T\}, y:A^\dagger, z:C^\dagger, x:A^\dagger \vdash b : B_j\{\leftarrow T\}$ by Lemma 17.5-1(4). Since $\emptyset \vdash B_j[A^\dagger]\{\leftarrow T\} \equiv B_j\{\leftarrow T\} <: \Sigma'_2(A^\dagger, \iota_j)$ we have $E\{\leftarrow T\}, y:A^\dagger, z:C^\dagger, x:A^\dagger \vdash b : \Sigma'_2(A^\dagger, \iota_j)$ by subsumption. That is, $E\{\leftarrow T\}, y:A^\dagger, z:C^\dagger, x:\Sigma'_1(\iota_j) \vdash b : \Sigma'_2(\Sigma'_1(\iota_j), \iota_j)$.

(3) Since $\Sigma^\dagger \models \sigma''$ must come from (Store Typing), σ'' has the shape $\varepsilon_k \mapsto (\zeta(x_k)b_k, S_k)$ $k \in 1..m$, and for all k such that $\varepsilon_k \neq \iota_j$ and for some E_k we have $\Sigma^\dagger \models S_k \bullet \emptyset : E_k$ and $E_k \times_k \Sigma^\dagger_1(\varepsilon_k) \vdash b_k : \Sigma^\dagger_2(\Sigma^\dagger_1(\varepsilon_k), \varepsilon_k)$.

- From $\Sigma' \models [l_i = l_i^{i \in 1..n}] : A^\dagger$ and $\Sigma^\dagger \geq \Sigma'$, by Lemma 17.5-3 we have $\Sigma^\dagger \models [l_i = l_i^{i \in 1..n}] : A^\dagger$.
- By Lemmas 17.5-2(3) and 17.5-1(2) we have $\emptyset \vdash A'\{\leftarrow T\} <: A\{\leftarrow T\}$, so by transitivity $\emptyset \vdash A^\dagger <: A\{\leftarrow T\}$.

We conclude that $\Sigma^\dagger \geq \Sigma \wedge \Sigma^\dagger \models \sigma^\dagger \wedge \text{dom}(\Sigma^\dagger) = \text{dom}(\sigma^\dagger) \wedge \Sigma^\dagger \models [l_i = l_i^{i \in 1..n}] : A^\dagger \wedge \emptyset \vdash A^\dagger <: A\{\leftarrow T\}$.

Case (Red Clone) (ι_i distinct)

$$\frac{\sigma \bullet S \vdash a \rightsquigarrow [l_i = l_i^{i \in 1..n}] \bullet \sigma' \quad \iota_i \in \text{dom}(\sigma') \quad \iota_i' \notin \text{dom}(\sigma') \quad \forall i \in 1..n}{\sigma \bullet S \vdash \text{clone}(a) \rightsquigarrow [l_i = l_i^{i \in 1..n}] \bullet (\sigma', \iota_i' \mapsto \sigma'(\iota_i))^{i \in 1..n}}$$

By hypothesis $E \vdash \text{clone}(a) : A \wedge \Sigma \models \sigma \wedge \text{dom}(\sigma) = \text{dom}(\Sigma) \wedge \Sigma \models S.T : E$.

Since $E \vdash \text{clone}(a) : A$, we must have $E \vdash \text{clone}(a) : A'$ by (Val Clone), for some A' with $E \vdash a : A'$ and $E \vdash A' <: D$ (where D is an object type), and such that $E \vdash A' <: A$.

By induction hypothesis, since $E \vdash a : A' \wedge \sigma \cdot S \vdash a \rightsquigarrow [l_i=l_i^{i \in 1..n}] \cdot \sigma' \wedge \Sigma \models \sigma \wedge \text{dom}(\sigma) = \text{dom}(\Sigma) \wedge \Sigma \models S.T : E$, there exist a closed type A^\dagger and a store type Σ' such that $\Sigma' \geq \Sigma \wedge \Sigma' \models \sigma' \wedge \text{dom}(\sigma') = \text{dom}(\Sigma') \wedge \Sigma' \models [l_i=l_i^{i \in 1..n}] : A^\dagger \wedge \emptyset \vdash A^\dagger <: A' \{\leftarrow T\}$.

Let $\Sigma^\dagger \equiv (\Sigma', l_i' \mapsto \Sigma'(l_i)_{i \in 1..n})$ and $\sigma^\dagger \equiv (\sigma', l_i' \mapsto \sigma'(l_i)_{i \in 1..n})$. We have $\Sigma^\dagger \models \diamond$ by (Store Type) because $l_i \notin \text{dom}(\sigma') = \text{dom}(\Sigma')$, l_i' are all distinct, and $\Sigma' \models \diamond$ (since $\Sigma' \models \sigma'$).

We conclude:

- $\emptyset \vdash A^\dagger <: A' \{\leftarrow T\} <: A \{\leftarrow T\}$, by Lemmas 17.5-2(3) and 17.5-1(2).
- $\Sigma^\dagger \geq \Sigma$, because $\Sigma' \geq \Sigma$ and $\Sigma^\dagger \geq \Sigma'$.
- $\text{dom}(\sigma^\dagger) = \text{dom}(\Sigma^\dagger)$, by construction and $\text{dom}(\sigma') = \text{dom}(\Sigma')$.
- We show that $\Sigma^\dagger \models \sigma^\dagger$. Since $\Sigma' \models \sigma'$ must come from (Store Typing), σ' is of the form $\varepsilon_k \mapsto \langle \zeta(x_k)b_k, S_k \rangle_{k \in 1..m}$, and for all $k \in 1..m$ and for some E_k we have $\Sigma' \models S_k \cdot \emptyset : E_k$ and $E_k, x_k : \Sigma'_1(\varepsilon_k) \vdash b_k : \Sigma'_2(\Sigma'_1(\varepsilon_k), \varepsilon_k)$. Then also $E_k, x_k : \Sigma^\dagger_1(\varepsilon_k) \vdash b_k : \Sigma^\dagger_2(\Sigma^\dagger_1(\varepsilon_k), \varepsilon_k)$, and $\Sigma^\dagger \models S_k \cdot \emptyset : E_k$ by Lemma 17.5-3. Let $f : 1..n \rightarrow 1..m$ be $\varepsilon^{-1}\circ\iota$, so that for all $i \in 1..n$, $l_i = \varepsilon f(i)$. We have $E_{f(i)}, x_{f(i)} : \Sigma'_1(\varepsilon f(i)) \vdash b_{f(i)} : \Sigma'_2(\Sigma'_1(\varepsilon f(i)), \varepsilon f(i))$ for $i \in 1..n$, so $E_{f(i)}, x_{f(i)} ; \Sigma'_1(l_i) \vdash b_{f(i)} : \Sigma'_2(\Sigma'_1(l_i), l_i)$. Moreover, since $\Sigma'(l_i) = \Sigma^\dagger(l_i')$, we have $E_{f(i)}, x_{f(i)} ; \Sigma^\dagger_1(l_i') \vdash b_{f(i)} : \Sigma^\dagger_2(\Sigma^\dagger_1(l_i'), l_i')$. The result follows by (Store Typing) from $\Sigma^\dagger \models S_k \cdot \emptyset : E_k$ and $\Sigma^\dagger \models S_{f(i)} \cdot \emptyset : E_{f(i)}$, and $E_k, x_k : \Sigma^\dagger_1(\varepsilon_k) \vdash b_k : \Sigma^\dagger_2(\Sigma^\dagger_1(\varepsilon_k), \varepsilon_k)$ and $E_{f(i)}, x_{f(i)} ; \Sigma^\dagger_1(l_i') \vdash b_{f(i)} : \Sigma^\dagger_2(\Sigma^\dagger_1(l_i'), l_i')$, for $k \in 1..m$ and $i \in 1..n$.
- We show that $\Sigma^\dagger \models [l_i=l_i^{i \in 1..n}] : A^\dagger$. First, $\Sigma' \models [l_i=l_i^{i \in 1..n}] : A^\dagger$ must come from the (Result Object) rule with $A^\dagger \equiv \Sigma'_1(l_i) \equiv \text{Obj}(X)[l_i; \forall i : \Sigma'_2(X, l_i)]_{i \in 1..n}$ for $i \in 1..n$, and $\Sigma' \models \diamond$. But $\Sigma^\dagger(l_i') \equiv \Sigma'(l_i)$ for $i \in 1..n$. So, $\Sigma^\dagger_1(l_i') \equiv \Sigma'_1(l_i) \equiv A^\dagger \equiv \text{Obj}(X)[l_i; \forall i : \Sigma'_2(X, l_i)]_{i \in 1..n} \equiv \text{Obj}(X)[l_i; \forall i : \Sigma^\dagger_2(X, l_i')]_{i \in 1..n}$, and by (Result Object) $\Sigma^\dagger \models [l_i=l_i^{i \in 1..n}] : \text{Obj}(X)[l_i; \forall i : \Sigma^\dagger_2(X, l_i')]_{i \in 1..n}$.

Case (Red Fun2)

$$\overline{\sigma \cdot S \vdash \lambda()b \rightsquigarrow (\lambda()b, S) \cdot \sigma}$$

By hypothesis, $E \vdash \lambda()b : A \wedge \Sigma \models \sigma \wedge \text{dom}(\sigma) = \text{dom}(\Sigma) \wedge \Sigma \models S.T : E$.

From $\Sigma \models S.T : E$ and $E \vdash \lambda()b : A$ we have $E \{\leftarrow T\} \vdash \lambda()b : A \{\leftarrow T\}$ by Lemma 17.5-2(3). Since $E \{\leftarrow T\} \vdash \lambda()b : A \{\leftarrow T\}$ we must have $E \{\leftarrow T\} \vdash \lambda()b : \forall(X <: C)B$ for some type $\forall(X <: C)B$ such that $E \{\leftarrow T\}, X <: C \vdash b : B$ and $E \{\leftarrow T\} \vdash \forall(X <: C)B <: A \{\leftarrow T\}$.

We have $\Sigma \models S \cdot \emptyset : E \{\leftarrow T\}$ by Lemma 17.5-2(2), and hence $\Sigma \models (\lambda()b, S) : \forall(X <: C)B$ by (Result Fun2 $\{\leftarrow\}$). Moreover, $\emptyset \vdash \forall(X <: C)B <: A \{\leftarrow T\}$ by Lemma 17.5-1(2), since $E \{\leftarrow T\}$ has no type variables.

Take $A^\dagger \equiv \forall(X <: C)B$ and $\Sigma^\dagger \equiv \Sigma$. We conclude $\Sigma^\dagger \geq \Sigma \wedge \Sigma^\dagger \models \sigma \wedge \text{dom}(\sigma) = \text{dom}(\Sigma^\dagger) \wedge \Sigma^\dagger \models (\lambda()b, S) : A^\dagger \wedge \emptyset \vdash A^\dagger <: A\{\leftarrow T\}$.

Case (Red Appl2)

$$\frac{\sigma \cdot S \vdash a \rightsquigarrow (\lambda()b, S') \cdot \sigma' \quad \sigma' \cdot S' \vdash b \rightsquigarrow v \cdot \sigma''}{\sigma \cdot S \vdash a() \rightsquigarrow v \cdot \sigma''}$$

By hypothesis, $E \vdash a() : A \wedge \Sigma \models \sigma \wedge \text{dom}(\sigma) = \text{dom}(\Sigma) \wedge \Sigma \models S \cdot T : E$.

Since $E \vdash a() : A$ we must have $E \vdash a() : B\{C\}$ with $E \vdash B\{C\} <: A$ by (Val Appl2<:) for some types C and $\forall(X <: D)B\{X\}$ such that $E \vdash a : \forall(X <: D)B\{X\}$ and $E \vdash C <: D$. We obtain $\emptyset \vdash C\{\leftarrow T\} <: D\{\leftarrow T\}$, by Lemmas 17.5-2(3) and 17.5-1(2).

By induction hypothesis, since $E \vdash a : \forall(X <: D)B \wedge \sigma \cdot S \vdash a \rightsquigarrow (\lambda()b, S') \cdot \sigma' \wedge \Sigma \models \sigma \wedge \text{dom}(\sigma) = \text{dom}(\Sigma) \wedge \Sigma \models S \cdot T : E$, there exist a closed type A' and a store type Σ' such that $\Sigma' \geq \Sigma \wedge \Sigma' \models \sigma' \wedge \text{dom}(\sigma') = \text{dom}(\Sigma') \wedge \Sigma' \models (\lambda()b, S') : A' \wedge \emptyset \vdash A' <: (\forall(X <: D)B)\{\leftarrow T\}$.

Now $\Sigma' \models (\lambda()b, S') : A'$ must come from (Result Fun2<:), with $A' \equiv \forall(X <: D')B'$, $\Sigma' \models S' \cdot \emptyset : E'$, and $E', X <: D' \vdash b : B'$. We have also $\emptyset \vdash D\{\leftarrow T\} <: D'$ and $\emptyset, X <: D\{\leftarrow T\} \vdash B' <: B\{\leftarrow T\}$ since $\emptyset \vdash A' <: (\forall(X <: D)B)\{\leftarrow T\}$, by Lemma 16.4-5.

Take $T' \equiv \emptyset, X \rightarrow C\{\leftarrow T\}$. We have $\Sigma' = S' \cdot \emptyset : E'$ and $\emptyset \vdash C\{\leftarrow T\} <: D'$ by transitivity. Therefore, $\Sigma' \models S' \cdot T' : (E', X <: D')$ by (Stack X Typing).

By induction hypothesis, since $E', X <: D' \vdash b : B' \wedge \sigma' \cdot S' \vdash b \rightsquigarrow v \cdot \sigma'' \wedge \Sigma' \models \sigma' \wedge \text{dom}(\sigma') = \text{dom}(\Sigma') \wedge \Sigma' \models S' \cdot T' : (E', X <: D')$, there exist a closed type A^\dagger and a store type Σ^\dagger such that $\Sigma^\dagger \geq \Sigma' \wedge \Sigma^\dagger \models \sigma'' \wedge \text{dom}(\sigma'') = \text{dom}(\Sigma^\dagger) \wedge \Sigma^\dagger \models v : A^\dagger \wedge \emptyset \vdash A^\dagger <: B'\{\leftarrow T'\}$.

We have $(B'\{X\})\{\leftarrow T'\} \equiv B'\{C\{\leftarrow T\}\}$ by definition of T' , hence $\emptyset \vdash A^\dagger <: B'\{C\{\leftarrow T\}\}$. Since $\emptyset \vdash C\{\leftarrow T\} <: D\{\leftarrow T\}$ and $\emptyset, X <: D\{\leftarrow T\} \vdash B'\{X\} <: B\{X\}\{\leftarrow T\}$, we have $\emptyset \vdash B'\{C\{\leftarrow T\}\} <: B'\{C\{\leftarrow T\}\}\{\leftarrow T\}$ by Lemma 17.5-1. Since $E \vdash B\{C\} <: A$, we have $\emptyset \vdash (B\{C\})\{\leftarrow T\} <: A\{\leftarrow T\}$ by Lemmas 17.5-2(3) and 17.5-1(2). Finally $\emptyset \vdash A^\dagger <: A\{\leftarrow T\}$ by transitivity.

We conclude $\Sigma^\dagger \geq \Sigma \wedge \Sigma^\dagger \models \sigma'' \wedge \text{dom}(\sigma'') = \text{dom}(\Sigma^\dagger) \wedge \Sigma^\dagger \models v : A^\dagger \wedge \emptyset \vdash A^\dagger <: A\{\leftarrow T\}$.

□

We have also an analogue to Corollary 11.4-4:

Corollary 17.5-5

If $\emptyset \vdash a : A$ and $\emptyset \cdot \emptyset \vdash a \rightsquigarrow v \cdot \sigma$, then there exist a type A^\dagger and a store type Σ^\dagger such that $\Sigma^\dagger \models \sigma$ and $\Sigma^\dagger \models v : A^\dagger$, with $\emptyset \vdash A^\dagger <: A$.

□

This corollary implies, for example, that if $\emptyset \vdash a : A$ and $\emptyset \cdot \emptyset \vdash a \rightsquigarrow v \cdot \sigma$ where A is an object type, then v is an object result of the form $[l_i = l_i]_{i \in 1..n}$ and never a type abstraction result of the form $(\lambda()b, S)$.

18 INTERPRETATIONS OF OBJECT CALCULI

Now that we have investigated several object calculi in much detail, we are in a good position to address a basic question:

Why should we study object calculi, instead of encoding objects in λ -calculi?

In order to answer this question, we describe several techniques for encoding objects, resuming the discussion of Section 6.7. These techniques provide interesting interpretations of objects. Still, we argue, these interpretations are not a replacement for object calculi.

18.1 The Interpretation Problem

Before describing particular interpretations of objects, we discuss the problem of finding interpretations, and cover some preliminary material.

18.1.1 Encoding Objects and Subtyping

On a case-by-case basis, several techniques have been developed to write λ -calculus expressions that emulate object-oriented programs. To be reasonably satisfactory, however, these emulation techniques should not be applied only to selected examples; they should be systematized.

Object calculi provide a testbed: the adequacy of these emulation techniques can be debated formally in terms of the existence of interpretations of object calculi within λ -calculi. We survey several such interpretations, pointing out their achievements and their deficiencies. Many of these interpretations are well known, but some are new. In particular, we introduce a new technique, developed in joint work with Viswanathan [9], that successfully combines many of the ideas developed in the past. We do not deal with techniques based on extensible records [43], because they require type systems for λ -calculi that are not discussed in this book.

The question stated above has a pragmatic analogue: why should we use object-oriented languages, instead of emulating objects in procedural languages? If object-oriented languages could be easily reduced to procedural languages, we could do without them. The wide diffusion of object-oriented languages seems to suggest that this reduction is not easy or practical. Correspondingly, as we shall see, faithful interpretations of object calculi in λ -calculi are not easy to find. The available interpretations do not lead to convenient programming techniques. Moreover, there is no canonical interpre-

tation: interpretations differ in their complexity, in the range of features they can model, and in the convenience of expressing examples.

A faithful interpretation of object calculi is one that preserves all judgments, in a formal sense. For typed calculi, in particular, we expect the preservation of subtyping judgments. That is, if a subtype relation exists between types of an object calculus, then a corresponding subtype relation should exist between the interpretations of those types in a λ -calculus with subtyping.

It is easier to find interpretations of object calculi that do not preserve subtyping. These interpretations map uses of the subsumption rule to the insertion of coercion functions [2]. Such interpretations are unsatisfactory in two respects. First, they fail to explain object subtyping in terms of more primitive subtype relations (e.g., in terms of record subtyping). Second, by introducing cumbersome coercion functions in place of subsumption, they fail to preserve the flavor of the original programs.

Therefore we strive to find interpretations that preserve subtyping. This turns out to be a difficult task that stimulates the development of sophisticated techniques. The difficulty of the task, in a sense, justifies the study of object calculi as separate systems.

18.1.2 Features

To understand the full scope of the problem of finding interpretations, and to learn potentially useful techniques, it is important to discuss interpretations that do not quite work. As a consequence, we do not always define the interpretations formally; furthermore, the interpretations generally fail to validate some of the rules of object calculi.

We discuss a number of interpretation techniques. Most of these can handle only restricted forms of our object calculi, for example calculi that distinguish fields from methods or that forbid method update. As a guide, we list below the syntactic forms that are handled (to different degrees) by the various interpretations. These tables are only an approximation: each interpretation handles a specific syntax with specific restrictions. Informally, we categorize operations as *internal* when they act on self (or an updated self) in the context of an object definition, and as *external* otherwise. Some interpretations deal differently with the two kinds of operations, and some deal only with one kind. Other interpretations treat these two kinds of operations uniformly.

We consider the following extended syntax for object calculi with separate fields and methods:

Syntax for separate fields and methods

$[f_k : B_k \ k \in 1..m \mid l_i : B_i \ i \in 1..n]$	object type with m fields and n methods
$[f_k = b_k \ k \in 1..m \mid l_i = \varsigma(x_i : A)b_i[x_i] \ i \in 1..n]$	object with m fields and n methods
$o_{A:f}$	field selection
$o_{A:f} := b$	field update
$o_A.l$	method invocation
$o.l = \varsigma(y : A)b$	method update

The subscripted term o_A indicates that o has type A ; we omit the subscript when this information is not needed. For untyped interpretations, we consider versions of this syntax where all type information has been erased. For imperative interpretations, we consider an additional cloning primitive.

When considering object types with Self, we write types and terms of the following forms:

Syntax with Self types

$\text{Obj}(X)[f_k:B_k \ k \in 1..m \mid l_i:B_i\{X^+\} \ i \in 1..n]$	object type with m fields and n methods
$\text{obj}(X=A)[f_k=b_k \ k \in 1..m \mid l_i=\varsigma(x_i:X)b_i\{X,x_i\} \ i \in 1..n]$	object with m fields and n methods
$o_A.f$	field selection
$o_A.f := b$	field update
$o_A.l$	method invocation
$o.l_j \Leftarrow (X <: A, y:X) \varsigma(x:X)b$	method update (with old self)

In $\text{Obj}(X)[f_k:B_k \ k \in 1..m \mid l_i:B_i\{X^+\} \ i \in 1..n]$, the variable X occurs only covariantly in the B_i ; moreover, X does not occur in the B_k and b_k .

When $m = 0$, the syntax in the previous two tables yields calculi with no fields (or, more precisely, calculi where fields are encoded as methods). This special case corresponds to our standard syntax with types of the form $[l_i:B_i \ i \in 1..n]$ or $\text{Obj}(X)[l_i:B_i\{X^+\} \ i \in 1..n]$ and objects of the form $[l_i=\varsigma(x_i:A)b_i\{x_i\} \ i \in 1..n]$ or $\text{obj}(X=A)[l_i=\varsigma(x_i:X)b_i\{X,x_i\} \ i \in 1..n]$.

18.2 Untyped Interpretations

We begin by reviewing several untyped interpretations of object calculi. These are used as the basis of typed interpretations in the next section. The interpretations rely on functions and records. We recall our notation for records: $\langle l_i=b_i \ i \in 1..n \rangle$ is a record; $a.l$ is record selection; and $a.l := b$ is record update.

18.2.1 Self-Application

The simplest interpretation of object calculi is the one that implements method invocation as self-application [73, 74]. The self-application interpretation maps objects to records of functions. On method invocation, a whole object is passed to a method as a parameter. In a functional context, we have:

Untyped self-application interpretation

$[l_i=\varsigma(x_i)b_i \ i \in 1..n] \triangleq \langle l_i=\lambda(x_i)b_i \ i \in 1..n \rangle$	$(l_i \text{ distinct})$
$o.l_j \triangleq o.l_j(o)$	$(j \in 1..n)$
$o.l_j \Leftarrow \varsigma(y)b \triangleq o.l_j := \lambda(y)b$	$(j \in 1..n)$

As discussed in Section 6.7, this interpretation correctly represents the semantics of both method invocation and method update, and it underlies most implementations of object-oriented languages.

The self-application interpretation can be easily adapted to an imperative context; cloning can be handled by an additional record component cl :

Untyped imperative self-application interpretation

$[f_k=b_k^{k \in 1..m} \mid l_i=\zeta(x_i)b_i^{i \in 1..n}] \triangleq$	$(f_k, l_i, cl \text{ distinct})$
$\begin{aligned} & (f_k=b_k^{k \in 1..m}, \\ & l_i=\lambda(x_i)b_i^{i \in 1..n}, \\ & cl=\lambda(x)(f_k=x.f_k^{k \in 1..m}, l_i=x.l_i^{i \in 1..n}, cl=x.cl)) \end{aligned}$	
$o.f_j \triangleq o.f_j$	$(j \in 1..m)$
$o.f_j := b \triangleq o.f_j := b$	$(j \in 1..m)$
$o.l_j \triangleq o.l_j(o)$	$(j \in 1..n)$
$o.l_j = \zeta(y).b \triangleq o.l_j := \lambda(y).b$	$(j \in 1..n)$
$clone(o) \triangleq o.cl(o)$	

Here we distinguish fields from methods because, imperatively, they call for different evaluation strategies.

18.2.2 Recursive Records

The recursive-record interpretation maps objects to recursively defined records [39, 74, 106]. As discussed earlier in Section 6.7, the equational rule for update (even field update) fails to hold under this interpretation, because update does not interact well with recursion.

Untyped recursive-record interpretation without update

$[l_i=\zeta(x_i)b_i\{x_i\}^{i \in 1..n}] \triangleq \mu(x)(l_i=b_i\{x\}^{i \in 1..n})$	$(l_i \text{ distinct})$
$o.l_j \triangleq o.l_j$	$(j \in 1..n)$

It is possible to modify the recursive-record interpretation so as to allow at least internal field update. This is achieved by defining an auxiliary function $init$ that is used both for the field initialization and for the creation of modified objects during update.

Untyped recursive-record interpretation with internal field update

$[f_k=b_k^{k \in 1..m} \mid l_i=\zeta(x_i)b_i\{x_i\}^{i \in 1..n}] \triangleq$	$(f_k, l_i \text{ distinct})$
$\begin{aligned} & let \text{rec } init(y_k^{k \in 1..m}) = \mu(x)(f_k=y_k^{k \in 1..m}, l_i=b_i\{x\}^{i \in 1..n}) \\ & in \text{ } init(b_k^{k \in 1..m}) \end{aligned}$	

$o \cdot f_j \triangleq o \cdot f_j$	$(j \in 1..m)$
$o \cdot f_j := b \triangleq \text{init}(o \cdot f_k^{k \in 1..j-1}, b, o \cdot f_k^{k \in j+1..m})$	$(j \in 1..m)$ (internal)
$o \cdot l_j \triangleq o \cdot l_j$	$(j \in 1..n)$

The occurrences of *init* in the translation of update are bound to the closest enclosing definition of *init*. Therefore only internal update is handled correctly. The translation of an external update would leave *init* unbound, or would bind it to the wrong definition. Subject to these limitations, this interpretation models object calculi with private fields over which only internal updates are allowed.

18.2.3 State-Application

The state-application interpretation is a variant of the self-application interpretation [103]. When considering calculi that support only field update, it becomes natural to treat fields and methods separately. The fields can be grouped into a state record *st*, separate from the method suite record *mt*. Methods receive *st* as a parameter on method invocation, instead of the whole object as in the self-application interpretation. The update operation modifies the *st* component and copies the *mt* component. The method suite is bound recursively with a μ , so that each method can invoke the others. Thus we obtain the following interpretation:

Untyped state-application interpretation

$[f_k = b_k^{k \in 1..m} \mid l_i = \zeta(x_i) b_i^{i \in 1..n}] \triangleq$	$(f_k, l_i \text{ distinct})$
$\langle st = \langle f_k = b_k^{k \in 1..m} \rangle, mt = \mu(m)(l_i = \lambda(s) b_i^{i \in 1..n}) \rangle$	(for appropriate b_i)
$o \cdot f_j \triangleq o \cdot st \cdot f_j$	$(j \in 1..m)$ (external)
$o \cdot f_j := b \triangleq \langle st = (o \cdot st \cdot f_j := b), mt = o \cdot mt \rangle$	$(j \in 1..m)$ (external)
$o \cdot l_j \triangleq o \cdot mt \cdot l_j(o \cdot st)$	$(j \in 1..n)$ (external)

In this interpretation it is difficult to express the precise translation of the method bodies, b_i , essentially because the separation of fields from methods causes self to split into two parts. Internal operations manipulate *s* directly, and are thus coded differently from external operations. Moreover, since the self parameter *s* gives access only to fields, internal method invocation is done through *m*:

Untyped state-application interpretation (continued)

$x_i \cdot f_j \triangleq s \cdot f_j$	in the context $\mu(m)(l_i = \lambda(s) \dots)$	$(j \in 1..m)$ (internal)
$x_i \cdot f_j := b \triangleq s \cdot f_j := b$	in the context $\mu(m)(l_i = \lambda(s) \dots)$	$(j \in 1..m)$ (internal)
$x_i \cdot l_j \triangleq m \cdot l_j(s)$	in the context $\mu(m)(l_i = \lambda(s) \dots)$	$(j \in 1..n)$ (internal)

There is also a more subtle difficulty. Methods that return self should produce a whole object, but the available self parameter contains only fields: a whole object, including a method suite, must be regenerated at the right time. The burden of regeneration can be placed either on the methods of the object, or on the user of the object. In the first case, methods produce structures of the form $\langle st=s', mt=m \rangle$, where s' is a modified state, and m is the recursively defined method suite. In the second case, the user of an object writes code such as the following:

```
let s = o·st and m = o·mt
in  let s' = ... s ... m·lj ...
     in ⟨st=s', mt=m⟩
```

The object o is decomposed into its state s and method suite m ; then various internal operations are performed using s and m , obtaining a new state s' ; finally a new object is assembled from s' and the initial m .

In either case, the code produced for methods that return self is rather different from the code produced for methods that do not return self. Although it is fairly clear how to translate specific examples, it is quite hard to describe formally an interpretation that works, especially in an untyped context. A proper formulation, however, can be found by letting types drive the translation; see Section 18.3.3.

When the translation is performed correctly, the state-application interpretation implements our primitive semantics of objects, except for the omission of proper method update. One advantage of this rather laborious encoding is that it corresponds to some extent to the behavior of class-based languages, where method suites are kept separate from state records.

18.2.4 Cyclic Records

The main problems with functional interpretations are ultimately due to difficulties in emulating state change. Side-effects trivially avoid these difficulties. Thus, in the presence of imperative features, additional encoding techniques become available [60].

The following cyclic-record interpretation is a version of the recursive-record interpretation where recursion is replaced by loops in the store. In defining the interpretation we use an informal imperative semantics for records. The construct $\langle f_k=b_k \mid k \in 1..m \rangle$ allocates storage for the components f_k , initializes them, and returns a reference to the result. The construct $a·f:=b$ is an assignment to the f component of the record a .

Untyped cyclic-record interpretation

$[f_k=b_k \mid k \in 1..m \mid l_i=\zeta(x_i)b_i\{x_i\}^{i \in 1..n}] \triangleq$	$(f_k, l_i \text{ distinct})$
let $x = \langle f_k=b_k \mid k \in 1..m, l_i=()^{i \in 1..n} \rangle$	$z \notin FV(b_i^{i \in 1..n}) \cup \{x\}$
in $x·l_i:=\lambda(z)b_i\{x\}^{i \in 1..n}; x$	
$o·f_j \triangleq o·f_j$	$(j \in 1..m)$
$o·f_j:=b \triangleq o·f_j:=b$	$(j \in 1..m)$

$o.l_j \triangleq o.l_j(\langle \rangle)$	$(j \in 1..n)$
$o.l_j = \zeta(y)b\{y\} \triangleq$	$(j \in 1..n)$
let $x = o$ in $x.l_j := \lambda(z)b\{x\}$	$z \notin FV(b) \cup \{x\}$

In the interpretation of object formation, a little imperative program allocates a skeletal record structure, and then fills it with methods by side-effects. As a result, during the evaluation of the methods, the self variable points circularly to the record structure. The execution of method bodies is delayed by prefixing them with dummy abstractions, which are stripped away on method invocation. On method invocation, there is no need to pass the object to the methods explicitly because the self variable already points to the record structure.

This interpretation handles both field update and method update. The treatment of fields is straightforward. Method update is interpreted simply as an assignment to a record field, again binding the self variable circularly. The update of a method can be observed by all siblings of the method because their self variables point to the data structure that is modified.

Nevertheless, the cyclic-record interpretation is not completely satisfactory. In an imperative prototype-based language one expects to have cloning as one of the primitives. Cloning, however, cannot easily be handled by cyclic records because recursion has been tied too soon. The self variables of methods are bound to a fixed structure, and copying the structure does not change those bindings. Hence, after cloning an object, the self variables of the clone keep referring to the old object, instead of referring to the clone.

We do not know how to modify the cyclic-record interpretation to allow cloning. One way to try to solve the problem is to delay the binding of self. Then, however, we are forced back into the self-application interpretation with cloning of Section 18.2.1.

18.2.5 Split Methods

The problem noted with the binding of *init* in the recursive-record interpretation can be eliminated by splitting methods into two parts [9]. Each method l_i is represented by two record components, l_i^{sel} and l_i^{upd} ; the former is used for method invocation, and the latter for method update. For clarity, we rename *init* to *create*. The occurrences of *create* used for update are now encapsulated within the definition of *create*. This solution, it turns out, supports even method update, so we do not distinguish fields from methods:

Untyped split-method interpretation

$[l_i = \zeta(x_i)b_i \text{ } i \in 1..n] \triangleq$	$(l_i \text{ distinct})$
let rec <i>create</i> ($y_i \text{ } i \in 1..n$) =	
$(l_i^{sel} = y_i,$	
$l_i^{upd} = \lambda(y_i')\text{create}(y_j \text{ } j \in 1..i-1, y_i', y_k \text{ } k \in i+1..n) \text{ } i \in 1..n)$	
in <i>create</i> ($\lambda(x_i)b_i \text{ } i \in 1..n$)	

$$\begin{aligned} o.l_j &\triangleq o.l_j^{sel}(o) & (j \in 1..n) \\ o.l_j = \zeta(y)b &\triangleq o.l_j^{upd}(\lambda(y)b) & (j \in 1..n) \end{aligned}$$

The *create* function takes a collection of functions and produces a record. A method l_j is updated by supplying the new code for l_j to the corresponding function l_j^{upd} ; this code is passed on to *create*, which is invoked with a revised collection of functions to produce a modified record. A method l_j is invoked by applying the corresponding function l_j^{sel} to o , much as in the self-application interpretation.

The split-method interpretation matches the primitive semantics of objects. In this respect it is just as good as the self-application interpretation. It is more complex, but as we shall see it has better typing properties.

18.3 Typed Interpretations

In this section we investigate typed interpretations based on record types, function types, recursive types, and existential types. Some of the untyped interpretations do not yield the desired typing and subtyping properties. Some others do, but with considerable complexity. Finally, we show that a typed version of the split-method interpretation is both faithful and uniform. It has an imperative variant that shares these characteristics.

The interpretations rely on record types. Record types have the form $\langle l_i:B_i \mid i \in 1..n \rangle$; they are covariant in their components for functional interpretations, and invariant for imperative interpretations (because of side-effects on record fields).

18.3.1 Self-Application

A typed version of the interpretation of Section 18.2.1 is obtained by representing object types as recursive record types:

Self-application interpretation

$$\begin{aligned} A \equiv [l_i:B_i \mid i \in 1..n] &\triangleq & (l_i \text{ distinct}) \\ \mu(X)\langle l_i:X \rightarrow B_i \mid i \in 1..n \rangle & \\ [l_i=\zeta(x_i:A)b_i \mid i \in 1..n] &\triangleq fold(A,\langle l_i=\lambda(x_i:A)b_i \mid i \in 1..n \rangle) \\ o.l_j &\triangleq unfold(o).l_j(o) & (j \in 1..n) \\ o.l_j = \zeta(y:A)b &\triangleq fold(A,unfold(o).l_j:=\lambda(y:A)b) & (j \in 1..n) \end{aligned}$$

Unfortunately, the subtyping rule for object types fails to hold: note that a contravariant X occurs in all method types. If we ignore this serious problem, it is easy to extend the self-application interpretation to Self types, by setting $Obj(X)[l_i:B_i\{X\} \mid i \in 1..n] \triangleq \mu(X)\langle l_i:X \rightarrow B_i\{X\} \mid i \in 1..n \rangle$. Similarly, the interpretation can be extended with cloning as in Section 18.2.1.

18.3.2 Recursive Records

We can try to obtain an interpretation that validates the subtyping rules for object types by interpreting object types as (non-recursive) record types [39]:

Recursive-record interpretation, without update

$$\begin{aligned}
 A &\equiv [l_i:B_i]_{i \in 1..n} \triangleq && (l_i \text{ distinct}) \\
 &\quad \langle l_i:B_i \rangle_{i \in 1..n} \\
 [l_i=\zeta(x_i:A)b_i\{x_i\}]_{i \in 1..n} &\triangleq \mu(x:A)\langle l_i=b_i[x] \rangle_{i \in 1..n} \\
 o.l_j &\triangleq o.l_j && (j \in 1..n)
 \end{aligned}$$

As we said in Section 18.2.2, a uniform update primitive is precluded by this interpretation. However, it is still possible to achieve the effect of internal updates by the *init* technique. We show this by translating an object calculus with Self types:

Recursive-record interpretation with internal field update

$$\begin{aligned}
 A &\equiv Obj(X)[f_k:B_k]_{k \in 1..m} \mid [l_i:B_i[X^+]]_{i \in 1..n} \triangleq && (f_k, l_i \text{ distinct}) \\
 &\quad \mu(X)\langle f_k:B_k \rangle_{k \in 1..m}, \langle l_i:B_i[X] \rangle_{i \in 1..n} \\
 obj(X=A)[f_k=b_k]_{k \in 1..m} \mid l_i=\zeta(x_i:X)b_i\{X, x_i\} &\triangleq && i \in 1..n \\
 &\quad let\ rec\ init(y_k:B_k) : A = \mu(x:A)fold(A, \langle f_k=y_k \rangle_{k \in 1..m}, \langle l_i=b_i[A, x] \rangle_{i \in 1..n}) \\
 &\quad in\ init(b_k) && \\
 o.f_j &\triangleq unfold(o).f_j && (j \in 1..m) \\
 o.f_j := b &\triangleq init(unfold(o).f_k)_{k \in 1..j-1}, b, unfold(o).f_k && (j \in 1..m) \text{ (internal)} \\
 o.l_j &\triangleq unfold(o).l_j && (j \in 1..n)
 \end{aligned}$$

This refined interpretation validates the subtyping rules for object types because the type variable X occurs only covariantly in $\mu(X)\langle f_k:B_k \rangle_{k \in 1..m}, \langle l_i:B_i[X] \rangle_{i \in 1..n}$.

This interpretation is frequently used to code object-oriented behavior within λ -calculi with records, for specific examples. A typical application concerns movable color points:

$$\begin{aligned}
 CPoint &\triangleq \\
 &\quad Obj(X)[x:Int, c:Color \mid mv:Int \rightarrow X] \\
 CPoint : CPoint &\triangleq \\
 &\quad [x = 0, c = black \mid mv = \zeta(s:CPoint) \lambda(dx:Int) s.x := s.x + dx]
 \end{aligned}$$

The translation is:

$$\begin{aligned}
 CPoint &\triangleq \\
 &\quad \mu(X)\langle x:Int, c:Color, mv:Int \rightarrow X \rangle
 \end{aligned}$$

```

cPoint : CPoint  ≡
let rec init(x0:Int, c0:Color):CPoint =
  μ(s:CPoint) fold(CPoint,
    {x = x0, c = c0,
     mv = λ(dx:Int) init(unfold(s).x+dx, unfold(s).c)))
  in init(0, black)

```

This translation achieves the desired effect, yielding the expected behavior for *cPoint* and the expected subtypings for *CPoint*. In this example, the correct binding of *init* does not pose any problem. In contrast, if the code for *mv* had been $\lambda(dx:\text{Int}) f(s).x := s.x + dx$, where *f* is a function of the appropriate type, it would not have been clear how to give the translation.

18.3.3 State-Application

As a first attempt at typing the state-application interpretation of Section 18.2.3, we can set:

$$\begin{aligned} [f_k:B_k^{k \in 1..m} \mid l_i:B_i^{i \in 1..n}] &\triangleq \\ (st:\langle f_k:B_k^{k \in 1..m} \rangle, mt:\langle l_i:\langle f_k:B_k^{k \in 1..m} \rangle \rightarrow B_i^{i \in 1..n} \rangle) \end{aligned}$$

Unfortunately there is a subtyping problem here because of the contravariance of the method argument types, just as in the self-application interpretation.

A better solution can be found via existential types (see Section 13.2 for notation). Interestingly, recursive types are not needed. The resulting interpretation handles object calculi with Self types that do not support method update. A full description of this technique is, however, not straightforward. We sketch the main idea here, and we refer to [71, 72, 103] for details.

The state of an object, represented by a collection of fields *st*, is hidden from view by existential abstraction, hence external updates are not possible. The troublesome method argument types are hidden as well, so this interpretation yields the desired subtypings.

State-application interpretation

$$\begin{aligned}
A \equiv Obj(X)[l_i:B_i[X^+]^{i \in 1..n}] &\triangleq && (l_i \text{ distinct}) \\
\exists(X)C\{X\} & \\
\text{where } C\{X\} \equiv (st:X, mt:\langle l_i:X \rightarrow B_i[X^+]^{i \in 1..n} \rangle) & \\
obj(X=A)[f_k=b_k^{k \in 1..m} \mid l_i=\varsigma(x_i:X)b_i\{x_i\}^{i \in 1..n}] &\triangleq && (f_k, l_i \text{ distinct}) \\
(\text{pack } X=\langle f_k:B_k^{k \in 1..m} \rangle) \\
\text{with } (st=\langle f_k=b_k^{k \in 1..m} \rangle, \\
mt=\mu(m:\langle l_i:X \rightarrow B_i^{i \in 1..n} \rangle)\langle l_i=\lambda(s:X)b_i^{i \in 1..n} \rangle) && & (\text{for appropriate } b_i') \\
:C\{X\}) & \\
x_i.f_j &\triangleq s.f_j && (j \in 1..m) \text{ (internal)}
\end{aligned}$$

$x_i.f_j := b \triangleq s.f_j := b$	$(j \in 1..m)$ (internal)
$x_i.l_j \triangleq m.l_j(s)$	$(j \in 1..n)$ (internal)
$o_A.l_j \triangleq \text{open } o \text{ as } X, p:C\{X\} \text{ in } p.mt.l_j(p.st) : B_j$	$(j \in 1..n)$ (external) for $X \notin FV(B_j)$
$o_A.l_j \triangleq$ $\text{open } o \text{ as } X, p:C\{X\}$ $\text{in } (\text{pack } X' = X \text{ with } (st=p.mt.l_j(p.st), mt=p.mt) : C\{X'\})$ $: A$	$(j \in 1..n)$ (external) for $B_j \equiv X$

The translation of internal operations on self is problematic, as we observed earlier. External method invocations also have to be handled carefully; in the table we deal with method invocation for the cases where $X \notin FV(B_j)$ and where $B_j \equiv X$. Other possibilities arise; the general case requires code generation driven by the structure of the object types [72].

The difficulties with this typed interpretation can be resolved, and then both invocation and internal field update work well, yielding encodings of class-based languages. However, there are no visible advantages of this existential interpretation with respect to the recursive-record interpretation given at the end of Section 18.3.2; even the hiding of fields can be obtained there by subtyping. The similarity is not completely accidental; these two interpretations can be formulated as special cases of a single scheme [72].

18.3.4 Cyclic Records

For the typed version of the cyclic-record interpretation, we write $nil(C)$ for the “uninitialized” value of type C :

Cyclic-record interpretation

$A \equiv [f_k:B_k^{k \in 1..m} \mid l_i:B_i^{i \in 1..n}] \triangleq$ $\langle f_k:B_k^{k \in 1..m}, l_i:\langle\rangle \rightarrow B_i^{i \in 1..n} \rangle$	$(l_i \text{ distinct})$
$[f_k=b_k^{k \in 1..m} \mid l_i=\zeta(x_i:A)b_i[x_i]^{i \in 1..n}] \triangleq$ $\text{let } x:A = \langle f_k=b_k^{k \in 1..m}, l_i=nil(\langle\rangle \rightarrow B_i)^{i \in 1..n} \rangle$ $\text{in } x \cdot l_i := \lambda(z:\langle\rangle)b_i[\{x\}]^{i \in 1..n}; x$	$z \notin FV(b_i^{i \in 1..n}) \cup \{x\}$
$o \cdot f_j \triangleq o \cdot f_j$	$(j \in 1..m)$
$o \cdot f_j := b \triangleq o \cdot f_j := b$	$(j \in 1..m)$
$o \cdot l_j \triangleq o \cdot l_j(\langle\rangle)$	$(j \in 1..n)$
$o \cdot l_j = \zeta(y:A)b[y] \triangleq$ $\text{let } x:A = o \text{ in } x \cdot l_j := \lambda(z:\langle\rangle)b[\{x\}]$	$(j \in 1..n)$ $z \notin FV(b) \cup \{x\}$

This interpretation has the expected behavior in an imperative context. Moreover, the self parameters are not reflected in the types of objects; thus the desired subtyping properties are obtained.

Cyclic records can be exploited further to obtain typed encodings of class-based imperative languages [60]. However, as discussed earlier, it does not seem possible to extend this interpretation to cloning, short of turning it into a self-application interpretation. But, if we do that, we run into the typing difficulties of self-application.

18.3.5 Split Methods

Building on the ideas described so far, we introduce interpretations that we find almost completely satisfactory. These are typed versions of the split-method interpretation.

A first idea for typing the code of the split-method interpretation could be to set:

$$[l_i : B_i^{i \in 1..n}] \triangleq \mu(X)(l_i^{sel} : X \rightarrow B_i^{i \in 1..n}, l_i^{upd} : (X \rightarrow B_i) \rightarrow X^{i \in 1..n})$$

but the body of this recursive type contains contravariant occurrences of X and therefore this definition does not yield the desired subtypings.

A second idea, due to Viswanathan [123], is to consider a similar type without the contravariance occurrences of X in the types of the l_i^{sel} components:

$$[l_i : B_i^{i \in 1..n}] \triangleq \mu(X)(l_i^{sel} : B_i^{i \in 1..n}, l_i^{upd} : (X \rightarrow B_i) \rightarrow X^{i \in 1..n})$$

We can adapt the untyped split-method interpretation to match this encoding of object types:

Split-method interpretation (initial version)

$A \equiv [l_i : B_i^{i \in 1..n}] \triangleq$	$(l_i \text{ distinct})$
$\mu(X)(l_i^{sel} : B_i^{i \in 1..n}, l_i^{upd} : (X \rightarrow B_i) \rightarrow X^{i \in 1..n})$	
$[l_i = \varsigma(x_i : A)b_i^{i \in 1..n}] \triangleq$	
let rec create($y_i : A \rightarrow B_i^{i \in 1..n}$) : $A =$	
fold(A ,	
$\langle l_i^{sel} = y_i(create(y_j^{j \in 1..n}))^{i \in 1..n},$	
$l_i^{upd} = \lambda(y_i' : A \rightarrow B_i).create(y_j^{j \in 1..i-1}, y_i', y_k^{k \in i+1..n})^{i \in 1..n} \rangle$	
in $create(\lambda(x_i : A)b_i^{i \in 1..n})$	
$o.A.l_j \triangleq unfold(o).l_j^{sel}$	$(j \in 1..n)$
$o.l_j = \varsigma(y : A)b_j \triangleq unfold(o).l_j^{upd}(\lambda(y : A)b_j)$	$(j \in 1..n)$

Note that method invocation is no longer modeled by self-application. Instead, a kind of self-application happens whenever an object is constructed: $l_i^{sel} = y_i(create(y_j^{j \in 1..n}))$ applies the function y_i to a freshly created copy of self.

This interpretation validates the subtyping rules for object types, since all occurrences of X , bound by μ , are covariant; and it also validates the typing rules for objects. Moreover, it behaves as expected with functional semantics. (It is written for call-by-

name evaluation, but it can be easily adapted for call-by-value evaluation.) In summary, it respects both the type rules and the primitive semantics of simple objects.

However, this interpretation is essentially incompatible with imperative semantics, because every method invocation creates a fresh object instead of reusing the current one. Furthermore, it does not extend to object types with Self because any occurrence of Self in a type B_i appears contravariantly within the type of the corresponding component l_i^{upd} . For these reasons, we go on to consider a more sophisticated version of the split-method interpretation.

A third idea is to go back to our initial type for l_i^{sel} and to use the Self-quantifier technique for obtaining covariance:

$$[l_i : B_i \text{ } i \in 1..n] \triangleq \mu(Y) \exists(X <: Y) (l_i^{sel} : X \rightarrow B_i \text{ } i \in 1..n, l_i^{upd} : (X \rightarrow B_i) \rightarrow X \text{ } i \in 1..n)$$

This encoding validates the subtyping rules for object types because all the occurrences of X , bound by \exists , are covariant. Unfortunately, with this encoding it is impossible to perform method invocations: after opening the existential type we do not have an appropriate argument of the abstract type X to which to apply the l_i^{sel} components. Since that argument should be the object itself, we can solve this problem by adding a record component, r , that is bound recursively to the whole record; this record component is analogous to the *recoup* method of Section 15.6.

This solution involves a simple change in the untyped interpretation and yields the following typed interpretation [9]:

Split-method interpretation

$$A \equiv [l_i : B_i \text{ } i \in 1..n] \triangleq \quad \quad \quad (l_i \text{ distinct})$$

$$\mu(Y) \exists(X <: Y) C\{X\}$$

$$\text{where } C\{X\} \equiv \langle r : X, l_i^{sel} : X \rightarrow B_i \text{ } i \in 1..n, l_i^{upd} : (X \rightarrow B_i) \rightarrow X \text{ } i \in 1..n \rangle$$

$$[l_i = \zeta(x_i : A) b_i \text{ } i \in 1..n] \triangleq$$

$$\text{let rec create}(y_i : A \rightarrow B_i \text{ } i \in 1..n) : A =$$

fold(A ,

pack $X = A$

with

$$\quad \quad \quad (r = \text{create}(y_i \text{ } i \in 1..n)),$$

$$\quad \quad \quad l_i^{sel} = y_i \text{ } i \in 1..n,$$

$$\quad \quad \quad l_i^{upd} = \lambda(y_i' : A \rightarrow B_i) \text{create}(y_j \text{ } j \in 1..i-1, y_i', y_k \text{ } k \in i+1..n) \text{ } i \in 1..n \rangle$$

: $C\{X\}$)

$$\text{in create}(\lambda(x_i : A) b_i \text{ } i \in 1..n)$$

$$o.A.l_j \triangleq$$

$(j \in 1..n)$

$$\text{open unfold}(o) \text{ as } X <: A, p : C\{X\} \text{ in } p.l_j^{sel}(p.r) : B_j$$

$$o.l_j = \zeta(y : A) b \triangleq$$

$(j \in 1..n)$

$$\text{open unfold}(o) \text{ as } X <: A, p : C\{X\} \text{ in } p.l_j^{upd}(\lambda(y : A) b) : A$$

With this interpretation, we obtain both the expected semantics and the expected subtyping properties. This interpretation validates all the typing, subtyping, and reduction rules for simple objects, up to Chapter 8. It also validates the equational rules except for (Eq Sub Object), which is not derivable but is consistent. Moreover, it can be extended in a number of ways, as we show next.

Variance annotations can be added with little extra effort. An invariant attribute, l^0 , is translated to two record components l^{sel} and l^{upd} as above; a covariant one, l^+ , is translated to just the l^{sel} component; a contravariant one, l^- , is translated to just the l^{upd} component. This extended interpretation validates the subtyping rules for variance annotations.

Since the interpretation uses the Self-quantifier idea, it is natural to extend it to Self types, as follows:

Split-method interpretation (with Self types)

$A \equiv Obj(X)[l_i:B_i\{X\}^{i \in 1..n}] \triangleq$	$(l_i \text{ distinct})$
$\mu(Y)\exists(X <: Y)C\{X\}$	
where $C\{X\} \equiv \langle r:X, l_i^{sel}:X \rightarrow B_i\{X\}^{i \in 1..n}, l_i^{upd}:(X \rightarrow B_i\{X\}) \rightarrow X^{i \in 1..n} \rangle$	
$obj(X=A)[l_i=\zeta(x_i:X)b_i^{i \in 1..n}] \triangleq$	
<i>let rec create(y_i:A → B_i{A})^{i ∈ 1..n}:A =</i>	
<i>fold(A,</i>	
<i>pack X=A</i>	
<i>with</i>	
$\langle r=create(y_j^{i \in 1..n}),$	
$l_i^{sel}=y_j^{i \in 1..n},$	
$l_i^{upd}=\lambda(y_{i'}:A \rightarrow B_i\{A\})create(y_j^{j \in 1..i-1}, y_{i'}, y_k^{k \in i+1..n})^{i \in 1..n}$	
$: C\{X\})$	
<i>in create(λ(x_i:A)b_i^{i ∈ 1..n})</i>	
$o_A.l_j \triangleq$	$(j \in 1..n)$
<i>open unfold(o) as X <: A, p:C\{X\} in p.l_j^{sel}(p.r) : B_j\{A\}</i>	
$o.l_j \triangleq (X <: A, y:X)\zeta(x:X)b\{y\} \triangleq$	$(j \in 1..n)$
<i>open unfold(o) as X <: A, p:C\{X\} in p.l_j^{upd}(\lambda(x:X)b\{p.r\}) : A</i>	

As an example, we translate the following type and term:

$$\begin{aligned}
 Cell &\triangleq \\
 &\quad Obj(X)[get:Nat, set:Nat \rightarrow X] \\
 cell : Cell &\triangleq \\
 &\quad obj(X=Cell)[get = \zeta(s:X) 0, set = \zeta(s:X) \lambda(n:Nat) s.get \triangleq \zeta(s':X) n]
 \end{aligned}$$

The translations are:

$Cell \equiv \mu(Y) \exists(X <: Y) C\{X\}$
 where $C\{X\} \equiv \langle r:X, get^{sel}:X \rightarrow Nat, set^{sel}:X \rightarrow (Nat \rightarrow X),$
 $get^{upd}:(X \rightarrow Nat) \rightarrow X, set^{upd}:(X \rightarrow (Nat \rightarrow X)) \rightarrow X \rangle$

$cell \equiv$
 $\text{let rec } create(\text{get:Cell} \rightarrow \text{Nat}, \text{set:Cell} \rightarrow (\text{Nat} \rightarrow \text{Cell})):\text{Cell} =$
 $\quad \text{fold}(Cell,$
 $\quad \quad \text{pack } X = Cell$
 $\quad \quad \text{with}$
 $\quad \quad \quad \langle r = create(\text{get}, \text{set}),$
 $\quad \quad \quad get^{sel} = \text{get},$
 $\quad \quad \quad set^{sel} = \text{set},$
 $\quad \quad \quad get^{upd} = \lambda(\text{get}' : \text{Cell} \rightarrow \text{Nat}) \text{ create}(\text{get}', \text{set}),$
 $\quad \quad \quad set^{upd} = \lambda(\text{set}' : \text{Cell} \rightarrow (\text{Nat} \rightarrow \text{Cell})) \text{ create}(\text{get}, \text{set}')$
 $\quad \quad \quad : C\{X\}\rangle$
 $\quad \quad \text{in } create(\lambda(s:Cell) 0,$
 $\quad \quad \quad \lambda(s:Cell) \lambda(n:Nat)$
 $\quad \quad \quad \text{open unfold}(s) \text{ as } X <: \text{Cell}, y: \dots, get^{upd}:(X \rightarrow \text{Nat}) \rightarrow X\rangle$
 $\quad \quad \quad \text{in } y.get^{upd}(\lambda(s':X) n)$
 $\quad \quad \quad : \text{Cell}\rangle$

As we can see from this example, the split-method interpretation may be adequate for representing objects as records, but is overwhelming as a programming technique.

The split-method interpretation handles an interesting source calculus that resembles the one of Chapter 16 (especially when variance annotations are added). As given, the interpretation does not support structural rules, which are desirable in conjunction with Self types. However, we believe that those rules can be accounted for by introducing structural rules in the target calculus.

18.3.6 From Split Methods back to Self-Application, Imperatively

When we consider an imperative version of the split-method interpretation, we discover that it is no longer necessary to split methods, because updates can be handled imperatively [9]. The remaining structure of the interpretation is still interesting; judging by the treatment of method invocation, it amounts to a self-application technique. The main technical novelties are the repacking performed in the update operations and the treatment of cloning.

Imperative self-application interpretation

$$A \equiv [f_k:B_k^{k \in 1..m} | l_i:B_i^{i \in 1..n}] \triangleq \mu(Y) \exists(X <: Y) C\{X\}$$

(f_k, l_i distinct)

with $C\{X\} \equiv \langle r:X, f_k:B_k^{k \in 1..m}, l_i:X \rightarrow B_i^{i \in 1..n}, cl:\langle \rangle \rightarrow X \rangle$

$$\begin{aligned}
& [f_k = b_k^{k \in 1..m} \mid l_i = \zeta(x_i : A) b_i^{i \in 1..n}] \triangleq \\
& \text{let rec } \text{create}(y_k : B_k^{k \in 1..m}, y_i : A \rightarrow B_i^{i \in 1..n}) : A = \\
& \quad \text{let } z : C\{A\} = \langle r = \text{nil}(A), f_k = y_k^{k \in 1..m}, l_i = y_i^{i \in 1..n}, cl = \text{nil}(\langle \rangle \rightarrow A) \rangle \\
& \quad \text{in } z.r := \text{fold}(A, \text{pack } X <: A = A \text{ with } z : C\{X\}); \\
& \quad z.cl := \lambda(x : \langle \rangle) \text{create}(z.f_k^{k \in 1..m}, z.l_i^{i \in 1..n}); \\
& \quad z.r \\
& \quad \text{in } \text{create}(b_k^{k \in 1..m}, \lambda(x_i : A) b_i^{i \in 1..n}) \\
o_{A \circ f_j} & \triangleq \text{open unfold}(o) \text{ as } X <: A, p : C\{X\} \text{ in } p.f_j : B_j \quad (j \in 1..m) \\
o_{A \circ f_j} := b & \triangleq \\
& \quad \text{open unfold}(o) \text{ as } X <: A, p : C\{X\} \\
& \quad \text{in } \text{fold}(A, \text{pack } X' <: X = X \text{ with } p.f_j := b : C\{X'\}) : A \\
o.A.l_j & \triangleq \text{open unfold}(o) \text{ as } X <: A, p : C\{X\} \text{ in } p.l_j(p.r) : B_j \quad (j \in 1..n) \\
o.l_j \neq \zeta(x : A) b & \triangleq \\
& \quad \text{open unfold}(o) \text{ as } X <: A, p : C\{X\} \\
& \quad \text{in } \text{fold}(A, \text{pack } X' <: X = X \text{ with } p.l_j := \lambda(x : A) b : C\{X'\}) : A \\
\text{clone}(o_A) & \triangleq \text{open unfold}(o) \text{ as } X <: A, p : C\{X\} \text{ in } p.cl(\langle \rangle) : A \quad (j \in 1..n)
\end{aligned}$$

As presented here, the interpretation is for a first-order imperative calculus with cloning (the first-order version of the calculus treated in Section 18.2.1). Much as in the split-method interpretation, we could also handle Self types.

18.3.7 Discussion

In sum, we have described a few techniques for translating typed object calculi into typed calculi without objects. Some of the techniques deal with rich object calculi that include, for example, method update, cloning, and (to some extent) Self types. However, there is not yet a simple, definitive interpretation that once and for all explicates the meaning of objects.

Given the difficulty in finding a definitive interpretation, object calculi provide a useful abstract view of objects and a basis for understanding object-oriented languages, independently of particular encoding techniques.

19 A SECOND-ORDER LANGUAGE

In this chapter we apply our second-order calculi to the study of a simple but expressive programming language. This language, called O-2, is similar to the language O-1 considered at the end of Part I. Like O-1, O-2 includes both class-based and object-based constructs.

O-2 illustrates the new possibilities that come with second-order typing and with Self types. Essentially, O-2 provides satisfactory subtyping for objects with methods that return self, and inheritance for those methods. It is more sophisticated than O-1 in these respects.

Much as we did for O-1, we translate O-2 into an object calculus. In this case the object calculus is S_V (Chapter 16), which features primitive covariant Self types and bounded universal quantifiers. The translation reduces the rules for O-2, which are fairly elaborate, to the simpler rules of the object calculus.

19.1 Features

O-2 preserves most of the features of O-1 and introduces a few new ones. At first sight O-2 may appear almost identical to O-1, but there are significant differences in their typing rules. We have adopted similar notations for O-1 and O-2 in order to emphasize how the “same” constructs may have subtly different meanings in different object-oriented languages.

Like O-1, O-2 includes both class-based and object-based features. Objects have embedded methods, which can be updated. Subclasses can be created by single inheritance; methods of superclasses may be overridden and specialized. Interfaces are separate from implementations, and inheritance is separate from subtyping. Subtyping supports subsumption.

The main new feature of O-2, with respect to O-1, is Self types. In O-1, the variable X of an object type $\text{Object}(X)[l_1; \dots; l_n; B_i[X]_{i \in 1..n}]$ is a recursion variable; our interpretation of O-1 maps $\text{Object}(X)[l_1; \dots; l_n; B_i[X]_{i \in 1..n}]$ to $\mu(X)[l_1; \dots; l_n; B_i[X]_{i \in 1..n}]$. In contrast, in O-2, the variable X is the Self type, in the sense of Chapters 16 and 17; our interpretation of O-2 maps $\text{Object}(X)[l_1; \dots; l_n; B_i[X]_{i \in 1..n}]$ to $\text{Obj}(X)[l_1; \dots; l_n; B_i[X]_{i \in 1..n}]$.

Because of Self types, some subtypings that do not hold in O-1 do hold in O-2, for example $\text{Object}(X)[l_1:X, l_2:Int] <: \text{Object}(Y)[l_1:Y]$. On the other hand, O-2 is not a proper extension of O-1. For example, $\text{Object}(X)[l:X \rightarrow Int]$ is a legal type in O-1 but not in O-2, because contravariant occurrences of Self types are not permitted. In principle, we could have mixed recursive types and Self types in order to make O-2 a proper extension of O-1, but this combination could have been confusing.

As the meaning of object types changes, so does the meaning of class types. In O-2, a class for $\text{Object}(X)[l_i; v_i; B_i[X]^{i \in 1..n}]$ has a method l_i that maps a self parameter of type X to a result of type $B_i[X]$, for an arbitrary subtype X of the object type A . Thus, the method must be parametric in Self. The same parametricity requirement applies to the code used for overriding a method of an object of type $\text{Object}(X)[l_i; v_i; B_i[X]^{i \in 1..n}]$ from the outside.

Because of the new parametricity requirement, it becomes harder to write the code of methods. For example, in O-1, the method l of a class for $\text{Object}(X)[l:X]$ may return a fixed object of type $\text{Object}(X)[l:X]$, which is easy to construct. Instead, in O-2, the method must return a value of type X for an unknown $X <: \text{Object}(X)[l:X]$; this value can be constructed only using the self parameter. The difficulty in defining methods is compensated by an increase in reusability. In particular, it is simple to inherit methods that return values of type Self, because these methods are parametric in Self.

In addition to Self types, O-2 introduces bounded universal types in support of polymorphism. The technical concepts needed for this are already present because of the parametricity requirement on methods, and hence the inclusion of bounded polymorphism is straightforward.

The introduction of Self types and polymorphism removes the need for many uses of typecase. For simplicity, we do not include typecase in O-2. As the examples of this chapter show, however, typecase would still be useful in some circumstances, in particular for handling binary methods.

19.2 Syntax

The following table gives the syntax of O-2 types; these are similar to O-1 types, with the addition of bounded universal types:

Syntax of O-2 types

$A, B ::=$	types
X	type variable
Top	the biggest type
$\text{Object}(X)[l_i; v_i; B_i^{i \in 1..n}]$	object type (each B_i covariant in X)
$\text{Class}(A)$	class type
$\text{All}(X <: A) B$	bounded universal type

As in O-1, we omit basic types (such *Int*) and function types, but we use them in examples. **Top** is the biggest type. An object type $\text{Object}(X)[l_i; v_i; B_i[X]^{i \in 1..n}]$ has as a Self type X , labels l_i , variance annotations v_i ($,$, \circ , or $+$), and attribute types $B_i[X]$ where X occurs only covariantly in each $B_i[X]$. A class type $\text{Class}(A)$ is a type for classes generating objects of type A , where A is an object type. A bounded universal type $\text{All}(X <: A) B$ is the type of an abstraction that for any subtype A' of A produces a result of type $B[A']$.

The notion of covariance for these types is analogous to that defined in Chapter 16. We omit a definition, and note only that $\text{Class}(A)$ is invariant in all the type variables that occur free in A .

The syntax of terms is given next. Most constructs are similar to the ones of O-1, and have roughly the same meaning. One of the syntactic differences is in the variable binders for **object**, **method**, and **subclass**, where $x:X<:A$ is short for $x:X$ and $X<:A$, and $x:X=A$ is short for $x:X$ and $X=A$. Once again, the syntax could be interpreted functionally or imperatively.

Syntax of O-2 terms

$a, b, c ::=$	terms
x	variable
object ($x:X=A$) $l_i=b_i \ i \in 1..n$ end	direct object construction
$a.l$	field selection / method invocation
$a.l := b$	update with a term
$a.l := \text{method}(x:X<:A) b \text{ end}$	update with a method
new c	object construction from a class
root	root class
subclass of $c:C$ with ($x:X<:A$)	subclass
$l_i=b_i \ i \in n+1..n+m$	additional attributes
override $l_i=b_i \ i \in \text{Over} \subseteq 1..n$ end	overridden attributes
$c^l(A,a)$	class selection
fun ($X<:A$) $b \text{ end}$	type abstraction
$b(A)$	type application

An object can be written as **object**($x:X=A$) $l_i=b_i\{X,x\} \ i \in 1..n$ **end**, where x is the self variable for all the attributes. The binding $X=A$ is just a convenience; it gives a name X to the Self type A within the object. The object type A must be known (must not be a type variable). For an imperative semantics, it is important to distinguish fields from proper methods. As in O-1, we keep the syntax simple by assuming an unspecified syntactic criterion for this distinction; fields should not contain occurrences of the self variable.

Methods that are used for update must be parametric in Self; thus the method update construct has a binder $x:X<:A$, where x is self and X is the Self type. For simplicity, we do not include a parameter representing the old self (Section 16.2.2).

In the subclass construct, the header $c:C$ **with**($x:X<:A$) identifies a superclass c , its class type C (having, say, n components), and the type A of the objects generated by the subclass (having $n+m$ components). The binder introduces a self variable x , and a Self type variable X , with the assumption that X is a subtype of A . Both the object type A and the class type C must be known (must not be type variables).

The class selection construct $c^l(A,a)$ extracts the method named l from c and applies it to the type A and the term a . The type A must be a subtype of the object type

corresponding to c , and a must have type A . As in O-1, class selection provides the functionality of super; in O-2, an occurrence of $c^l(X,x)$ as part of the code for a subclass of c with self variable x and with Self type X corresponds to a use of super.

Bounded polymorphic abstraction and application are the standard constructs of Chapter 13.

As in O-1, we define abbreviations for the **Root** class type, for direct subclasses of **root**, for **super**, and for object-based inheritance:

```

Root  $\triangleq$ 
  Class(Object( $X$ )[])
class with( $x:X<:A$ )  $l_i=b_i$  $i \in 1..n$  end  $\triangleq$ 
  subclass of root:Root with( $x:X<:A$ )  $l_i=b_i$  $i \in 1..n$  override end
subclass of  $c:C$  with( $x:X<:A$ ) ... super.l ... end  $\triangleq$ 
  subclass of  $c:C$  with( $x:X<:A$ ) ...  $c^l(X,x)$  ... end
object( $x:X=A$ ) ...  $l$  copied from  $c$  ... end  $\triangleq$ 
  object( $x:X=A$ ) ...  $l=c^l(X,x)$  ... end

```

19.3 Examples

In this section we revisit some of our examples. These examples illustrate the features of O-2, and its main differences with O-1.

19.3.1 Points

We start by rewriting the O-1 examples of Section 12.3, emphasizing the differences. Those examples concern points and color points, with eq and mv methods. Because of the covariance restriction, we can no longer have the binary method eq with type $X \rightarrow Bool$. Instead, we use an auxiliary type $Coord$ that carries sufficient information for some implementations of eq .

```

Coord  $\triangleq$  Object( $X$ )[ $x: Int$ ]
Point  $\triangleq$  Object( $X$ )[ $x: Int$ ,  $eq^+: Coord \rightarrow Bool$ ,  $mv^+: Int \rightarrow X$ ]
CPoint  $\triangleq$  Object( $X$ )[ $x: Int$ ,  $c: Color$ ,  $eq^+: Coord \rightarrow Bool$ ,  $mv^+: Int \rightarrow X$ ]

```

The type of mv in $Point$ is still $Int \rightarrow X$, but this now has a stronger meaning. The result type variable X does not stand for $Point$ as in O-1, but for the Self type. Because of subsumption, the “true type” that Self represents may be any subtype of $Point$. By the subtyping rules (detailed in Section 19.4), we obtain $CPoint <: Point$.

Corresponding to these two types, we define the classes:

```

pointClass : Class(Point)  $\triangleq$ 
  class with (self:  $X <: Point$ )
     $x = 0$ ,

```

```

 $eq = \text{fun}(other: Coord) self.x = other.x \text{ end},$ 
 $mv = \text{fun}(dx: Int) self.x := self.x + dx \text{ end}$ 
end

cPointClass : Class(CPoint)  $\triangleq$ 
  subclass of pointClass: Class(Point)
  with (self: X<:CPoint)
    c = black
  override
    mv = fun(dx: Int) super.mv(dx).c := red end
end

```

The subclass *cPointClass* overrides the method *mv*, but this override is optional. The method *mv* could just as well be inherited, because in both classes it is required to return a result of the Self type. In *cPointClass*, the method *mv* invokes *super* and then modifies the result, updating its color to *red*. There *super.mv* means *pointClass^mv(X, self)*, which has type *X*; the update to *red* preserves this type.

The method *eq* from *pointClass* is simply inherited into *cPointClass*, and therefore it ignores color information. To override *eq* with a version that tests the color of *self* and *other*, we would need *typecase*, as in Chapter 12.

We can use *cPointClass* to create color points of type *CPoint*:

```
cPoint : CPoint  $\triangleq$  new cPointClass
```

Thanks to Self types, calls to *mv* preserve the color information without need of *typecase* either in the definition of *mv* or in its use:

```
movedColor : Color  $\triangleq$  cPoint.mv(1).c
```

19.3.2 Cells

As a further illustration of the subclass mechanisms of O-2, we recode the cell examples of Section 15.2. The O-2 code for these examples is not always straightforward, but it is as direct as we might expect; it is simpler than the corresponding terms of Section 15.2. In addition, there are no uses of *typecase*.

```

Cell  $\triangleq$ 
  Object(X)[contents: Nat, get: Nat, set: Nat  $\rightarrow$  X]

cellClass : Class(Cell)  $\triangleq$ 
  class with(self: X<:Cell)
    contents = 0,
    get = self.contents,
    set = fun(n: Nat) self.contents := n end
  end

```

```

ReCell  $\triangleq$ 
Object(X)[contents: Nat, get: Nat, set: Nat $\rightarrow$ X, backup: Nat, restore: X]

reCellClass : Class(ReCell)  $\triangleq$ 
  subclass of cellClass: Class(Cell)
  with(self: X $<:$ ReCell)
    backup = 0,
    restore = self.contents := self.backup
  override
    set = fun(n: Nat) cellClass $\wedge$ set(X, self.backup := self.contents)(n) end
  end

ReCell'  $\triangleq$ 
Object(X)[contents: Nat, get: Nat, set: Nat $\rightarrow$ X, restore: X]

reCellClass' : Class(ReCell')  $\triangleq$ 
  subclass of cellClass: Class(Cell)
  with(self: X $<:$ ReCell')
    restore = self.contents := 0
  override
    set = fun(n: Nat)
      let m = self.contents
      in cellClass $\wedge$ set(X,
        self.restore := method(y: Y $<:$ X) y.contents := m end)
      (n)
    end
  end
end

```

We have *ReCell* $<:$ *ReCell'* $<:$ *Cell*. In these type definitions, we have not included any variance annotations; it would be sensible to adorn some components with $^+$, in order to protect them against overrides, but this is not necessary for the subtyping *ReCell* $<:$ *ReCell'* $<:$ *Cell*.

Finally, we briefly illustrate the use of polymorphism by writing a generic equality function and a generic doubling function:

```

eqPredicate : All(X $<:$ Cell) X $\rightarrow$ X $\rightarrow$ Bool  $\triangleq$ 
  fun(X $<:$ Cell) fun(self: X) fun(other: X) self.get = other.get end end end

double : All(X $<:$ Cell) X $\rightarrow$ X  $\triangleq$ 
  fun(X $<:$ Cell) fun(x: X) x.set(2 $\ast$ x.get) end end

```

These functions can be applied directly, or they can be embedded in classes for subtypes of *Cell*.

19.4 Typing

The type rules for O-2 are based on the following judgments:

Judgments

$E \vdash \diamond$	environment E is well-formed
$E \vdash A$	A is a well-formed type in E
$E \vdash A <: B$	A is a subtype of B in E
$E \vdash vA <: v'B$	A is a subtype of B in E , with variance annotations v and v'
$E \vdash a : A$	a has type A in E

The rules for environments are standard, and identical to the ones in O-1.

Environments

(Env \emptyset)	(Env $X <:$)	(Env x)
	$E \vdash A \quad X \notin \text{dom}(E)$	$E \vdash A \quad x \notin \text{dom}(E)$
$\emptyset \vdash \diamond$	$E, X <: A \vdash \diamond$	$E, x:A \vdash \diamond$

In the rules for types, the only novelty with respect to O-1 is the covariance requirement for object types. (We omit the definition of covariance. A similar definition is given in Section 16.2; it suffices to add that $\text{Class}(A)$ is invariant in all of the type variables that occur free in A .)

Types

(Type X)	(Type Top)
$E', X <: A, E'' \vdash \diamond$	$E \vdash \diamond$
$E', X <: A, E'' \vdash X$	$E \vdash \text{Top}$
(Type Object) $(l_i \text{ distinct}, v_i \in \{^o, ^-, ^+\})$	(Type Class) $(\text{where } A \equiv \text{Object}(X)[l_i v_i : B_i(X)]^{i \in 1..n})$
$E, X <: \text{Top} \vdash B_i(X^+)$	$E \vdash A$
$E \vdash \text{Object}(X)[l_i v_i : B_i(X)]^{i \in 1..n}$	$E \vdash \text{Class}(A)$

For subtyping, we have:

Subtyping

(Sub Refl)	(Sub Trans)	(Sub X)	(Sub Top)
$E \vdash A$	$E \vdash A <: B \quad E \vdash B <: C$	$E', X <: A, E'' \vdash \diamond$	$E \vdash A$
$E \vdash A <: A$	$E \vdash A <: C$	$E', X <: A, E'' \vdash X <: A$	$E \vdash A <: \text{Top}$

(Sub Object) (where $A \equiv \text{Object}(X)[l_i v_i : B_i(X)]^{i \in 1..n+m}$, $A' \equiv \text{Object}(X')[l_i v_i' : B_i'(X')]^{i \in 1..n}$)

$$\frac{E \vdash A \quad E \vdash A' \quad E, X <: A \vdash v_i B_i(X) <: v_i' B_i'(X) \quad \forall i \in 1..n}{E \vdash A <: A'}$$

(Sub Invariant)

$$E \vdash B$$

$$E \vdash {}^\circ B <: {}^\circ B$$

(Sub Covariant)

$$E \vdash B <: B' \quad v \in \{^o, +\}$$

$$E \vdash v B <: {}^+ B'$$

(Sub Contravariant)

$$E \vdash B' <: B \quad v \in \{^o, -\}$$

$$E \vdash v B <: {}^- B'$$

The rule (Sub Object) is slightly different from the corresponding rule of O-1. The difference is a change in assumption, from $E, X <: A' \vdash v_i B_i(X) <: v_i' B_i'(A')$ to $E, X <: A \vdash v_i B_i(X) <: v_i' B_i'(X)$. This change reflects the change in interpretation from O-1 to O-2: from recursive object types to object types with covariant Self. The difference is important but subtle; it constitutes a prime example of the issues one encounters in giving sound typing rules for object-oriented languages, and of the usefulness of developing a solid foundation for these rules.

The differences between O-2 and O-1 are also evident in the following typing rules for terms:

Terms

(Val Subsumption)

$$\frac{E \vdash a : A \quad E \vdash A <: B}{E \vdash a : B}$$

(Val x)

$$\frac{E', x:A, E'' \vdash \diamond}{E', x:A, E'' \vdash x : A}$$

(Val Object) (where $A \equiv \text{Object}(X)[l_i v_i : B_i(X)]^{i \in 1..n}$)

$$E, x:A \vdash b_i(A) : B_i(A) \quad \forall i \in 1..n$$

$$E \vdash \text{object}(x:X=A) \ l_i = b_i(X)^{i \in 1..n} \ \text{end} : A$$

(Val Select) (where $A \equiv \text{Object}(X)[l_i v_i : B_i(X)]^{i \in 1..n}$)

$$E \vdash a : A' \quad E \vdash A' <: A \quad v_j \in \{^o, +\} \quad j \in 1..n$$

$$E \vdash a.l_j : B_j(A)$$

(Val Update) (where $A \equiv \text{Object}(X)[l_i v_i : B_i(X)]^{i \in 1..n}$)

$$E \vdash a : A' \quad E \vdash A' <: A \quad E \vdash b : B_j \quad X \notin FV(B_j) \quad v_j \in \{^o, -\} \quad j \in 1..n$$

$$E \vdash a.l_j := b : A'$$

(Val Method Update) (where $A \equiv \text{Object}(X)[l_i v_i : B_i(X)]^{i \in 1..n}$)

$$E \vdash a : A' \quad E \vdash A' <: A \quad E, X <: A', x:X \vdash b : B_j(X) \quad v_j \in \{^o, -\} \quad j \in 1..n$$

$$E \vdash a.l_j := \text{method}(x:X <: A') \ b \ \text{end} : A'$$

(Val New)

$$\frac{E \vdash c : \mathbf{Class}(A)}{E \vdash \mathbf{new} c : A}$$

(Val Root)

$$\frac{E \vdash \diamond}{E \vdash \mathbf{root} : \mathbf{Class}(\mathbf{Object}(X)[])}$$

(Val Subclass) (where $A \equiv \mathbf{Object}(X)[l_i v_i; B_i]^{i \in 1..n+m}$, $A' \equiv \mathbf{Object}(X')[l_i v'_i; B'_i]^{i \in 1..n}$, $Ovr \subseteq 1..n$)

$$E \vdash c' : \mathbf{Class}(A') \quad E \vdash A <: A'$$

$$E, X <: A \vdash B'_i <: B_i \quad \forall i \in 1..n - Ovr$$

$$E, X <: A, x : X \vdash b_i : B_i \quad \forall i \in Ovr \cup n+1..n+m$$

$$\frac{E \vdash \mathbf{subclass\ of\ } c' : \mathbf{Class}(A') \text{ with } (x : X <: A) \ l_i = b_i \ l_i \in n+1..n+m \text{ override } l_i = b_i \ l_i \in Ovr \text{ end}}{\text{ : Class}(A)}$$

(Val Class Select) (where $A \equiv \mathbf{Object}(X)[l_i v_i; B_i[X]^{i \in 1..n}]$)

$$E \vdash a : A' \quad E \vdash A' <: A \quad E \vdash c : \mathbf{Class}(A) \quad j \in 1..n$$

$$E \vdash c \wedge l_j(A', a) : B_j[A']$$

(Val Fun<:)

$$E, X <: A \vdash b : B$$

(Val Appl<:)

$$E \vdash b : \mathbf{All}(X <: A)B[X]$$

$$E \vdash A' <: A$$

$$E \vdash \mathbf{fun}(X <: A) b \mathbf{end} : \mathbf{All}(X <: A)B$$

$$E \vdash b(A') : B[A']$$

The rule for constructing a single object, (Val Object), has no parametricity requirement. The object is built with knowledge of the Self type (A), and its methods need work only with a self argument of this type.

The rules (Val Select), (Val Update), and (Val Method Update) are all structural rules (see Chapter 16). The rule for field update, (Val Update), guarantees that a new field is vacuously parametric in Self, by requiring that the Self variable does not occur in the field type. The rule for method update, (Val Method Update), requires a new method to be parametric in Self. The rule for selection, (Val Select), exploits this parametricity.

Similarly, the rule for building subclasses, (Val Subclass), requires new methods to be parametric in Self (through the assumption $E, X <: A, x : X \vdash b_i : B_i$). The rule for class selection, (Val Class Select), exploits this parametricity.

In the rule (Val Subclass), the assumption $E \vdash A <: A'$ requires that the object type associated with a subclass be a subtype of the object type associated with its superclass. The subtyping rules for variance annotations come into play. For invariant components (often fields) the component type in A must be the same as the corresponding component type in A' . For covariant components (often proper methods) the component type

in A can be a subtype of the corresponding component type in A' . Thus, in general, there is no type specialization for fields, but there may be type specialization for methods; for methods, there is Self type specialization.

As in O-1, not all methods of a class for A' can be inherited in a class for A . Method l_i is inheritable if $E, X <: A \vdash B'_i <: B_i$ where B'_i and B_i are the result types in A' and A , respectively. This condition holds when B'_i and B_i are identical, for example when they are both equal to the Self type variable X ; therefore methods that return self are inheritable. In this respect O-2 is more sophisticated and liberal than O-1.

19.5 Translation

We relate O-2 to our calculi via a translation, proceeding much as for O-1. We show a translation into the functional calculus S_V of Chapter 16; a similar translation could be given into the imperative calculus of Chapter 17. We write $\langle\!\langle A \rangle\!\rangle$ for the translation of the type A , and $\langle\!\langle a \rangle\!\rangle$ for the translation of the term a .

As announced, the translation of the object type $\text{Object}(X)[l_i; v_i; B_i \ i \in 1..n]$ is $Obj(X)[l_i; v_i; \langle\!\langle B_i \rangle\!\rangle \ i \in 1..n]$. The translation of a class type is less straightforward; using the ideas of Section 16.6, we map a class type $\text{Class}(\text{Object}(X)[l_i; v_i; B_i \ i \in 1..n])$ to an object type that contains components for pre-methods, and now the pre-methods have bounded universal types. We assume that the label *new* does not occur in the source language.

Translation of O-2 types

$$\langle\!\langle X \rangle\!\rangle \triangleq X$$

$$\langle\!\langle \text{Top} \rangle\!\rangle \triangleq \text{Top}$$

$$\langle\!\langle \text{Object}(X)[l_i; v_i; B_i \ i \in 1..n] \rangle\!\rangle \triangleq Obj(X)[l_i; v_i; \langle\!\langle B_i \rangle\!\rangle \ i \in 1..n]$$

$$\langle\!\langle \text{Class}(A) \rangle\!\rangle \triangleq Obj(Z)[\text{new}^+ \cdot \langle\!\langle A \rangle\!\rangle, l_i^+ : \forall (X <: \langle\!\langle A \rangle\!\rangle) X \rightarrow \langle\!\langle B_i \rangle\!\rangle \ i \in 1..n]$$

where $A \equiv \text{Object}(X)[l_i; v_i; B_i \ i \in 1..n]$ and $Z \notin FV(\langle\!\langle A \rangle\!\rangle) \cup FV(\langle\!\langle B_i \rangle\!\rangle \ i \in 1..n) \cup \{X\}$

$$\langle\!\langle \text{All}(X <: A)B \rangle\!\rangle \triangleq \forall (X <: \langle\!\langle A \rangle\!\rangle) \langle\!\langle B \rangle\!\rangle$$

This translation of types induces a translation of environments:

Translation of O-2 environments

$$\langle\!\langle \emptyset \rangle\!\rangle \triangleq \emptyset$$

$$\langle\!\langle E, X <: A \rangle\!\rangle \triangleq \langle\!\langle E \rangle\!\rangle, X <: \langle\!\langle A \rangle\!\rangle$$

$$\langle\!\langle E, x:A \rangle\!\rangle \triangleq \langle\!\langle E \rangle\!\rangle, x:\langle\!\langle A \rangle\!\rangle$$

The translation of terms resembles that for O-1; we add a type annotation to field update in order to make the translation syntax-directed.

Translation of O-2 terms

$\langle\langle x \rangle\rangle \triangleq x$
 $\langle\langle \text{object}(x:X=A) l_i=b_i^{i \in 1..n} \text{ end} \rangle\rangle \triangleq obj(X=\langle\langle A \rangle\rangle)[l_i=\zeta(x:X)\langle\langle b_i \rangle\rangle^{i \in 1..n})]$
 $\langle\langle a.l_j \rangle\rangle \triangleq \langle\langle a \rangle\rangle.l_j$
 $\langle\langle (a:A).l := b \rangle\rangle \triangleq \langle\langle a \rangle\rangle.l \Leftarrow (X <: \langle\langle A \rangle\rangle, y:X)\zeta(x:X)\langle\langle b \rangle\rangle$
 where $y, x \notin FV(\langle\langle b \rangle\rangle)$
 $\langle\langle a.l := \text{method}(x:X <: A) b \text{ end} \rangle\rangle \triangleq \langle\langle a \rangle\rangle.l \Leftarrow (X <: \langle\langle A \rangle\rangle, y:X)\zeta(x:X)\langle\langle b \rangle\rangle$
 where $y \notin FV(\langle\langle b \rangle\rangle) \cup \{x\}$
 $\langle\langle \text{new } c \rangle\rangle \triangleq \langle\langle c \rangle\rangle.\text{new}$
 $\langle\langle \text{root} \rangle\rangle \triangleq obj(Z=Obj(X)[\text{new}^+:A])[new=\zeta(z:Z)obj(X=A)[]]$
 where $A \equiv Obj(Y)[]$
 $\langle\langle \text{subclass of } c':\text{Class}(A) \text{ with}(x:X <: A) l_i=b_i^{i \in n+1..n+m} \text{ override } l_i=b_i^{i \in Ovr} \text{ end} \rangle\rangle \triangleq$
 $obj(Z=\langle\langle \text{Class}(A) \rangle\rangle)[$
 $[new=\zeta(z:Z)obj(X=\langle\langle A \rangle\rangle)[l_i=\zeta(s:X)z.l_i(X)(s)^{i \in 1..n+m}]$
 $l_i=\zeta(z:Z)\langle\langle c' \rangle\rangle.l_i^{i \in 1..n-Ovr},$
 $l_i=\zeta(z:Z)\lambda(X <: \langle\langle A \rangle\rangle)\lambda(x:X)\langle\langle b_i \rangle\rangle^{i \in Ovr \cup n+1..n+m}]$
 where $A \equiv \text{Object}(X)[l_i:v_i:B_i^{i \in 1..n+m}]$ and $A' \equiv \text{Object}(X)[l_i:v_i':B_i^{i \in 1..n}]$
 and $z, Z \notin FV(\langle\langle A \rangle\rangle) \cup FV(\langle\langle c' \rangle\rangle) \cup FV(\langle\langle b_i \rangle\rangle^{i \in Ovr \cup n+1..n+m}) \cup \{X\}$
 $\langle\langle c \wedge l_j(A,a) \rangle\rangle \triangleq \langle\langle c \rangle\rangle.l_j(\langle\langle A \rangle\rangle)(\langle\langle a \rangle\rangle)$
 $\langle\langle \text{fun}(X <: A) b \text{ end} \rangle\rangle \triangleq \lambda(X <: \langle\langle A \rangle\rangle)\langle\langle b \rangle\rangle$
 $\langle\langle b(A) \rangle\rangle \triangleq \langle\langle b \rangle\rangle(\langle\langle A \rangle\rangle)$

Finally, we can extend the translation to whole judgments, for example by setting $\langle\langle E \vdash a : A \rangle\rangle \triangleq \langle\langle E \rangle\rangle \vdash \langle\langle a \rangle\rangle : \langle\langle A \rangle\rangle$.

The translation validates all the rules of O-2; we prove a similar result in detail in Chapter 21. Here we give only an informal argument for (Val Subclass), as it gives insight into many parts of the translation at once.

Let $A \equiv \text{Object}(X)[l_i:v_i:B_i^{i \in 1..n+m}]$ and $A' \equiv \text{Object}(X)[l_i:v_i':B_i^{i \in 1..n}]$. Informally, let us assume that $\langle\langle c' \rangle\rangle : \langle\langle \text{Class}(A') \rangle\rangle$, that $\langle\langle A \rangle\rangle <: \langle\langle A' \rangle\rangle$, that $\langle\langle B_i \rangle\rangle <: \langle\langle B_i' \rangle\rangle$ for all $X <: \langle\langle A \rangle\rangle$ and $i \in 1..n-Ovr$, and that $\langle\langle b_i \rangle\rangle : \langle\langle B_i \rangle\rangle$ for all $X <: \langle\langle A \rangle\rangle$, $x:\langle\langle X \rangle\rangle$, and $i \in Ovr \cup n+1..n+m$. We need to show that $\langle\langle \text{subclass of } c':\text{Class}(A') \text{ with}(x:X <: A) l_i=b_i^{i \in n+1..n+m} \text{ override } l_i=b_i^{i \in Ovr} \text{ end} \rangle\rangle : \langle\langle \text{Class}(A) \rangle\rangle$. For this purpose, it suffices to check that: (1) for the *new* method, $obj(X=\langle\langle A \rangle\rangle)[l_i=\zeta(s:X)z.l_i(X)(s)^{i \in 1..n+m}] : \langle\langle A \rangle\rangle$ if $z:\langle\langle \text{Class}(A) \rangle\rangle$; (2) for inherited methods, $\langle\langle c' \rangle\rangle.l_i : \forall(X <: \langle\langle A \rangle\rangle)X \rightarrow \langle\langle B_i \rangle\rangle$ if $i \in 1..n-Ovr$; (3) for overridden and fresh methods, $\lambda(X <: \langle\langle A \rangle\rangle)\lambda(x:X)\langle\langle b_i \rangle\rangle : \forall(X <: \langle\langle A \rangle\rangle)X \rightarrow \langle\langle B_i \rangle\rangle$ if $i \in Ovr \cup n+1..n+m$. Condition (2) follows from the subtyping $\forall(X <: \langle\langle A' \rangle\rangle)X \rightarrow \langle\langle B_i' \rangle\rangle <: \forall(X <: \langle\langle A \rangle\rangle)X \rightarrow \langle\langle B_i \rangle\rangle$.

PART III

Higher-Order Calculi

20 A HIGHER-ORDER CALCULUS

In Part III we study the interaction between objects and higher-order features. In this chapter, we start afresh from our basic calculus with first-order object types and subtyping, $\mathbf{Ob}_{1\ll}$. We enrich this calculus with a structural rule for method update and with variance annotations (introduced in Sections 8.7 and 13.1.3, and studied in Chapters 16 and 17). We put the resulting fragments in the context of a type theory with higher-order subtyping, based on Girard's F_ω [42, 62]. The resulting calculus is the object-oriented counterpart of $F_{\omega\ll}$, the higher-order polymorphic λ -calculus with subtyping [51, 102]. By further adding recursive types, we obtain the calculus $\mathbf{Ob}_{\omega\ll:\mu}$.

The calculus $\mathbf{Ob}_{\omega\ll:\mu}$ is the main subject of this chapter. After defining $\mathbf{Ob}_{\omega\ll:\mu}$ we prove a subject reduction theorem for it, using Compagnoni's approach [51], and then we develop some examples. As these examples show, $\mathbf{Ob}_{\omega\ll:\mu}$ enables us to capture some advanced features of object-oriented languages. In particular, $\mathbf{Ob}_{\omega\ll:\mu}$ provides a general account of inheritance and Self types. It enables us to formalize the notion of object protocol that we introduced informally in Section 3.5 in order to deal with binary methods.

20.1 Syntax of $\mathbf{Ob}_{\omega\ll:\mu}$

The main new feature of $\mathbf{Ob}_{\omega\ll:\mu}$ consists of *type operators* such as, for example:

$$\lambda(X)\forall(Y\ll:X)Y$$

This is a function mapping a type X to the type $\forall(Y\ll:X)Y$. A structure of *kinds* is introduced to classify types and operators. The kind of all types is called Ty . An operator from types to types has kind $Ty \Rightarrow Ty$. Higher-order operators can be expressed as well, with kinds such as $(Ty \Rightarrow Ty) \Rightarrow Ty$. In general, $K \Rightarrow L$ is the kind of the operators mapping kind K to kind L .

Types and operators are collectively called *constructors*. Every well-formed constructor has a kind; we write $A :: K$ to say that constructor A has kind K . The subtype relation is generalized to a higher-order relation: the *subconstructor* (or simply, *inclusion*) relation. On types, this relation reduces to ordinary subtyping. On operators, it is defined inductively as pointwise inclusion; that is, $B <: B'$ holds at kind $K \Rightarrow L$ if for all A of kind K we have $B(A) <: B'(A)$ at kind L . The inclusion relation is written in full in the form $A <: B :: K$; this means that the constructors A and B are both of kind K , and that A is included in B . For example, $A <: Top$ is written in full as $A <: Top :: Ty$.

The general form of bounds for constructor variables is $X <: A :: K$; this means that X has kind K and is included in the constructor A of kind K . These bounds may appear

in quantified types and their terms: $\forall(X:A::K)B$ is the type of those constructor abstractions $\lambda(X:A::K)b$ mapping a constructor X of kind K , included in A , to a term b of type B . In order to simplify the technical treatment of higher-order features, we restrict operators to have bounds of the form $X::K$. We write $\lambda(X::K)B$ for the operator that maps X of kind K to B , and we take $\lambda(X)B$ as an abbreviation for $\lambda(X::Ty)B$. Our initial example of an operator is written in full as:

$$\lambda(X::Ty)\forall(Y<:X::Ty)Y :: Ty \Rightarrow Ty$$

We postpone giving the type rules of $Ob_{\omega<:\mu}$ to Section 20.3; the syntax of $Ob_{\omega<:\mu}$ is summarized in the following table:

Syntax of $Ob_{\omega<:\mu}$

$K, L ::=$	kinds
Ty	types
$K \Rightarrow L$	operators from K to L
$A, B ::=$	constructors
X	constructor variable
Top	the biggest constructor at kind Ty
$[l_i v_i : B_i]^{i \in 1..n}$	object type (l_i distinct, $v_i \in \{^-, -, +\}$)
$\forall(X <: A :: K)B$	bounded universal type
$\mu(X)A$	recursive type
$\lambda(X :: K)B$	operator
$B(A)$	operator application
$a, b ::=$	terms
x	variable
$[l = \zeta(x_i : A_i) b_i]^{i \in 1..n}$	object formation (l_i distinct)
$a.l$	method invocation
$a.l \Leftarrow \zeta(x : A)b$	method update
$\lambda(X <: A :: K)b$	constructor abstraction
$b(A)$	constructor application
$fold(A, a)$	recursive fold
$unfold(a)$	recursive unfold

Thus kinds are either the kind of types or the kinds of operators. Constructors are either variables, types (including a maximum type, object types, bounded universal types, and recursive types), operators, or operator applications. Terms include objects, constructor abstractions, and recursive folds. All expressions are identified up to renaming of bound variables.

We do not include primitive existential quantifiers; we can encode these in terms of universal quantifiers, losing only some equational properties (Section 13.2). Simi-

larly, we do not include function types, which we can encode in terms of object types with variance annotations (Section 8.7).

20.2 Operational Semantics

As usual, the operational semantics is based on reduction judgments of the form $\vdash b \rightsquigarrow v$, where b is a term and v is a result. Results are a subset of the terms, and they cannot be further reduced.

Results are described by the following grammar:

Results

v	$::=$	results
	$[l_i = \zeta(x_i; A_i) b_i]^{i \in 1..n}$	object result
	$\lambda(X <: A :: K) b$	constructor abstraction result
	$fold(A, v)$	recursion result

The reduction relation is defined by the following rules:

Operational semantics

(Red Object) (where $v \equiv [l_i = \zeta(x_i; A_i) b_i]^{i \in 1..n}$)

$$\overline{\vdash v \rightsquigarrow v}$$

(Red Select) (where $v' \equiv [l_i = \zeta(x_i; A_i) b_i]^{i \in 1..n}$)

$$\frac{\vdash a \rightsquigarrow v' \quad \vdash b_j \{v'\} \rightsquigarrow v \quad j \in 1..n}{\vdash a.l_j \rightsquigarrow v}$$

(Red Update)

$$\frac{\vdash a \rightsquigarrow [l_i = \zeta(x_i; A_i) b_i]^{i \in 1..n} \quad j \in 1..n}{\vdash a.l_j \Leftarrow \zeta(x; A) b \rightsquigarrow [l_j = \zeta(x; A_j) b, l_i = \zeta(x_i; A_i) b_i]^{i \in (1..n) - \{j\}}}$$

(Red Fun2::) (where $v \equiv \lambda(X <: A :: K) b$)

$$\overline{\vdash v \rightsquigarrow v}$$

(Red Appl2::)

$$\frac{\vdash b \rightsquigarrow \lambda(X <: A :: K) c\{X\} \quad \vdash c\{A'\} \rightsquigarrow v}{\vdash b(A') \rightsquigarrow v}$$

(Red Fold)

$$\frac{\vdash b \rightsquigarrow v}{\vdash \text{fold}(A,b) \rightsquigarrow \text{fold}(A,v)}$$

(Red Unfold)

$$\frac{\vdash a \rightsquigarrow \text{fold}(A,v)}{\vdash \text{unfold}(a) \rightsquigarrow v}$$

20.3 Typing

The type rules of $\text{Ob}_{\omega<:\mu}$ are formulated in terms of the following judgments:

Judgments for $\text{Ob}_{\omega<:\mu}$

$E \vdash \diamond$	E is an environment
$E \vdash K \text{ kind}$	K is a kind
$E \vdash A :: K$	constructor A has kind K
$E \vdash A \leftrightarrow B :: K$	A and B are equivalent constructors of kind K
$E \vdash A <: B :: K$	A is a subconstructor of B , both of kind K
$E \vdash vA <: v'B$	A is a subtype of B according to variances v and v'
$E \vdash a : A$	a is a value of type A

The need for the judgment $E \vdash A \leftrightarrow B :: K$ arises because the presence of operators entails computation at the constructor level. Because of this, we need to define an equational theory of constructors that accounts for operator evaluation. This equational theory is analogous to the one for the terms of F_1 , but lifted to the level of constructors. For example, the theory implies that $(\lambda(X::Ty)\forall(Y <: X::Ty)Y)(Top)$ equals $\forall(Y <: Top::Ty)Y$. On the other hand, there is no judgment for equivalence of terms; we have given an operational semantics for terms, but will not give a corresponding equational theory. In this respect we proceed as for previous calculi with structural rules.

Several abbreviations are used extensively throughout Part III:

Notation

$[Ty]$	\triangleq	Top
$[K \Rightarrow L]$	\triangleq	$\lambda(X::K)[L]$
$X :: K$	\triangleq	$X <: [K] :: K$ (in environments and some binders)
$X <: A$	\triangleq	$X <: A :: Ty$ (in environments and some binders)
X	\triangleq	$X <: \text{Top} :: Ty$ (in environments and some binders)
$E \vdash A$	\triangleq	$E \vdash A :: Ty$
$E \vdash A \leftrightarrow B$	\triangleq	$E \vdash A \leftrightarrow B :: Ty$
$E \vdash A <: B$	\triangleq	$E \vdash A <: B :: Ty$

These abbreviations allow us to omit some bounds and some kinds. According to the abbreviations, $\lceil K \rceil$ denotes the maximum constructor at kind K .

Using these judgments and notations, we list the inference rules for $Ob_{\omega<\mu}$ grouping them by judgment. The rules are mostly straightforward.

Environment formation

(Env \emptyset)	(Env $X <:$)	(Env x)
	$E \vdash A :: K \quad X \notin dom(E)$	$E \vdash A \quad x \notin dom(E)$
$\emptyset \vdash \diamond$	$E, X <: A :: K \vdash \diamond$	
	$E, x : A \vdash \diamond$	

Kind formation

(Kind Ty)	(Kind \Rightarrow)
$E \vdash \diamond$	$E \vdash K \text{ kind} \quad E \vdash L \text{ kind}$
$E \vdash Ty \text{ kind}$	$E \vdash K \Rightarrow L \text{ kind}$

Constructor formation

(Con X)	(Con Top)
$E', X <: A :: K, E'' \vdash \diamond$	$E \vdash \diamond$
$E', X <: A :: K, E'' \vdash X :: K$	$E \vdash Top :: Ty$
(Con Object) $(l_i \text{ distinct}, v_i \in \{^0, ^-, +\})$	(Con All)
$E \vdash B_i \quad \forall i \in 1..n$	$E, X <: A :: K \vdash B$
$E \vdash [l_i v_i : B_i]^{i \in 1..n}$	$E \vdash \forall (X <: A :: K) B$
(Con Rec)	$E, X \vdash A$
	$E \vdash \mu(X) A$
(Con Abs)	(Con Appl)
$E, X :: K \vdash B :: L$	$E \vdash B :: K \Rightarrow L \quad E \vdash A :: K$
$E \vdash \lambda(X :: K) B :: K \Rightarrow L$	$E \vdash B(A) :: L$

Constructor equivalence

(Con Eq Symm)	(Con Eq Trans)
$E \vdash A \leftrightarrow B :: K$	$E \vdash A \leftrightarrow B :: K \quad E \vdash B \leftrightarrow C :: K$
$E \vdash B \leftrightarrow A :: K$	$E \vdash A \leftrightarrow C :: K$
(Con Eq X)	(Con Eq Top)
$E \vdash X :: K$	$E \vdash \diamond$
$E \vdash X \leftrightarrow X :: K$	$E \vdash Top \leftrightarrow Top :: Ty$

(Con Eq Object) $(l_i \text{ distinct}, v_i \in \{^0, ^-, +\})$	$E \vdash B_i \leftrightarrow B'_i \quad \forall i \in 1..n$	(Con Eq All)
	$\underline{E \vdash [l_i v_i : B_i]^{i \in 1..n} \leftrightarrow [l_i v_i : B'_i]^{i \in 1..n}}$	$E \vdash A \leftrightarrow A' :: K \quad E, X <: A :: K \vdash B \leftrightarrow B'$
		$\underline{E \vdash \forall(X <: A :: K)B \leftrightarrow \forall(X <: A' :: K)B'}$
(Con Eq Rec)		
	$E, X \vdash B \leftrightarrow B'$	
	$\underline{E \vdash \mu(X)B \leftrightarrow \mu(X)B'}$	
(Con Eq Abs)	$E, X :: K \vdash B \leftrightarrow B' :: L$	(Con Eq Appl)
	$\underline{E \vdash \lambda(X :: K)B \leftrightarrow \lambda(X :: K)B' :: K \Rightarrow L}$	$E \vdash B \leftrightarrow B' :: K \Rightarrow L \quad E \vdash A \leftrightarrow A' :: K$
		$\underline{E \vdash B(A) \leftrightarrow B'(A') :: L}$
(Con Eval Beta)		
	$E, X :: K \vdash B[X] :: L \quad E \vdash A :: K$	
	$\underline{E \vdash (\lambda(X :: K)B[X])(A) \leftrightarrow B[A] :: L}$	

Constructor inclusion

(Con Sub Refl)	$E \vdash A \leftrightarrow B :: K$	(Con Sub Trans)	$E \vdash A <: B :: K \quad E \vdash B <: C :: K$
	$\underline{E \vdash A <: B :: K}$		$E \vdash A <: C :: K$
(Con Sub X)	$E', X <: A :: K, E'' \vdash \diamond$	(Con Sub Top)	$E \vdash A :: Ty$
	$\underline{E', X <: A :: K, E'' \vdash X <: A :: K}$		$\underline{E \vdash A <: Top :: Ty}$
(Con Sub Object) $(l_i \text{ distinct})$	$E \vdash v_i B_i <: v'_i B'_i \quad \forall i \in 1..n$	$E \vdash B_i \quad \forall i \in n+1..n+m$	
	$\underline{E \vdash [l_i v_i : B_i]^{i \in 1..n+m} <: [l_i v'_i : B'_i]^{i \in 1..n}}$		
(Con Sub All)	$E \vdash A' <: A :: K \quad E, X <: A' :: K \vdash B <: B'$		
	$\underline{E \vdash \forall(X <: A :: K)B <: \forall(X <: A' :: K)B'}$		
(Con Sub Rec)	$E \vdash \mu(X)A \quad E \vdash \mu(Y)B \quad E, Y, X <: Y \vdash A <: B$		
	$\underline{E \vdash \mu(X)A <: \mu(Y)B}$		

$$\begin{array}{c}
 (\text{Con Sub Abs}) \\
 E, X::K \vdash B <: B' :: L \\
 \hline
 E \vdash \lambda(X::K)B <: \lambda(X::K)B' :: K \Rightarrow L
 \end{array}
 \quad
 \begin{array}{c}
 (\text{Con Sub Appl}) \\
 E \vdash B <: B' :: K \Rightarrow L \quad E \vdash A :: K \\
 \hline
 E \vdash B(A) <: B'(A) :: L
 \end{array}$$

$$\begin{array}{ccc}
 (\text{Con Sub Invariant}) & (\text{Con Sub Covariant}) & (\text{Con Sub Contravariant}) \\
 E \vdash B & E \vdash B <: B' \quad v \in \{^o, ^+\} & E \vdash B' <: B \quad v \in \{^o, ^-\} \\
 \hline
 E \vdash {}^o B <: {}^o B & E \vdash v B <: {}^+ B' & E \vdash v B <: {}^- B'
 \end{array}$$

Term typing

$$\begin{array}{cc}
 (\text{Val Subsumption}) & (\text{Val } x) \\
 E \vdash a : A \quad E \vdash A <: B & E', x:A, E'' \vdash \diamond \\
 \hline
 E \vdash a : B & E', x:A, E'' \vdash x : A
 \end{array}$$

$$\begin{array}{c}
 (\text{Val Object}) \\
 E, x_i:A \vdash b_i : B_i \quad \forall i \in 1..n \quad E \vdash A \leftrightarrow [l_i \forall_i : B_i]^{i \in 1..n} \\
 \hline
 E \vdash [l_i = \zeta(x_i : A) b_i]^{i \in 1..n} : A
 \end{array}$$

$$\begin{array}{c}
 (\text{Val Select}) \\
 E \vdash a : [l_i \forall_i : B_i]^{i \in 1..n} \quad v_j \in \{^o, ^+\} \quad j \in 1..n \\
 \hline
 E \vdash a.l_j : B_j
 \end{array}$$

$$\begin{array}{c}
 (\text{Val Update}) \quad (\text{where } A \equiv [l_i \forall_i : B_i]^{i \in 1..n}) \\
 E \vdash C <: A \quad E \vdash a : C \quad E, x:C \vdash b : B_j \quad v_j \in \{^o, ^-\} \quad j \in 1..n \\
 \hline
 E \vdash a.l_j = \zeta(x:C) b : C
 \end{array}$$

$$\begin{array}{cc}
 (\text{Val Fun2::}) & (\text{Val Appl2::}) \\
 E, X <: A :: K \vdash b : B & E \vdash b : \forall(X <: A :: K)B\{X\} \quad E \vdash A' <: A :: K \\
 \hline
 E \vdash \lambda(X <: A :: K)b : \forall(X <: A :: K)B
 \end{array}
 \quad
 \begin{array}{c}
 E \vdash b(A') : B\{A'\}
 \end{array}$$

$$\begin{array}{cc}
 (\text{Val Fold}) & (\text{Val Unfold}) \quad (\text{where } A \equiv \mu(X)B\{X\}) \\
 E \vdash b : B\{A\} \quad E \vdash A \leftrightarrow \mu(X)B\{X\} & E \vdash a : A \\
 \hline
 E \vdash fold(A, b) : A & E \vdash unfold(a) : B\{A\}
 \end{array}$$

Note that we adopt a structural rule for method update (the one introduced in Section 13.1.3). The structural rule plays an important role in our applications of $Ob_{\omega \leqslant \mu}$; we do not know whether it can be avoided.

20.4 Binary Methods

The treatment of Self types in Part II is restricted to covariant occurrences; there, that restriction is necessary in order to keep subtyping and subsumption working smoothly. As we discussed in Chapter 3, one may want to give up certain subtyping properties in exchange for a treatment of general Self types (Self types without the covariance restriction) [36]. A proper treatment of general Self types naturally involves higher-order constructions [4], and hence was delayed until this chapter.

20.4.1 Binary-Tree Objects

The covariant Self types of Part II were motivated by a number of examples, of which the purest was perhaps the typing of the numerals. A slight modification of that example shows the need for binary methods and general Self types.

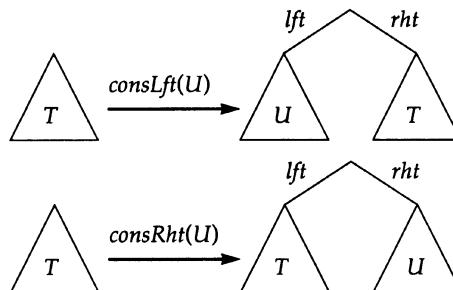


Figure 20-1. Binary trees.

The numerals can be seen as unary trees, with zero as the initial root, and with the successor method adding a new root and a connecting edge to an existing tree. With two distinct successor functions, we obtain binary trees; a *consLft* method adds a new root and a given tree to the left of an existing one, and conversely for *consRht* (Figure 20-1). The methods *consLft* and *consRht* are binary methods in the sense of Section 3.4.

These trees can be coded as follows in $Ob_{\omega < \mu}$:

Object-oriented binary trees

$$\begin{aligned} Bin &\triangleq \mu(X)[isLeaf:Bool, lft:X, rht:X, consLft:X \rightarrow X, consRht:X \rightarrow X] \\ UBin &\triangleq [isLeaf:Bool, lft:Bin, rht:Bin, consLft:Bin \rightarrow Bin, consRht:Bin \rightarrow Bin] \\ leaf : Bin &\triangleq \\ &\quad fold(Bin, \\ &\quad \quad [isLeaf = true, \\ &\quad \quad lft = \zeta(self:UBin) self.lft, \\ &\quad \quad rht = \zeta(self:UBin) self.rht, \end{aligned}$$

$$\begin{aligned}
 consLft &= \zeta(self:UBin) \lambda(lft:Bin) \\
 &\quad fold(Bin, ((self.isLeaf := false).lft := lft).rht := fold(Bin, self)), \\
 consRht &= \zeta(self:UBin) \lambda(rht:Bin) \\
 &\quad fold(Bin, ((self.isLeaf := false).lft := fold(Bin, self)).rht := rht)]
 \end{aligned}$$

Here, as usual, we write $a.l:=b$ for $a.l\Leftarrow\zeta(x:A)b$ when $x\notin FV(b)$.

For example, a tree with two leaves and a joining root can be written:

$unfold(leaf).consLft(leaf) : Bin$

20.4.2 Binary-Tree Classes

We wish to define a class type $BinClass$, and a class $binClass$ of type $BinClass$ that generates trees of type Bin . Our intent is to write $BinClass$ and $binClass$ so that the methods of $binClass$ can be inherited. An inheriting class could, for example, generate trees with nodes containing natural numbers. Such trees would have type:

$NatBin \triangleq \mu(X)[n:Nat, isLeaf:Bool, lft:X, rht:X, consLft:X\rightarrow X, consRht:X\rightarrow X]$

Note that $NatBin <: Bin$ cannot hold, but we still aim to reuse, for example, the $consLft$ binary method.

Working towards this goal, we first transform the type Bin into a type operator $BinOp$. We introduce the following notation in order to facilitate working with type operators:

Notation

- Op is the kind of simple type operators; it stands for $Ty\Rightarrow Ty$.
- $A <: B$ means that A is a suboperator of B ; it stands for $A <: B :: Op$.
- A^* is the fixpoint of the operator A . It stands for the type $\mu(X)A(X)$ where $A :: Op$.

We define the operator $BinOp$ simply by converting the recursion binder of Bin into an abstraction binder:

Object-oriented binary-tree operator

$BinOp :: Op \triangleq \lambda(X)[isLeaf:Bool, lft:X, rht:X, consLft:X\rightarrow X, consRht:X\rightarrow X]$

$Bin :: Ty \triangleq BinOp^*$

$UBin :: Ty \triangleq BinOp(Bin)$

The operator $BinOp$ is the object protocol of Bin , in the sense of Section 3.5. The type Bin can be recovered from $BinOp$ by taking a fixpoint. The type $UBin$, the unfolding of Bin , is obtained by applying $BinOp$ to Bin .

We can now write the definition of $BinClass$, which is based on the familiar notion of parametric pre-methods. The new twist is that the types of pre-methods are quanti-

fied over the suboperators of BinOp , instead of being quantified over the subtypes of Bin (which are scarce). Fixpoints must be introduced where appropriate, to collapse operators down to types:

Object-oriented binary-tree class type

$$\begin{aligned} \text{BinClass} &\triangleq \\ &[\text{new}^+ : \text{Bin}, \\ &\quad \text{isLeaf}^+ : \forall(X \triangleleft \text{BinOp}) X^* \rightarrow \text{Bool}, \\ &\quad \text{lft}^+, \text{rht}^+ : \forall(X \triangleleft \text{BinOp}) X^* \rightarrow X^*, \\ &\quad \text{consLft}^+, \text{consRht}^+ : \forall(X \triangleleft \text{BinOp}) X^* \rightarrow X^* \rightarrow X^*] \end{aligned}$$

The use of $^+$ guarantees that classes cannot be accidentally modified, but is not necessary for the typing of classes.

Writing the raw code for a class of type BinClass is not difficult, but typing such code requires understanding some subtle issues about what is typable within $\text{Ob}_{\omega \triangleleft \mu}$. The following properties, and their derivations, are useful exercises. The final Ex. 5 shows how a binary tree whose type is (partially) unknown can be paired with itself, yielding a binary tree of the unknown type.

Ex. 1: $\emptyset, X \triangleleft \text{BinOp} \vdash X^*$

$$\begin{cases} \emptyset, X \triangleleft \text{BinOp}, Y \vdash X :: \text{Op} & \text{by (Con X)} \\ \downarrow \emptyset, X \triangleleft \text{BinOp}, Y \vdash Y \triangleleft \text{Top} & \text{by (Con Sub X)} \\ \emptyset, X \triangleleft \text{BinOp}, Y \vdash X(Y) & \text{(Con Appl)} \\ \emptyset, X \triangleleft \text{BinOp} \vdash \mu(Y)X(Y) & \text{(Con Rec)} \end{cases}$$

Ex. 2: $\emptyset, X \triangleleft \text{BinOp} \vdash X(X^*) \triangleleft \text{BinOp}(X^*)$

$$\begin{cases} \emptyset, X \triangleleft \text{BinOp} \vdash X \triangleleft \text{BinOp} & \text{by (Con Sub X)} \\ \downarrow \emptyset, X \triangleleft \text{BinOp} \vdash X^* \triangleleft \text{Top} & \text{similar to Ex. 1} \\ \emptyset, X \triangleleft \text{BinOp} \vdash X(X^*) \triangleleft \text{BinOp}(X^*) & \text{(Con Sub Appl)} \end{cases}$$

Ex. 3: $\emptyset, X \triangleleft \text{BinOp}, x : X^* \vdash \text{unfold}(x) : X(X^*)$

$$\begin{cases} \emptyset, X \triangleleft \text{BinOp}, x : X^* \vdash x : X^* & \text{by (Env } x \text{)} \\ \emptyset, X \triangleleft \text{BinOp}, x : X^* \vdash \text{unfold}(x) : X(X^*) & \text{(Val Unfold)} \end{cases}$$

Ex. 4: $\emptyset, X \triangleleft \text{BinOp}, x : X^* \vdash \text{unfold}(x).\text{lft} := x : X(X^*)$

$$\begin{cases} \emptyset, X \triangleleft \text{BinOp}, x : X^* \vdash X(X^*) \triangleleft \text{BinOp}(X^*) & \text{by Ex. 2} \\ \emptyset, X \triangleleft \text{BinOp}, x : X^* \vdash \text{unfold}(x) : X(X^*) & \text{by Ex. 3} \\ \downarrow \emptyset, X \triangleleft \text{BinOp}, x : X^*, y : X(X^*) \vdash x : X^* & \text{by (Env } x \text{)} \\ \emptyset, X \triangleleft \text{BinOp}, x : X^* \vdash \text{unfold}(x).\text{lft} \Leftarrow \varsigma(y : X(X^*))x : X(X^*) & \text{(Val Update)} \end{cases}$$

Ex. 5: $\emptyset, X \triangleleft \text{BinOp}, x:X^* \vdash \text{unfold}(x).\text{consLft}(x) : X^*$

$\emptyset, X \triangleleft \text{BinOp}, x:X^* \vdash \text{unfold}(x) : \text{BinOp}(X^*)$	see Ex. 2 and Ex. 3
$\emptyset, X \triangleleft \text{BinOp}, x:X^* \vdash \text{unfold}(x).\text{consLft} : X^* \rightarrow X^*$	by (Val Subsumption)
$\emptyset, X \triangleleft \text{BinOp}, x:X^* \vdash x : X^*$	by (Val Select)
$\emptyset, X \triangleleft \text{BinOp}, x:X^* \vdash \text{unfold}(x).\text{consLft}(x) : X^*$	by (Env x) by (Val Appl)

We are finally ready to write a class that generates objects of type Bin . The coding of the method *new* is, as usual, lengthy but straightforward. The code for *isLeaf* is trivial, and the code for *lft* and *rht* relies on Ex. 2 and Ex. 3.

Object-oriented binary-tree class

```

binClass : BinClass  ≡
[new =
   $\zeta(z:\text{BinClass})\text{fold}(\text{Bin},$ 
    [ $\text{isLeaf} = \zeta(s:\text{UBin}) z.\text{isLeaf}(\text{BinOp})(\text{fold}(\text{Bin}, s)),$ 
      $\text{lft} = \zeta(s:\text{UBin}) z.\text{lft}(\text{BinOp})(\text{fold}(\text{Bin}, s)),$ 
      $\text{rht} = \zeta(s:\text{UBin}) z.\text{rht}(\text{BinOp})(\text{fold}(\text{Bin}, s)),$ 
      $\text{consLft} = \zeta(s:\text{UBin}) z.\text{consLft}(\text{BinOp})(\text{fold}(\text{Bin}, s)),$ 
      $\text{consRht} = \zeta(s:\text{UBin}) z.\text{consRht}(\text{BinOp})(\text{fold}(\text{Bin}, s))),$ 
      $\text{isLeaf} = \lambda(X \triangleleft \text{BinOp}) \lambda(\text{self}:X^*) \text{true},$ 
      $\text{lft} = \lambda(X \triangleleft \text{BinOp}) \lambda(\text{self}:X^*) \text{unfold}(\text{self}).\text{lft},$ 
      $\text{rht} = \lambda(X \triangleleft \text{BinOp}) \lambda(\text{self}:X^*) \text{unfold}(\text{self}).\text{rht},$ 
      $\text{consLft} =$ 
        $\lambda(X \triangleleft \text{BinOp}) \lambda(\text{rht}:X^*) \lambda(\text{lft}:X^*)$ 
        $\text{fold}(X^*, ((\text{unfold}(\text{rht}).\text{isLeaf} := \text{false}).\text{lft} := \text{lft}).\text{rht} := \text{rht}),$ 
      $\text{consRht} =$ 
        $\lambda(X \triangleleft \text{BinOp}) \lambda(\text{lft}:X^*) \lambda(\text{rht}:X^*)$ 
        $\text{fold}(X^*, ((\text{unfold}(\text{lft}).\text{isLeaf} := \text{false}).\text{lft} := \text{lft}).\text{rht} := \text{rht}))$ 
  ]
]

```

The most intriguing part of this code is the successful typing of *consLft* (and of *consRht*). We start from $\text{unfold}(\text{rht}) : X(X^*)$, as in Ex. 3. The chain of updates maintains the typing $X(X^*)$, as in Ex. 4, by virtue of the structural update rule. The final *fold* maps $X(X^*)$ back into X^* , which is the desired result type.

We can verify that inheritance of binary pre-methods is possible. For example, for the pre-method *consLft* of type $\forall(X \triangleleft \text{BinOp})X^* \rightarrow X^* \rightarrow X^*$, we have:

$$\begin{aligned} \forall(X \triangleleft \text{BinOp})X^* \rightarrow X^* \rightarrow X^* & \triangleleft: \forall(X \triangleleft \text{BinOp}')X^* \rightarrow X^* \rightarrow X^* \\ & \text{for any } \text{BinOp}' \triangleleft: \text{BinOp} \end{aligned}$$

A suboperator of *BinOp* is, for example:

$$\text{NatBinOp} \triangleq$$

$$\lambda(X)[n:\text{Nat}, \text{isLeaf}:\text{Bool}, \text{lft}:X, \text{rht}:X, \text{consLft}:X \rightarrow X, \text{consRht}:X \rightarrow X]$$

so consLft can be inherited into a class built for the type NatBin . In defining such a class, the pre-method consLft can simply be binClass.consLft .

20.5 Basic Properties of $\text{Ob}_{\omega <:\mu}$

In the rest of this chapter we study properties of $\text{Ob}_{\omega <:\mu}$ proving a subject reduction result; we imitate Compagnoni's work on a different higher-order calculus [51]. Discussion about applications of $\text{Ob}_{\omega <:\mu}$ resume in Chapter 21.

Direct proofs by induction on derivations rarely work in $\text{Ob}_{\omega <:\mu}$ because the β rule (Con Eval Beta) introduces expressions of arbitrary shape. In this section we define a *normal system*: a subsystem of $\text{Ob}_{\omega <:\mu}$ that lacks the β rule, but that is sound and complete with respect to $\text{Ob}_{\omega <:\mu}$ over normal forms at the constructor level. The purpose of the normal system is to help us analyze subtyping and constructor equivalence; therefore, the normal system is not concerned with typing judgments. We will be able to carry out proofs by induction on the derivations of the normal system, and then transfer lemmas to $\text{Ob}_{\omega <:\mu}$.

We begin by stating some basic properties of $\text{Ob}_{\omega <:\mu}$.

Lemma 20.5-1

(1) Bound weakening

If $E, X <: A :: K, E' \vdash \mathfrak{S}$ and $E \vdash A' :: K$ and $E \vdash A' <: A :: K$,
then $E, X <: A' :: K, E' \vdash \mathfrak{S}$.

If $E, x:A, E' \vdash \mathfrak{S}$ and $E \vdash A'$ and $E \vdash A' <: A$,
then $E, x:A', E' \vdash \mathfrak{S}$.

(2) Type substitution

If $E, X <: A :: K, E'(X) \vdash \mathfrak{S}(X)$ and $E \vdash A' :: K$ and $E \vdash A' <: A :: K$,
then $E, E'\{A'\} \vdash \mathfrak{S}\{A'\}$.

(3) Value substitution

If $E, x:A, E' \vdash \mathfrak{S}(x)$ and $E \vdash a : A$, then $E, E' \vdash \mathfrak{S}\{a\}$.

(4) Maximum of a kind

If $E \vdash K \text{ kind}$, then $E \vdash \lceil K \rceil :: K$.

If $E \vdash A :: K$, then $E \vdash A <: \lceil K \rceil :: K$.

□

The definition of the normal system relies on the notions of normal form $A^{\eta f}$ of a constructor A , and of the least upper bound $\text{lub}_E(A)$ of a constructor A in an environment E . The least upper bound is defined only for terms of the form $X(A_1)\dots(A_n)$, for $n \geq 0$. If the immediate bound of X in E is B , then $\text{lub}_E(X(A_1)\dots(A_n)) = B(A_1)\dots(A_n)$.

Least upper bounds and normal forms

$\text{lub}_{E,X<:A,E}(X) \triangleq A$
$\text{lub}_E(B(A)) \triangleq \text{lub}_E(B)(A)$
$\text{lub}_E(A)$ undefined otherwise
$X^{nf} \triangleq X$
$\text{Top}^{nf} \triangleq \text{Top}$
$[l_i v_i; B_i]^{nf} \triangleq [l_i v_i; B_i^{nf}]^{i \in 1..n}$
$(\forall(X <: A :: K)B)^{nf} \triangleq \forall(X <: A^{nf} :: K)B^{nf}$
$(\mu(X)A)^{nf} \triangleq \mu(X)A^{nf}$
$(\lambda(X :: K)B)^{nf} \triangleq \lambda(X :: K)B^{nf}$
$(B(A))^{nf} \triangleq \text{if } B^{nf} \equiv \lambda(X :: K)C[X] \text{ for some } X, K, C, \text{ then } (C[A])^{nf} \text{ else } B^{nf}(A^{nf})$
$\emptyset^{nf} \triangleq \emptyset$
$(E, X <: A :: K)^{nf} \triangleq E^{nf}, X <: A^{nf} :: K$
$(E, x:A)^{nf} \triangleq E^{nf}, x:A^{nf}$

We omit a proof that these definitions are proper.

The normal system replaces the judgments $E \vdash A <: B :: K$ and $E \vdash vA <: v'B$ of $\mathcal{Ob}_{\omega <: \mu}$ with the judgments $E \vdash^n A <: B :: K$ and $E \vdash^n vA <: v'B$. The judgments $E \vdash \diamond$, $E \vdash K \text{ kind}$, and $E \vdash A :: K$ are unchanged, and their definition is not repeated. The judgment $E \vdash A \leftrightarrow B :: K$ is dropped because we intend to avoid the problems caused by β -equivalence.

Judgments for the normal system

$E \vdash \diamond$	E is an environment
$E \vdash K \text{ kind}$	K is a kind
$E \vdash A :: K$	constructor A has kind K
$E \vdash^n A <: B :: K$	A is a subconstructor of B , both of kind K
$E \vdash^n vA <: v'B$	A is a subtype of B according to variances v and v'

The rule (Con Sub Refl) of $\mathcal{Ob}_{\omega <: \mu}$ is replaced by the simpler (NCon Sub Refl), which removes the dependence of the inclusion judgment from the equivalence judgment; the judgment $E \vdash A :: K$ is used instead.

The most interesting rules in the normal system are (NCon Sub X) and (NCon Sub Appl). Together they handle inclusions of the form $X(A_1) \dots (A_n) <: C$, by requiring that $(\text{lub}_E(X(A_1) \dots (A_n)))^{nf} <: C$. All other inclusions of the form $B(A) <: C$ must be validated by (NCon Sub Refl).

The handling of normal forms is rather delicate. There are no rules in the normal system able to derive inclusions of terms that are not in normal form. Environments may contain terms that are not in normal form, but the rules (NCon Sub X) and (NCon Sub Appl) normalize them whenever they are accessed.

Normal constructor inclusion

$$\frac{\begin{array}{c} (\text{NCon Sub Refl}) \\ E \vdash A :: K \end{array}}{E \vdash^n A <: A :: K} \quad \frac{\begin{array}{c} (\text{NCon Sub Trans}) \\ E \vdash^n A <: B :: K \quad E \vdash^n B <: C :: K \end{array}}{E \vdash^n A <: C :: K}$$

$$\frac{\begin{array}{c} (\text{NCon Sub X}) \\ E', X <: A :: K, E'' \vdash^n A'^f <: B :: K \end{array}}{E', X <: A :: K, E'' \vdash^n X <: B :: K} \quad \frac{\begin{array}{c} (\text{NCon Sub Top}) \\ E \vdash A :: Ty \end{array}}{E \vdash^n A <: Top :: Ty}$$

$$\frac{\begin{array}{c} (\text{NCon Sub Abs}) \\ E, X :: K \vdash^n B <: B' :: L \end{array}}{E \vdash^n \lambda(X :: K)B <: \lambda(X :: K)B' :: K \Rightarrow L}$$

$$\frac{\begin{array}{c} (\text{NCon Sub Appl}) \quad (\text{when } lub_E(B(A)) \text{ is defined}) \\ E \vdash B :: K \Rightarrow L \quad E \vdash A :: K \quad E \vdash^n (lub_E(B(A)))^f <: C :: L \end{array}}{E \vdash^n B(A) <: C :: L}$$

$$\frac{\begin{array}{c} (\text{NCon Sub Object}) \quad (l_i \text{ distinct}) \\ E \vdash^n v_i B_i <: v'_i B'_i \quad \forall i \in 1..n \quad E \vdash B_i \quad \forall i \in n+1..n+m \end{array}}{E \vdash^n [l_i v_i : B_i]_{i \in 1..n+m} <: [l_i v'_i : B'_i]_{i \in 1..n}}$$

$$\frac{\begin{array}{c} (\text{NCon Sub All}) \\ E \vdash^n A' <: A :: K \quad E, X <: A' :: K \vdash^n B <: B' \end{array}}{E \vdash^n \forall(X <: A :: K)B <: \forall(X <: A' :: K)B'}$$

$$\frac{\begin{array}{c} (\text{NCon Sub Rec}) \\ E \vdash \mu(X)A \quad E \vdash \mu(Y)B \quad E, Y, X <: Y \vdash^n A <: B \end{array}}{E \vdash^n \mu(X)A <: \mu(Y)B}$$

$$\frac{\begin{array}{ccc} (\text{NCon Sub Invariant}) & (\text{NCon Sub Covariant}) & (\text{NCon Sub Contravariant}) \\ E \vdash B & E \vdash^n B <: B' & \nu \in \{^o, ^+\} \\ \hline E \vdash^n {}^o B <: {}^o B & E \vdash^n \nu B <: {}^+ B' & E \vdash^n B' <: B \quad \nu \in \{^o, ^-\} \end{array}}{E \vdash^n \nu B <: {}^- B'}$$

The main properties of the normal system are its soundness and completeness with respect to $Ob_{\omega <: \mu}$ over normal forms. These properties are expressed in Lemma 20.5-5. Lemmas 20.5-2, 20.5-3, and 20.5-4 are all intermediate results for proving Lemma 20.5-5.

Lemma 20.5-2 (Equivalence to normal form)

If $E \vdash A :: K$, then $E \vdash A \leftrightarrow A^{\eta f} :: K$.

□

Lemma 20.5-3 (Substitution for unbounded type variables)

If $E, X::K, E'[X] \vdash^n \mathfrak{S}[X]$ and $E \vdash A :: K$, then $E, (E'[A])^{\eta f} \vdash^n (\mathfrak{S}[A])^{\eta f}$, where $\mathfrak{S} \equiv B <: B' :: K$ or $\mathfrak{S} \equiv v B <: v' B'$.

□

Lemma 20.5-4 (Normal inclusion of operator application)

If $E \vdash^n A <: B :: L \Rightarrow K$ and $E \vdash C :: L$, then $E \vdash^n (A C)^{\eta f} <: (B C)^{\eta f} :: K$.

□

Lemma 20.5-5 (Soundness and completeness of normal system)

If $E \vdash^n A <: B :: K$, then $E \vdash A <: B :: K$.

If $E \vdash^n v B <: v' B'$, then $E \vdash v B <: v' B'$.

If $E \vdash A <: B :: K$, then $E^{\eta f} \vdash^n A^{\eta f} <: B^{\eta f} :: K$.

If $E \vdash v B <: v' B'$, then $E^{\eta f} \vdash^n v B^{\eta f} <: v' B'^{\eta f}$.

□

The normal system allows us to prove lemmas about subtyping in $Ob_{\omega<\mu}$. The lemmas are first proved for the normal judgments, and then transferred to $Ob_{\omega<\mu}$ by soundness and completeness.

Lemma 20.5-6 (Structural subtyping)

- (1) If $E \vdash Top <: C$, then $E \vdash C \leftrightarrow Top$.
- (2) Let $E \vdash \forall(X <: D :: L)C <: \forall(X <: D' :: L')C'$.
Then $L \equiv L'$, $E \vdash D' <: D :: L$, and $E, X <: D :: L \vdash C <: C'$.
- (3) Let $E \vdash [I_i v_i : B_i <: I^{\eta f}] <: [I_i v'_i : B'_i <: I^{\eta f}]$.
 - (1) $J \subseteq I$.
 - (2) If $v_j \in \{^0, +\}$ for some $j \in J$, then $v_j \in \{^0, +\}$ and $E \vdash B_j <: B'_j$.
 - (3) If $v_j \in \{^0, -\}$ for some $j \in J$, then $v_j \in \{^0, -\}$ and $E \vdash B_j <: B'_j$.
- (4) Let $E \vdash \mu(X)B[X] <: \mu(X')B'[X']$.
Then either $E, X \vdash B[X] \leftrightarrow B'[X]$, or $E, X', X <: X' \vdash B[X] <: B'[X']$.

□

20.6 Subject Reduction

Our main result for $Ob_{\omega<\mu}$ is a subject reduction theorem. The proof is quite standard: all the higher-order difficulties have been hidden in the lemmas of the previous section.

Theorem 20.6-1 (Subject reduction)

If $\emptyset \vdash a : A$ and $\vdash a \rightsquigarrow v$, then $\emptyset \vdash v : A$.

Proof

By induction on the derivation of $\vdash a \rightsquigarrow v$.

Cases (Red Object) and (Red Fun2::)

$$\overline{\vdash v \rightsquigarrow v}$$

Immediate, since $a \equiv v$.

Case (Red Select) (where $v' \equiv [l_i=\zeta(x_i:A_i)b_i\{x_i\}^{i \in 1..n}]$)

$$\frac{\vdash c \rightsquigarrow v' \quad \vdash b_j\{v'\} \rightsquigarrow v \quad j \in 1..n}{\vdash c.l_j \rightsquigarrow v}$$

By hypothesis $\emptyset \vdash c.l_j : A$. This must have come from (1) an application of (Val Select) with assumptions $\emptyset \vdash c : C$ with C of the form $[l_i v_i : B_j, \dots]$, $v_i \in \{^o, ^+\}$, and with conclusion $\emptyset \vdash c.l_j : B_j$, followed by (2) a number of subsumption steps implying $\emptyset \vdash B_j <: A$ by transitivity.

By the first induction hypothesis, since $\emptyset \vdash c : C$ and $\vdash c \rightsquigarrow v'$, we have $\emptyset \vdash v' : C$.

Now $\emptyset \vdash v' : C$ must have come from (1) an application of (Val Object) with assumptions $\emptyset, x_i:C \vdash b_i : B_i$ and $\emptyset \vdash C' \leftrightarrow [l_i v_i : B_i]^{i \in 1..n}$, and with conclusion $\emptyset \vdash v' : C'$, followed by (2) a number of subsumption steps implying $\emptyset \vdash C' <: C$ by transitivity. Therefore $\emptyset \vdash [l_i' v_i' : B_i]^{i \in 1..n} <: C$ by (Con Eq Symm), (Con Sub Refl), and (Con Sub Trans). Hence by Lemma 20.5-6(3) we must have $\emptyset \vdash B_j' <: B_j$. From $\emptyset, x_j:C \vdash b_j\{x_j\} : B_j'$ we obtain $\emptyset \vdash b_j\{v'\} : B_j'$ by Lemma 20.5-1(3). Also, $\emptyset \vdash B_j' <: A$ by transitivity. By subsumption, $\emptyset \vdash b_j\{v'\} : A$.

By the second induction hypothesis, since $\emptyset \vdash b_j\{v'\} : A$ and $\vdash b_j\{v'\} \rightsquigarrow v$, we have $\emptyset \vdash v : A$.

Case (Red Update) (where $v \equiv [l_i=\zeta(x_i:A_i)b_i\{x_i\}^{i \in 1..n}]$)

$$\frac{\vdash c \rightsquigarrow v \quad j \in 1..n}{\vdash c.l_j = \zeta(x:C)b \rightsquigarrow [l_i=\zeta(x:A_i)b_i\{x_i\}^{i \in 1..n - |j|}]}$$

By hypothesis $\emptyset \vdash c.l_j = \zeta(x:C)b : A$. This must have come from (1) an application of (Val Update) with assumptions $\emptyset \vdash c : C$, and $\emptyset \vdash C <: D$ where D has the form $[l_j v_j : B_j, \dots]$, and $\emptyset, x:C \vdash b : B_j$ with $v_j \in \{^o, ^-\}$, and with conclusion $\emptyset \vdash c.l_j = \zeta(x:C)b : C$, followed by (2) a number of subsumption steps implying $\emptyset \vdash C <: A$ by transitivity.

By induction hypothesis, since $\emptyset \vdash c : C$ and $\vdash c \rightsquigarrow v$, we have $\emptyset \vdash v : C$.

Now $\emptyset \vdash v : C$ must have come from (1) an application of (Val Object) with assumptions $\emptyset, x:C \vdash b_i : B_i'$ and $\emptyset \vdash C' \leftrightarrow [l_i u_i : B_i' \text{ } i \in 1..n]$ (hence $A_i \equiv C'$ for $i \in 1..n$) and with conclusion $\emptyset \vdash v : C'$, followed by (2) a number of subsumption steps implying $\emptyset \vdash C' <: C$ by transitivity. Therefore $\emptyset \vdash [l_i u_i : B_i' \text{ } i \in 1..n] <: C$ by (Con Eq Symm), (Con Sub Refl), and (Con Sub Trans). By transitivity, $\emptyset \vdash [l_i u_i : B_i' \text{ } i \in 1..n] <: D$. Hence by Lemma 20.5-6(3), we must have $\emptyset \vdash B_j <: B_j'$. From $\emptyset, x:C \vdash b : B_j$ and Lemma 20.5-1(1) we obtain $\emptyset, x:C \vdash b : B_j$, and by subsumption $\emptyset, x:C \vdash b : B_j'$. Then by (Val Object) we obtain $\emptyset \vdash [l_j = \zeta(x;C') \ b, l_i = \zeta(x;C') b_i \text{ } i \in 1..n - \{j\}] : C'$. Since $\emptyset \vdash C' <: A$ by transitivity, we have $\emptyset \vdash [l_j = \zeta(x;C') \ b, l_i = \zeta(x;C') b_i \text{ } i \in 1..n - \{j\}] : A$ by subsumption.

Case (Red Appl2::)

$$\frac{\vdash b \rightsquigarrow \lambda(X <: D :: L) c\{X\} \quad \vdash c\{D''\} \rightsquigarrow v}{\vdash b(D'') \rightsquigarrow v}$$

By hypothesis $\emptyset \vdash b(D'') : A$. This must have come from (1) an application of (Val Appl2<::>) with assumptions $\emptyset \vdash b : \forall(X <: D :: L)C\{X\}$ and $\emptyset \vdash D'' <: D :: L$ and conclusion $\emptyset \vdash b(D'') : C\{D''\}$, followed by (2) a number of subsumption steps implying $\emptyset \vdash C\{D''\} <: A$ by transitivity.

By the first induction hypothesis, since $\emptyset \vdash b : \forall(X <: D :: L)C\{X\}$ and $\vdash b \rightsquigarrow \lambda(X <: D :: L) c\{X\}$, we have $\emptyset \vdash \lambda(X <: D :: L) c\{X\} : \forall(X <: D :: L)C\{X\}$.

Now, $\emptyset \vdash \lambda(X <: D :: L) c\{X\} : \forall(X <: D :: L)C\{X\}$ must have come from (1) an application of (Val Fun2::) with assumption $\emptyset, X <: D :: L' \vdash c\{X\} : C'\{X\}$ and conclusion $\emptyset \vdash \lambda(X <: D :: L') c\{X\} : \forall(X <: D :: L')C'\{X\}$, followed by (2) a number of subsumption steps implying $\emptyset \vdash \forall(X <: D :: L')C'\{X\} <: \forall(X <: D :: L)C\{X\}$. Hence $L \equiv L'$, $\emptyset \vdash D <: D :: L$, and $\emptyset, X <: D :: L \vdash C'\{X\} <: C\{X\}$ by Lemma 20.5-6(2).

We have $\emptyset \vdash D'' <: D :: L$, and also $\emptyset \vdash D'' <: D :: L$ by transitivity. Therefore $\emptyset \vdash C\{D''\} <: C\{D''\}$, and $\emptyset \vdash c\{D''\} : C\{D''\}$ by Lemma 20.5-1(2).

By the second induction hypothesis, since $\emptyset \vdash c\{D''\} : C\{D''\}$ and $\vdash c\{D''\} \rightsquigarrow v$, we have $\emptyset \vdash v : C\{D''\}$. By transitivity and subsumption, $\emptyset \vdash v : A$.

Case (Red Fold)

$$\frac{\vdash b \rightsquigarrow v}{\vdash fold(C,b) \rightsquigarrow fold(C,v)}$$

Assume that $\emptyset \vdash fold(C,b) : A$. This must have come from (1) an application of (Val Fold) with assumptions $\emptyset \vdash C \leftrightarrow \mu(X)B\{X\}$ and $\emptyset \vdash b : B\{C\}$, and conclusion $\emptyset \vdash fold(C,b) : C$, followed by (2) a number of subsumption steps implying $\emptyset \vdash C <: A$ by transitivity. By induction hypothesis, we have $\emptyset \vdash v : B\{C\}$. Therefore we obtain $\emptyset \vdash fold(C,v) : C$ by (Val Fold), and $\emptyset \vdash fold(C,v) : A$ by subsumption.

Case (Red Unfold)

$$\frac{\vdash c \rightsquigarrow \text{fold}(C, v)}{\vdash \text{unfold}(c) \rightsquigarrow v}$$

Assume that $\emptyset \vdash \text{unfold}(c) : A$. This must have come from (1) an application of (Val Unfold) with assumptions $\emptyset \vdash c : \mu(X)B\{X\}$ and conclusion $\emptyset \vdash \text{unfold}(c) : B\{\mu(X)B\{X\}\}$, followed by (2) a number of subsumption steps implying $\emptyset \vdash B\{\mu(X)B\{X\}\} <: A$ by transitivity. By induction hypothesis, we have $\emptyset \vdash \text{fold}(C, v) : \mu(X)B\{X\}$. Now $\emptyset \vdash \text{fold}(C, v) : \mu(X)B\{X\}$ must have come from (1) an application of (Val Fold) with assumptions $\emptyset \vdash C \leftrightarrow \mu(X')B'\{X'\}$ and $\emptyset \vdash v : B'\{C\}$, and conclusion $\emptyset \vdash \text{fold}(C, v) : C$, followed by (2) a number of subsumption steps implying $\emptyset \vdash C <: \mu(X)B\{X\}$ by transitivity. Therefore $\emptyset \vdash \mu(X')B'\{X'\} <: \mu(X)B\{X\}$ by (Con Eq Symm), (Con Sub Refl), and (Con Sub Trans).

By Lemma 20.5-6(4), either:

- (1) $\emptyset, X \vdash B\{X\} \leftrightarrow B'\{X\}$; therefore $\emptyset \vdash \mu(X)B\{X\} \leftrightarrow \mu(X')B'\{X'\}$ by (Con Eq Rec). Hence also $\emptyset \vdash \mu(X)B\{X\} \leftrightarrow C$. Then by (Con Eq Abs), (Con Eq Appl), (Con Eval Beta), we obtain $\emptyset \vdash B\{\mu(X)B\{X\}\} \leftrightarrow B'\{C\}$. Therefore, $\emptyset \vdash B'\{C\} <: A$.
- (2) $\emptyset, X, X' <: X \vdash B'\{X'\} <: B\{X\}$, and therefore $\emptyset, X' <: \mu(X)B\{X\} \vdash B'\{X'\} <: B\{\mu(X)B\{X\}\}$ and then $\emptyset \vdash B'\{C\} <: B\{\mu(X)B\{X\}\}$ by Lemma 20.5-1(2). By transitivity, $\emptyset \vdash B'\{C\} <: A$.

In either case, by subsumption, $\emptyset \vdash v : A$.

□

21 A LANGUAGE WITH MATCHING

In this chapter we introduce a programming language named O-3 that supports general Self types. The surface features of O-3 are similar to those of the language O-1 of Chapter 12, and of the language O-2 of Chapter 19. In contrast with these languages, O-3 handles well both covariant and contravariant Self types, as well as the inheritance of binary methods, but restricts subsumption. O-3 is inspired by the class-based language PolyTOIL, but includes both class-based and object-based constructs.

We give the type rules of O-3, including the rules for a new relation called matching, and we show that these rules are sound by a translation into $Ob_{\omega<\mu}$. In particular, we interpret the matching relation via higher-order subtyping, as first discussed in Chapter 3. The translation gives a concrete example of how $Ob_{\omega<\mu}$ can model both class-based and object-based languages at once, with general Self types.

21.1 Features

Many features of O-3 should be familiar from Chapters 12 and 19, for example object types, class types, and single inheritance. The main new feature is a matching relation, written $A <# B$ [37].

Basically, an object type $A \equiv \text{Object}(X)[\dots]$ matches another object type $B \equiv \text{Object}(X)[\dots]$ (written $A <# B$) when all the object components of B are literally present in A , including any occurrence of the common variable X . So, for example, $\text{Object}(X)[l:X \rightarrow X, m:X]$ matches $\text{Object}(X)[l:X \rightarrow X]$. The rules for matching are thus different from the rules for subtyping between recursive object types; in fact $\text{Object}(X)[l:X \rightarrow X, m:X] <: \text{Object}(X)[l:X \rightarrow X]$ does not hold.

Matching is the basis for inheritance in O-3, much as subtyping is the basis for inheritance in O-1 and O-2. That is, if $A <# B$, then a method of a class for B may be inherited as a method of a class for A . In particular, binary methods can be inherited. For example, a method l of a class for $\text{Object}(X)[l:X \rightarrow X]$ can be inherited as a method of a class for $\text{Object}(X)[l:X \rightarrow X, m:X]$.

An important difference between subtyping and matching is that matching does not support subsumption: when a has type A and $A <# B$, it is not sound in general to infer that a has type B . With the loss of subsumption, it is often necessary to parameterize over all types that match a given type. For example, a function with type $(\text{Object}(X)[l:X \rightarrow X]) \rightarrow C$ cannot be applied to an object of type $\text{Object}(X)[l:X \rightarrow X, m:X]$. For flexibility, one should try to rewrite it with type $\text{All}(Y <# \text{Object}(X)[l:X \rightarrow X]) Y \rightarrow C$, enabling the application to an object of type $\text{Object}(X)[l:X \rightarrow X, m:X]$. Unfortunately, this parameterization may be cumbersome, and requires some foresight.

The matching relation is defined only between object types, and between variables bounded by object types. No subtype relation appears in the syntax of O-3, although subtyping is still used in its type rules. We will have that if A and B are object types and $A <: B$, then $A <\# B$. Moreover, if all occurrences of Self in B are covariant and $A <\# B$, then $A <: B$.

21.2 Syntax

The syntax of O-3 is given below. The most salient syntactic difference with respect to O-2 is that the variable binders are restricted by matching, not by subtyping.

Syntax of O-3 types

$A, B ::=$	types
X	type variable
Top	maximum type
$\text{Object}(X)[l_i; v_i; B_i \ i \in 1..n]$	object type
$\text{Class}(A)$	class type
$\text{All}(X <\# A) B$	match-bound quantified type

Syntax of O-3 terms

$a, b, c ::=$	terms
x	variable
$\text{object}(x:A) l_i = b_i \ i \in 1..n \ \text{end}$	direct object construction
$a.l$	field selection / method invocation
$a.l := \text{method}(x:A) b \ \text{end}$	update with a method
$\text{new } c$	object construction from a class
root	root class
$\text{subclass of } c:C \ \text{with} (x:X <\# A)$	subclass
$l_i = b_i \ i \in n+1..n+m$	additional attributes
$\text{override } l_i = b_i \ i \in Ovr \subseteq 1..n \ \text{end}$	overridden attributes
$c^l(A, a)$	class selection
$\text{fun}(X <\# A) b \ \text{end}$	match-bound type abstraction
$b(A)$	match-bound type application

The type $\text{Class}(A)$ is intended as the type of classes that generate objects of type A , just as in O-1 and O-2. However, the interpretation of these types is different here: $\text{Object}(X)[\dots]$ is understood simply as $\mu(X)[\dots]$ (as in O-1), whereas $\text{Class}(A)$ is interpreted in a more sophisticated way (to be discussed later), using higher-order operators. As in O-1 (but not O-2) there is no variance restriction on the occurrences of the Self variable in an object type. The bounded universal quantifiers of O-2 are replaced

by different bounded universal quantifiers, reflecting the switch from the subtype relation to the matching relation.

The **object** construct provides a way of creating objects directly, without first defining a class; this can be useful because methods in a direct object construction need not be as parametric as in a class definition. A method update construct provides a way of replacing object attributes with new attributes; unlike in O-2, there is no parametricity requirement on the new attributes. The method update construct is intended with a functional semantics: we do not consider imperative features in Part III. Therefore, and for simplicity, we do not include a field update construct.

The construct **new** generates an object from a class. The base class is **root**. In the **subclass** construct, the header $c:C$ **with**($x:X<\#A$) identifies a superclass c , its class type C (having, say, n components), and the type A (having $n+m$ components) of the objects generated by the subclass. The binder introduces a self variable x , and a Self type variable X that matches A . (The binder $x:X<\#A$ stands for $x:X$ and $X<\#A$.) Both the object type A and the class type C must be known; that is, they must not be type variables.

As in O-2, the methods of a superclass c are implicitly inherited into a subclass except when they are replaced by methods in the **override** list. In addition, the methods in the **with** list are added to the subclass. The effect of super is obtained via class selection: the construct $c^l(A,a)$ activates a method l of a class c , provided that appropriate object types and arguments are supplied; the criterion for appropriateness will be given in terms of matching.

We use our standard convention that a missing variance annotation indicates invariance. In examples, we use basic types and function types. As usual we can encode the function type $A \rightarrow B$ as **Object**(X)[$\text{arg}^-:A, \text{val}^+:B$], for X fresh. In addition, we use some abbreviations:

$$\begin{aligned}
\text{Root} &\triangleq \\
&\text{Class}(\text{Object}(X)[]) \\
\text{class with}(x:X<\#A) \ l_i=b_i \ i \in 1..n \ \text{end} &\triangleq \\
&\text{subclass of root:Root with}(x:X<\#A) \ l_i=b_i \ i \in 1..n \ \text{override end} \\
\text{subclass of } c:C \ \text{with}(x:X<\#A) \dots \text{super}.l \dots \text{end} &\triangleq \\
&\text{subclass of } c:C \ \text{with}(x:X<\#A) \dots c^l(X,x) \dots \text{end} \\
\text{object}(x:A) \dots l \ \text{copied from } c \dots \text{end} &\triangleq \\
&\text{object}(x:A) \dots l=c^l(A,x) \dots \text{end} \\
a.l := b &\triangleq \quad \text{where } x \notin FV(b) \text{ and } a : A, \\
a.l := \text{method}(x:A) b \ \text{end} &\quad \text{with } A \text{ clear from context}
\end{aligned}$$

21.3 Examples

As a demonstration of the features of O-3, we consider three examples: binary trees, cells, and points. We discuss the examples informally; the formal details can be filled in using the type rules of Section 21.4.

21.3.1 Binary Trees

First we rewrite, more compactly, the binary-tree example of Section 20.4.1. The code for binary trees in O–3 is reasonably simple: there is no explicit mention of higher-order operators, subtyping, *fold*, or *unfold*; the code looks like program source.

```
Bin  $\triangleq$ 
Object(X)[isLeaf: Bool, lft: X, rht: X, consLft: X→X, consRht: X→X]

binClass : Class(Bin)  $\triangleq$ 
  class with(self: X<# Bin)
    isLeaf = true,
    lft = self.lft,
    rht = self.rht,
    consLft = fun(lft: X) ((self.isLeaf := false).lft := lft). rht := self end,
    consRht = fun(rht: X) ((self.isLeaf := false).lft := self). rht := rht end
  end

leaf: Bin  $\triangleq$ 
  new binClass
```

The definition of the object type *Bin* is the same one we could have given in O–1, but it would be illegal in O–2 because of the contravariant occurrences of *X*. The method bodies rely on some new facts about typing; in particular, if *self* has type *X* and *X*<# *Bin*, then *self.lft* and *self.isLeaf*:=false have type *X*.

Let us consider now a type *NatBin* of binary trees with natural number components. This type is not a subtype of *Bin*. However, *NatBin* matches *Bin* because it has more components and is otherwise identical to *Bin*.

```
NatBin  $\triangleq$ 
Object(X)[n: Nat, isLeaf: Bool, lft: X, rht: X, consLft: X→X, consRht: X→X]
```

Thus we have *NatBin* <# *Bin*. If *b* has type *Bin* and *nb* has type *NatBin*, then *b.consLft(b)* and *nb.consLft(nb)* are allowed, but *b.consLft(nb)* and *nb.consLft(b)* are not.

The methods *consLft* and *consRht* can be used as binary operations on any pair of objects whose common type matches *Bin*. Therefore O–3 allows inheritance of *consLft* and *consRht*. A class for *NatBin* may inherit *consLft* and *consRht* from *binClass*.

Because *NatBin* is not a subtype of *Bin*, generic operations must be explicitly parameterized over all types that match *Bin*. For example, we may write:

```
selfCons : All(X<# Bin)X→X  $\triangleq$ 
  fun(X<# Bin) fun(x: X) x.consLft(x) end end

selfCons(NatBin)(nb) : NatBin for nb : NatBin
```

Explicit parameterization must be used systematically in order to guarantee future flexibility in usage, especially for object types that contain binary methods.

21.3.2 Cells

As a second example, we recode storage cells. In this example, the proper methods are indicated with variance annotations $^+$; the *contents* and *backup* attributes are fields.

```

Cell  ≡
  Object(X)[contents: Nat, get+: Nat, set+: Nat→X]

cellClass : Class(Cell)  ≡
  class with(self: X<#Cell)
    contents = 0,
    get = self.contents,
    set = fun(n: Nat) self.contents := n end
  end

ReCell  ≡
  Object(X)[contents: Nat, get+: Nat, set+: Nat→X, backup: Nat, restore+: X]

reCellClass : Class(ReCell)  ≡
  subclass of cellClass:Class(Cell)
  with(self: X<#ReCell)
    backup = 0,
    restore = self.contents := self.backup
  override
    set = fun(n: Nat) cellClass^set(X, self.backup := self.contents)(n) end
  end

```

We can also write a version of *ReCell* that uses method update instead of a *backup* field:

```

ReCell'  ≡
  Object(X)[contents: Nat, get+: Nat, set+: Nat→X, restore: X]

reCellClass' : Class(ReCell')  ≡
  subclass of cellClass:Class(Cell)
  with(self: X<#ReCell')
    restore = self.contents := 0
  override
    set = fun(n: Nat)
      let m = self.contents
      in cellClass^set(X,
          self.restore := method(y: X) y.contents := m end)
        (n)
      end
    end
  end

```

We obtain $\text{ReCell} <: \text{Cell}$ and $\text{ReCell}' <: \text{Cell}$, because of the covariance of X and the positive variance annotations on the method types of Cell where X occurs. On the other hand, we have also $\text{ReCell} <\# \text{Cell}$ and $\text{ReCell}' <\# \text{Cell}$, and this does not depend on the variance annotations. In O-2, the subtypings $\text{ReCell} <: \text{Cell}$ and $\text{ReCell}' <: \text{Cell}$ do not depend on the variance annotations.

A generic doubling function for all types that match Cell can be written as follows:

```
double : All(X<\#Cell) X→X ≡
  fun(X<\#Cell) fun(x: X) x.set(2*x.get) end end
```

21.3.3 Points

As a final example, in order to allow comparisons with O-1 and O-2, we recode the points of Sections 12.3 and 19.3.1. In O-3, we define:

```
Point ≡ Object(X)[x: Int, eq+: X→Bool, mv+: Int→X]
CPoint ≡ Object(X)[x: Int, c: Color, eq+: X→Bool, mv+: Int→X]
```

These definitions freely use covariant and contravariant occurrences of Self types. The liberal treatment of Self types in O-3 yields $\text{CPoint} <\# \text{Point}$. In O-1, the same definitions of Point and CPoint are valid, but they are less satisfactory because $\text{CPoint} <: \text{Point}$ fails; therefore the O-1 definitions adopt a different type for eq . In O-2, contravariant occurrences of Self types are illegal; therefore the O-2 definitions have a different type for eq , too.

We define two classes pointClass and cPointClass that correspond to the types Point and CPoint , respectively:

```
pointClass : Class(Point) ≡
  class with (self: X<\#Point)
    x = 0,
    eq = fun(other: X) self.x = other.x end,
    mv = fun(dx: Int) self.x := self.x+dx end
  end

cPointClass : Class(CPoint) ≡
  subclass of pointClass: Class(Point)
  with (self: X<\#CPoint)
    c = black
  override
    eq = fun(other: X) super.eq(other) and self.c = other.c end,
    mv = fun(dx: Int) super.mv(dx).c := red end
  end
```

The subclass cPointClass could have inherited both mv and eq . However, we chose to override both of these methods in order to adapt them to deal with colors.

In contrast with the corresponding programs in O-1 and O-2, no uses of typecase are required in this code. The use of typecase is not needed for accessing the color of a point after moving it. (Typecase is needed in O-1 but not in O-2.) Specifically, the overriding code for *mv* does not need a typecase on the result of *super.mv(dx)* in the definition of *cPointClass*. Other code that moves color points does not need a typecase either:

$$\begin{aligned} cPoint : CPoint &\triangleq \text{new } cPointClass \\ movedColor : Color &\triangleq cPoint.mv(1).c \end{aligned}$$

Moreover, O-3 allows us to specialize the binary method *eq* as we have done in the definition of *cPointClass* (unlike O-2). This specialization does not require dynamic typing: we can write *super.eq(other)* without first doing a typecase on *other*.

Thus the treatment of points in O-3 circumvents the previous needs for dynamic typing. The price for this is the loss of the subtyping *CPoint <: Point*, and hence the loss of subsumption between *CPoint* and *Point*.

21.4 Typing

The type rules of O-3 are based on two inclusion judgments: subtyping between types and matching between object types. Correspondingly, there are two well-formedness judgments: being a type and being an object type. Every object type is of course a type.

Judgments

$E \vdash \diamond$	environment E is well-formed
$E \vdash A$	A is a well-formed type in E
$E \vdash A :: Obj$	A is a well-formed object type in E
$E \vdash A <: B$	A is a subtype of B in E
$E \vdash vA <: v'B$	A is a subtype of B in E , with variance annotations v and v'
$E \vdash A <\# B$	A matches B in E
$E \vdash a : A$	a has type A in E

Environments may contain both subtyping and matching assumptions:

Environments

(Env \emptyset)	(Env $X <:$)	(Env $X <\#$)	(Env x)
$\emptyset \vdash \diamond$	$E \vdash A \quad X \notin \text{dom}(E)$	$E \vdash A :: Obj \quad X \notin \text{dom}(E)$	$E \vdash A \quad x \notin \text{dom}(E)$
$E, X <: A \vdash \diamond$		$E, X <\# A \vdash \diamond$	$E, x:A \vdash \diamond$

The judgments for types and object types are connected by the (Type Obj) rule. The other rules are straightforward, except for the fact that variables bound by matching count as object types.

Types

(Type Obj) $E \vdash A :: Obj$	(Type X) $E', X <: A, E'' \vdash \diamond$	(Type Top) $E \vdash \diamond$ $E \vdash Top$
$E \vdash A$	$E', X <: A, E'' \vdash X$	$E \vdash Top$
(Type Class) (where $A \equiv \text{Object}(X)[l_i v_i : B_i^{i \in 1..n}]$) $E, X <: \# A \vdash B_i \quad \forall i \in 1..n$		(Type All<#>) $E, X <: \# A \vdash B$ $E \vdash \text{All}(X <: \# A)B$
$E \vdash \text{Class}(A)$		

Object types

(Obj X) $E', X <: \# A, E'' \vdash \diamond$	(Obj Object) (l_i distinct, $v_i \in \{\circ, -, +\}$) $E, X <: \text{Top} \vdash B_i \quad \forall i \in 1..n$
$E', X <: \# A, E'' \vdash X :: Obj$	$E \vdash \text{Object}(X)[l_i v_i : B_i^{i \in 1..n}] :: Obj$

It follows from these rules that a Self type can never occur as a quantifier bound. This is not possible because in the rule (Obj Object) the type B_i is formed under the assumption $X <: \text{Top}$, where X is the Self type. Object types and class types may still contain quantifiers, as long as their bounds are not a Self type.

There are two inclusion rules for object types. The rule (Sub Object) is essentially a combination of standard rules for subtyping recursive types and object types; the rule (Match Object) is more liberal, and corresponds to higher-order subtyping of operators. Note that the subtype relation is critically used in the definition of matching of two object types, even though matching is the only relation visible in the syntax of O-3. The matching relation is axiomatized as a reflexive and transitive relation; this is a novelty with respect to early treatments of matching based on F-bounded quantification [35].

Subtyping

(Sub Refl) $E \vdash A$	(Sub Trans) $E \vdash A <: B \quad E \vdash B <: C$	(Sub X) $E', X <: A, E'' \vdash \diamond$	(Sub Top) $E \vdash A$
$E \vdash A <: A$	$E \vdash A <: C$	$E', X <: A, E'' \vdash X <: A$	$E \vdash A <: \text{Top}$
<hr/>			
(Sub Object) (where $A \equiv \text{Object}(X)[l_i v_i : B_i^{i \in 1..n+m}]$, $A' \equiv \text{Object}(Y)[l'_i v'_i : B'_i^{i \in 1..n}]$) $E \vdash A \quad E \vdash A' \quad E, Y <: \text{Top}, X <: Y \vdash v_i B_i <: v'_i B'_i \quad \forall i \in 1..n$			
$E \vdash A <: A'$			
<hr/>			
(Sub All<#>) $E \vdash A' <: \# A \quad E, X <: \# A' \vdash B <: B'$			
$E \vdash \text{All}(X <: \# A)B <: \text{All}(X <: \# A')B'$			

(Sub Invariant)	(Sub Covariant)	(Sub Contravariant)
$E \vdash B$	$E \vdash B <: B' \quad v \in \{^o, +\}$	$E \vdash B' <: B \quad v \in \{^o, -\}$
$E \vdash {}^o B <: {}^o B'$	$E \vdash v B <: {}^+ B'$	$E \vdash v B <: {}^- B'$

Matching

(Match Refl)	(Match Trans)	(Match X)
$E \vdash A :: Obj$	$E \vdash A <: B \quad E \vdash B <: C$	$E', X <: A, E'' \vdash \diamond$
$E \vdash A <: A$	$E \vdash A <: C$	$E', X <: A, E'' \vdash X <: A$
(Match Object) (l_i distinct) $E, X <: \text{Top} \vdash v_i B_i <: v'_i B'_i \quad \forall i \in 1..n \quad E, X <: \text{Top} \vdash B_i \quad \forall i \in n+1..m$ $E \vdash \text{Object}(X)[l_i v_i : B_i \text{ }^{i \in 1..n+m}] <: \text{Object}(X)[l_i v'_i : B'_i \text{ }^{i \in 1..n}]$		

We now come to the typing rules for terms. They include a subsumption rule: although subtyping is not generally expected to hold between object types, the subsumption rule is still handy when subtyping does hold. The rules (Val Select), (Val Update), and (Val Class Select) are all structural rules with respect to matching. This is necessary because the methods that form classes and subclasses must typecheck with respect to an unknown type (the Self type) that matches a given object type.

Terms

(Val Subsumption)	(Val x)
$E \vdash a : A \quad E \vdash A <: B$	$E', x:A, E'' \vdash \diamond$

(Val Object) (where $A \equiv \text{Object}(X)[l_i v_i : B_i \{X\}^{i \in 1..n}]$)
$E, x:A \vdash b_i : B_i \{A\} \quad \forall i \in 1..n$

$$E \vdash \text{object}(x:A) \ l_i = b_i \text{ }^{i \in 1..n} \text{ end} : A$$

(Val Select) (where $A \equiv \text{Object}(X)[l_i v_i : B_i \{X\}^{i \in 1..n}]$)
$E \vdash a : A' \quad E \vdash A' <: A \quad v_j \in \{{}^o, +\} \quad j \in 1..n$

$$E \vdash a.l_j : B_j \{A'\}$$

(Val Method Update) (where $A \equiv \text{Object}(X)[l_i v_i : B_i \{X\}^{i \in 1..n}]$)
$E \vdash a : A' \quad E \vdash A' <: A \quad E, x:A' \vdash b : B_j \{A'\} \quad v_j \in \{{}^o, -\} \quad j \in 1..n$

$$E \vdash a.l_j := \text{method}(x:A')b \text{ end} : A'$$

(Val New)

$$E \vdash c : \mathbf{Class}(A)$$

$$\frac{}{E \vdash \mathbf{new} \ c : A}$$

(Val Root)

$$E \vdash \diamond$$

$$\frac{}{E \vdash \mathbf{root} : \mathbf{Class}(\mathbf{Object}(X)[])}$$

(Val Subclass) (where $A \equiv \mathbf{Object}(X)[l_1; B_i^{i \in 1..n+m}]$, $A' \equiv \mathbf{Object}(X')[l_1; B'_i^{i \in 1..n}]$, $Ovr \subseteq 1..n$)

$$E \vdash \mathbf{Class}(A) \quad E \vdash c' : \mathbf{Class}(A') \quad E \vdash A < \# A'$$

$$E, X < \# A \vdash B'_i <: B_i \quad \forall i \in 1..n - Ovr$$

$$E, X < \# A, x : X \vdash b_i : B_i \quad \forall i \in Ovr \cup n+1..n+m$$

$$\frac{}{E \vdash \mathbf{subclass \ of} \ c' : \mathbf{Class}(A') \ \mathbf{with} \ (x : X < \# A) \ l_i = b_i^{i \in n+1..n+m} \ \mathbf{override} \ l_i = b_i^{i \in Ovr} \ \mathbf{end} \\ : \mathbf{Class}(A)}$$

(Val Class Select) (where $A \equiv \mathbf{Object}(X)[l_1; B_i[X]^{i \in 1..n}]$)

$$E \vdash a : A' \quad E \vdash A' < \# A \quad E \vdash c : \mathbf{Class}(A) \quad j \in 1..n$$

$$\frac{}{E \vdash c^{\wedge} l_j(A', a) : B_j[A']}$$

(Val Fun< $\#$)

$$E, X < \# A \vdash b : B$$

$$\frac{}{E \vdash \mathbf{fun}(X < \# A) \ b \ \mathbf{end} : \mathbf{All}(X < \# A)B}$$

(Val Appl< $\#$)

$$E \vdash b : \mathbf{All}(X < \# A)B[X]$$

$$E \vdash A' < \# A$$

$$\frac{}{E \vdash b(A') : B[A']}$$

The rule for building subclasses, (Val Subclass), requires new methods to be parametric in Self (through the assumption $E, X < \# A, x : X \vdash b_i : B_i$). The rule for class selection, (Val Class Select), exploits this parametricity.

The assumption $E \vdash \mathbf{Class}(A)$ in (Val Subclass) is used only to simplify the translation of Section 21.5. The assumption $E \vdash A < \# A'$ is particularly important; it compares the object type associated with the subclass against the object type associated with the superclass. The common attributes of A and A' may differ only in the overridden positions, whose indices are in Ovr . The subtyping rules for variance come into play here. For invariant attributes (often fields) the attribute type in A must be the same as the corresponding attribute type in A' : there is no type specialization on fields. Note, though, that a field may have type Self. For covariant attributes (often proper methods) the overriding attribute type can be a subtype of the overridden attribute type. Thus we can have method specialization on override.

As in O-1 and O-2, not all methods of a class for A' can be inherited in a class for A . Method l_i is inheritable if $E, X < \# A \vdash B'_i <: B_i$ where B'_i and B_i are the result types in A' and A , respectively. This condition holds when B'_i and B_i are identical, for example when they are both equal to the Self type variable X (hence methods that return self are

inheritable), and when they are both equal to a binary method type $X \rightarrow C$ (hence binary methods are inheritable). In this respect O-3 is more liberal than both O-1 and O-2.

21.5 Translation

In Chapter 20 we saw by example how general Self types could be represented in the higher-order calculus $Ob_{\omega<:\mu}$. In this chapter, the language O-3 provides a more appealing syntax for those examples. Now we show that O-3 can be translated into $Ob_{\omega<:\mu}$.

Although the translation is fairly complex, it produces intelligible output. For example, the result of translating the binary-tree example from O-3 into $Ob_{\omega<:\mu}$ yields terms similar to the original $Ob_{\omega<:\mu}$ terms of Section 20.4.1.

21.5.1 Types and Operators

The central point of the translation of O-3 into $Ob_{\omega<:\mu}$ is in the treatment of object types. In certain contexts, the object types of O-3 are interpreted as types of $Ob_{\omega<:\mu}$, while in other contexts they are interpreted as operators of $Ob_{\omega<:\mu}$.

In particular, $\text{Object}(Y)\dots$ is translated either as a type $\mu(Y)\dots$ or as an operator $\lambda(Y)\dots$ in different contexts. Two such contexts are the subtyping and the matching judgments. A subtyping judgment $\emptyset \vdash A <: B$ of O-3 is mapped to a subtyping judgment $\emptyset \vdash \underline{A} <: \underline{B}$ of $Ob_{\omega<:\mu}$ where \underline{A} and \underline{B} are types that correspond to A and B , respectively. On the other hand, a matching judgment $\emptyset \vdash A <\# B$ of O-3 is mapped to a higher-order subtyping judgment $\emptyset \vdash \underline{\underline{A}} <: \underline{\underline{B}}$ of $Ob_{\omega<:\mu}$ where $\underline{\underline{A}}$ and $\underline{\underline{B}}$ are operators.

As in this example, we often use underlining to represent the results of the translation, with double underlining for operators. When both \underline{A} and $\underline{\underline{A}}$ occur in the same context, they are different translations of the same type A , and are related by $\underline{A} \equiv \underline{\underline{A}}^*$.

In this dual translation process, the treatment of type variables is critical. Consider an environment assumption $X <\# \text{Object}(Y)[\dots]$ in O-3, which is mapped to an environment assumption $X <\lambda(Y)[\dots]$ in $Ob_{\omega<:\mu}$. In the translation of the remainder of the judgment, to the right of that assumption, the variable X may be subject to type or operator translations. As an operator, it is translated unchanged as X ; as a type it is turned into its fixpoint X^* .

For example, the judgment:

$$\emptyset, X <\# \text{Object}(W)[l_0:W], Y <\# X, y:Y \vdash \text{Object}(Z)[l_1:X, l_2:Y, l_3:Z]$$

is mapped to:

$$\emptyset, X <\lambda(W)[l_0:W], Y <\# X, y:Y^* \vdash (\lambda(Z)[l_1:X^*, l_2:Y^*, l_3:Z])^*$$

where both object types and variables receive different translations depending on their context.

As another example, let $\underline{A} \equiv \lambda(X)[l:X]$, so that $\underline{A} \equiv \underline{A}^* \equiv \mu(Y)(\lambda(X)[l:X])(Y) = \mu(X)[l:X]$ is the translation of $\text{Object}(X)[l:X]$. Consider the class type:

$$\emptyset \vdash \text{Class}(\text{Object}(X)[l:X])$$

This type is mapped to:

$$\emptyset \vdash [new^+: \underline{A}, l^+: \forall(X <: \underline{A})X^* \rightarrow X^*]$$

which specifies a type for *new* and a type for the pre-method *l*. The type *X* of attribute *l* is translated as *X* within the object type *A*, but as *X** within the pre-method type $\forall(X <: \underline{A})X^* \rightarrow X^*$. As in this example, pre-methods will always have types of the form $\forall(X <: \underline{A})X^* \rightarrow B$; method types will be translated differently in class types and in object types.

21.5.2 Towards a Translation

The following tables are extracted from the translation presented in Section 21.5.3. Our intent here is to summarize the translation of types and terms. These tables are not precise, but they should be useful for developing intuitions.

In this section, the symbol \cong means “informally translates to”, with \cong_{Ty} for translations that yield types, and \cong_{Op} for translations that yield operators. We continue using informal underlining conventions: we represent the translation of a term *a* by *a*, the type translation of a type *A* by *A*, and its operator translation by *A*. We say that a variable *X* is subtype-bound when it is introduced as $X <: A$ for some *A*; we say that *X* is match-bound when it is introduced as $X \# A$ for some *A*.

Translation summary

$X \cong_{Op} X$	(where <i>X</i> is match-bound in the environment)
$\text{Object}(X)[l_i; B_i]^{i \in 1..n} \cong_{Op} \lambda(X)[l_i; B_i]^{i \in 1..n}$	
<hr/>	
$X \cong_{Ty} X$	(when <i>X</i> is subtype-bound in the environment)
$X \cong_{Ty} X^*$	(when <i>X</i> is match-bound in the environment)
$\text{Top} \cong_{Ty} \text{Top}$	
$\text{Object}(X)[l_i; B_i]^{i \in 1..n} \cong_{Ty} (\lambda(X)[l_i; B_i]^{i \in 1..n})^*$	
$\text{Class}(A) \cong_{Ty} [new^+ : \underline{A}, l_i^+ : \forall(X <: \underline{A})X^* \rightarrow B_i]^{i \in 1..n}$	
where $A \equiv \text{Object}(X)[l_i; B_i]^{i \in 1..n}$	
$\text{All}(X \# A)B \cong_{Ty} \forall(X <: \underline{A})B$	
<hr/>	
$x \cong x$	
$\text{object}(x:A) l_i = b_i[x]^{i \in 1..n} \text{ end} \cong \text{fold}(\underline{A}, [l_i = \zeta(x : \underline{A}(A))]_{i \in 1..n})$	
$a.l_j \cong \text{unfold}(a).l_j$	

$a.l_j := \mathbf{method}(x:A')b\{x\} \mathbf{end} \cong$
 $\quad fold(\underline{A'}, unfold(a), l_j = \zeta(x:\underline{A'}(A'))b \{ fold(\underline{A'}, x) \})$
 $\mathbf{new} c \cong \underline{c}.new$
 $\mathbf{root} \cong [new = \zeta(z:[new^+; \mu(X)[]])fold(\mu(X)[], [])]$
subclass of $c':C'$ **with** $(x:X < \# A) l_i = b_i^{i \in n+1..n+m}$ **override** $l_i = b_i^{i \in Ovr}$ **end** \cong
 $[new = \zeta(z:C)fold(\underline{A}[l_i = \zeta(s:\underline{A}(A))z, l_i(\underline{A})(fold(\underline{A}, s))^{i \in 1..n+m}])]$
 $l_i = \zeta(z:C) \underline{c'}.l_i^{i \in 1..n-Ovr},$
 $l_i = \zeta(z:C)\lambda(X < \underline{A})\lambda(x:X^*)b_i^{i \in Ovr \cup n+1..n+m}]$
 where $C \equiv \mathbf{Class}(A)$
 $c^{\wedge}l_j(A', a) \cong \underline{c}.l_j(\underline{A'})(a)$
 $\mathbf{fun}(X < \# A)b \mathbf{end} \cong \lambda(X < \underline{A})b$
 $b(A') \cong \underline{b}(\underline{A'})$

21.5.3 Translation

As the tables of the previous section indicate, the treatment of type variables depends on the environment in which the type variables are bound. The tables show this dependence informally; in this section we give a formal environment-dependent translation of types and terms. The translation of a valid judgment $E \vdash \mathfrak{J}$ of O-3 is a judgment of $Ob_{\omega <: \mu}$ of the form $E \vdash \mathfrak{J}$ where \mathfrak{J} is a function of both \mathfrak{J} and E . A soundness theorem establishes that if $E \vdash \mathfrak{J}$ is derivable in O-3, then its translation is defined and derivable in $Ob_{\omega <: \mu}$.

We define the translation of O-3 through several functions:

Translations for O-3

- | | |
|--------------------------------|--|
| $\langle E \rangle$ | translates an environment E such that $E \vdash \diamond$ in O-3
to an environment of $Ob_{\omega <: \mu}$ |
| $\langle E \vdash A \rangle$ | translates a type A such that $E \vdash A :: Obj$ in O-3
to an operator of $Ob_{\omega <: \mu}$ |
| $\langle E \vdash A \rangle$ | translates a type A such that $E \vdash A$ in O-3
to a type of $Ob_{\omega <: \mu}$ |
| $\langle E \vdash a \rangle$ | translates a term a such that $E \vdash a : A$ in O-3 for some A
to a term of $Ob_{\omega <: \mu}$ |
| $\langle E \vdash E' \rangle$ | translates an environment suffix E' such that $E, E' \vdash \diamond$ in O-3
to an environment suffix of $Ob_{\omega <: \mu}$ |
| $\langle \mathfrak{J} \rangle$ | translates a valid judgment of O-3 to a judgment of $Ob_{\omega <: \mu}$ |
-

The translations for types, terms, environments, and judgments are as follows.

Translation of O-3 types

$$\langle\!\langle E', X < \# A, E'' \vdash X \rangle\!\rangle \triangleq X$$

$$\langle\!\langle E \vdash \text{Object}(X)[l_i v_i : B_i]^{i \in 1..n} \rangle\!\rangle \triangleq \lambda(X)[l_i v_i : \langle\!\langle E, X < : \text{Top} \vdash B_i \rangle\!\rangle^{i \in 1..n}]$$

$$\langle\!\langle E', X < : A, E'' \vdash X \rangle\!\rangle \triangleq X$$

$$\langle\!\langle E', X < \# A, E'' \vdash X \rangle\!\rangle \triangleq X^*$$

$$\langle\!\langle E \vdash \text{Top} \rangle\!\rangle \triangleq \text{Top}$$

$$\langle\!\langle E \vdash \text{Object}(X)[l_i v_i : B_i]^{i \in 1..n} \rangle\!\rangle \triangleq (\lambda(X)[l_i v_i : \langle\!\langle E, X < : \text{Top} \vdash B_i \rangle\!\rangle^{i \in 1..n}])^*$$

$$\langle\!\langle E \vdash \text{Class}(A) \rangle\!\rangle \triangleq$$

$$[\text{new}^+ : \langle\!\langle E \vdash A \rangle\!\rangle, l_i^+ : \forall (X < : \langle\!\langle E \vdash A \rangle\!\rangle) X^* \rightarrow \langle\!\langle E, X < \# A \vdash B_i \rangle\!\rangle^{i \in 1..n}]$$

$$\text{where } A \equiv \text{Object}(X)[l_i v_i : B_i]^{i \in 1..n}$$

$$\langle\!\langle E \vdash \text{All}(X < \# A) B \rangle\!\rangle \triangleq \forall (X < : \langle\!\langle E \vdash A \rangle\!\rangle) \langle\!\langle E, X < \# A \vdash B \rangle\!\rangle$$

Because of the dual interpretation of types, some clauses of the definition are more delicate and interesting than they might appear at first. In particular, the X of $\text{Object}(X)[\dots]$ is treated as a type, whereas the X of $\text{Class}(\text{Object}(X)[\dots])$ is treated as an operator.

Translation of O-3 environments

$$\langle\!\langle \emptyset \rangle\!\rangle \triangleq \emptyset$$

$$\langle\!\langle E, x : A \rangle\!\rangle \triangleq \langle\!\langle E \rangle\!\rangle, x : \langle\!\langle E \vdash A \rangle\!\rangle$$

$$\langle\!\langle E, X < : A \rangle\!\rangle \triangleq \langle\!\langle E \rangle\!\rangle, X < : \langle\!\langle E \vdash A \rangle\!\rangle$$

$$\langle\!\langle E, X < \# A \rangle\!\rangle \triangleq \langle\!\langle E \rangle\!\rangle, X < \# \langle\!\langle E \vdash A \rangle\!\rangle$$

$$\langle\!\langle E \vdash \emptyset \rangle\!\rangle \triangleq \emptyset$$

$$\langle\!\langle E \vdash E', x : A \rangle\!\rangle \triangleq \langle\!\langle E \vdash E' \rangle\!\rangle, x : \langle\!\langle E, E' \vdash A \rangle\!\rangle$$

$$\langle\!\langle E \vdash E', X < : A \rangle\!\rangle \triangleq \langle\!\langle E \vdash E' \rangle\!\rangle, X < : \langle\!\langle E, E' \vdash A \rangle\!\rangle$$

$$\langle\!\langle E \vdash E', X < \# A \rangle\!\rangle \triangleq \langle\!\langle E \vdash E' \rangle\!\rangle, X < \# \langle\!\langle E, E' \vdash A \rangle\!\rangle$$

Translation of O-3 terms

$$\langle\!\langle E', x : A, E'' \vdash x \rangle\!\rangle \triangleq x$$

$\langle E \vdash \text{object}(x:A) l_i = b_i \{x\}^{i \in 1..n} \text{ end} \rangle \triangleq$
 $\quad \text{fold}(\underline{A}, [l_i = \zeta(x:A)(A)] \langle E, x:A \vdash b_i \rangle \{x \leftarrow \text{fold}(\underline{A}, x)\}^{i \in 1..n})]$
 $\quad \text{where } \underline{A} \equiv \langle E \vdash A \rangle \text{ and } \underline{\underline{A}} \equiv \langle E \vdash A \rangle$
 $\langle E \vdash a.l_j \rangle \triangleq \text{unfold}(\langle E \vdash a \rangle).l_j$
 $\langle E \vdash a.l_j := \text{method}(x:A')b \{x\} \text{ end} \rangle \triangleq$
 $\quad \text{fold}(\underline{A'}, \text{unfold}(\langle E \vdash a \rangle).l_j = \zeta(x:\underline{\underline{A'}}(A')) \langle E, x:A' \vdash b \rangle \{x \leftarrow \text{fold}(\underline{A'}, x)\})$
 $\quad \text{where } \underline{A'} \equiv \langle E \vdash A' \rangle \text{ and } \underline{\underline{A'}} \equiv \langle E \vdash A' \rangle$
 $\langle E \vdash \text{new } c \rangle \triangleq \langle E \vdash c \rangle.\text{new}$
 $\langle E \vdash \text{root} \rangle \triangleq [\text{new} = \zeta(z:[\text{new}^+ : \mu(X)[[],[]]) \text{fold}(\mu(X)[[],[]])]$
 $\langle E \vdash \text{subclass of } c':C' \text{ with } (x:X < \# A) l_i = b_i^{i \in n+1..n+m} \text{ override } l_i = b_i^{i \in Ovr} \text{ end} \rangle \triangleq$
 $\quad [\text{new} = \zeta(z:\underline{C}) \text{fold}(\underline{A}, [l_i = \zeta(s:\underline{\underline{A}}(A))z.l_i(A)(\text{fold}(\underline{A}, s))^{i \in 1..n+m}])]$
 $\quad l_i = \zeta(z:\underline{C}) \langle E \vdash c' \rangle.l_i^{i \in 1..n-Ovr},$
 $\quad l_i = \zeta(z:\underline{C}) \lambda(X < \underline{A}) \lambda(x:X^*) \langle E, X < \# A, x:X \vdash b_i \rangle^{i \in Ovr \cup n+1..n+m}]$
 $\quad \text{where } \underline{C} \equiv \langle E \vdash \text{Class}(A) \rangle, \underline{A} \equiv \langle E \vdash A \rangle, \underline{\underline{A}} \equiv \langle E \vdash A \rangle, \text{ and } z \notin \text{dom}(E) \cup \{s\}$
 $\langle E \vdash c \wedge l_j(A', a) \rangle \triangleq \langle E \vdash c \rangle.l_j(\langle E \vdash A' \rangle)(\langle E \vdash a \rangle)$
 $\langle E \vdash \text{fun}(X < \# A)b \text{ end} \rangle \triangleq \lambda(X < : \langle E \vdash A \rangle) \langle E, X < \# A \vdash b \rangle$
 $\langle E \vdash b(A') \rangle \triangleq \langle E \vdash b \rangle(\langle E \vdash A' \rangle)$

Translation of O-3 judgments

$\langle E \vdash \diamond \rangle \triangleq \langle E \rangle \vdash \diamond$
 $\langle E \vdash A \rangle \triangleq \langle E \rangle \vdash \langle E \vdash A \rangle$
 $\langle E \vdash A :: Obj \rangle \triangleq \langle E \rangle \vdash \langle E \vdash A \rangle :: Op$
 $\langle E \vdash A <: B \rangle \triangleq \langle E \rangle \vdash \langle E \vdash A \rangle <: \langle E \vdash B \rangle$
 $\langle E \vdash v A <: v' B \rangle \triangleq \langle E \rangle \vdash v \langle E \vdash A \rangle <: v' \langle E \vdash B \rangle$
 $\langle E \vdash A <\# B \rangle \triangleq \langle E \rangle \vdash \langle E \vdash A \rangle <: \langle E \vdash B \rangle$
 $\langle E \vdash a : A \rangle \triangleq \langle E \rangle \vdash \langle E \vdash a \rangle : \langle E \vdash A \rangle$

21.5.4 Soundness of the Translation

The following lemmas describe some relationships between the translations of types and environments.

Lemma 21.5-1 (Environment suffix)

If $E, E' \vdash \diamond$ in O-3, then $\langle E, E' \rangle \equiv \langle E \rangle, \langle E \vdash E' \rangle$.

□

Lemma 21.5-2 (Relationship between type and operator translations)

If $E \vdash A :: Obj$ in O-3, then $\langle\!\langle E \vdash A \rangle\!\rangle \equiv \langle\!\langle E \vdash A \rangle\!\rangle^*$.

□

Lemma 21.5-3 (Weakening)

- (1) If $E \vdash A$ and $E, E' \vdash \diamond$ in O-3, then $\langle\!\langle E \vdash A \rangle\!\rangle \equiv \langle\!\langle E, E' \vdash A \rangle\!\rangle$.
- (2) If $E \vdash A :: Obj$ and $E, E' \vdash \diamond$ in O-3, then $\langle\!\langle E \vdash A \rangle\!\rangle \equiv \langle\!\langle E, E' \vdash A \rangle\!\rangle$.

□

Lemma 21.5-4 (Substitution)

- (1) If $E, X <:\text{Top}, E' \vdash B$ and $E \vdash A'$ in O-3,
then $\langle\!\langle E, X <:\text{Top}, E' \vdash B \rangle\!\rangle[X \leftarrow \langle\!\langle E \vdash A' \rangle\!\rangle] \equiv \langle\!\langle E, E' \{X \leftarrow A'\} \vdash B \{X \leftarrow A'\} \rangle\!\rangle$.
- (2) If $E, X < \# A, E' \vdash B$ and $E \vdash A' :: Obj$ in O-3,
then $\langle\!\langle E, X < \# A, E' \vdash B \rangle\!\rangle[X \leftarrow \langle\!\langle E \vdash A' \rangle\!\rangle] \equiv \langle\!\langle E, E' \{X \leftarrow A'\} \vdash B \{X \leftarrow A'\} \rangle\!\rangle$.
- (3) If $E, X <:\text{Top}, E' \vdash B :: Obj$ and $E \vdash A'$ in O-3,
then $\langle\!\langle E, X <:\text{Top}, E' \vdash B \rangle\!\rangle[X \leftarrow \langle\!\langle E \vdash A' \rangle\!\rangle] \equiv \langle\!\langle E, E' \{X \leftarrow A'\} \vdash B \{X \leftarrow A'\} \rangle\!\rangle$.
- (4) If $E, X < \# A, E' \vdash B :: Obj$ and $E \vdash A' :: Obj$ in O-3,
then $\langle\!\langle E, X < \# A, E' \vdash B \rangle\!\rangle[X \leftarrow \langle\!\langle E \vdash A' \rangle\!\rangle] \equiv \langle\!\langle E, E' \{X \leftarrow A'\} \vdash B \{X \leftarrow A'\} \rangle\!\rangle$.

□

The following theorem states that the translation of O-3 preserves validity of judgments. As a consequence, the type rules of O-3 are sound.

Theorem 21.5-5

The translation of a valid judgment of O-3 is a valid judgment of $Obj_{\omega <:\mu}$.

Proof

Assume ϑ is a valid judgment of O-3, with a given derivation tree. We show that $\langle\!\langle \vartheta \rangle\!\rangle$ is a valid judgment of $Obj_{\omega <:\mu}$ by induction on the length of the derivation of ϑ .

Cases (Env \emptyset), (Env $X <:$), (Env $X < \#$), (Env x)

Easy.

Case (Type Obj)

We have $\vartheta \equiv E \vdash A$, obtained from $E \vdash A :: Obj$. By induction hypothesis, $\langle\!\langle E \rangle\!\rangle \vdash \langle\!\langle E \vdash A \rangle\!\rangle :: Op$ is derivable in $Obj_{\omega <:\mu}$. Therefore $\langle\!\langle E \rangle\!\rangle \vdash \langle\!\langle E \vdash A \rangle\!\rangle^*$ is derivable as well; this is the translation of ϑ by Lemma 21.5-2.

Cases (Type X), (Type Top)

Easy.

Case (Type Class)

We have $\vartheta \equiv E \vdash \text{Class}(A)$, where $A \equiv \text{Object}(X)[l_i v_i; B_i]^{i \in 1..n}$; this judgment is obtained from $E, X < \# A \vdash B_i$ for $i \in 1..n$. By induction hypothesis, the judgments $\langle\!\langle E,$

$X < \# A \vdash B_i$ for $i \in 1..n$ are derivable in $Ob_{\omega <:\mu}$ these judgments can be rewritten as $\langle E \rangle, X <:(E \vdash A) \vdash \langle E, X < \# A \vdash B_i \rangle$, for $i \in 1..n$. Therefore $\langle E \rangle \vdash [new^+:(E \vdash A)^*, l_i^+:\forall(X <:(E \vdash A))X^* \rightarrow \langle E, X < \# A \vdash B_i \rangle]^{i \in 1..n}$ is derivable; by Lemma 21.5-2 this is the translation of ϑ .

Cases (Type All<#), (Obj X), (Obj Object), (Sub Refl), (Sub Trans)

Easy.

Case (Sub X)

We have $\vartheta \equiv E', X <: A, E'' \vdash X <: A$, obtained from $E', X <: A, E'' \vdash \diamond$. By induction hypothesis, $\langle E', X <: A, E'' \vdash \diamond \rangle$ is derivable in $Ob_{\omega <:\mu}$ this judgment is $\langle E' \rangle, X <:(E' \vdash A)$, $\langle E', X <: A \vdash E'' \rangle \vdash \diamond$ by Lemma 21.5-1. By (Sub X) we obtain $\langle E' \rangle, X <:(E' \vdash A)$, $\langle E', X <: A \vdash E'' \rangle \vdash X <: (E' \vdash A)$. By Lemma 21.5-3, this judgment is $\langle E' \rangle, X <:(E' \vdash A)$, $\langle E', X <: A \vdash E'' \rangle \vdash X <: \langle E', X <: A, E'' \vdash A \rangle$; this is the translation of ϑ .

Cases (Sub Top), (Sub Object), (Sub All<#), (Sub Invariant), (Sub Covariant), (Sub Contravariant), (Match Refl), (Match Trans)

Easy.

Case (Match X)

We have $\vartheta \equiv E', X < \# A, E'' \vdash X < \# A$, obtained from $E', X < \# A, E'' \vdash \diamond$. By induction hypothesis, $\langle E', X < \# A, E'' \vdash \diamond \rangle$ is derivable in $Ob_{\omega <:\mu}$ this judgment is $\langle E' \rangle, X <:(E' \vdash A)$, $\langle E', X < \# A \vdash E'' \rangle \vdash \diamond$ by Lemma 21.5-1. By (Match X) we obtain $\langle E' \rangle, X <:(E' \vdash A)$, $\langle E', X < \# A \vdash E'' \rangle \vdash X <: (E' \vdash A)$. By Lemma 21.5-3, this judgment is $\langle E' \rangle, X <:(E' \vdash A)$, $\langle E', X < \# A \vdash E'' \rangle \vdash X <: \langle E', X < \# A, E'' \vdash A \rangle$; this is the translation of ϑ .

Case (Match Object), (Val Subsumption)

Easy.

Case (Val x)

We have $\vartheta \equiv E', x:A, E'' \vdash x : A$, obtained from $E', x:A, E'' \vdash \diamond$. By induction hypothesis, $\langle E', x:A, E'' \vdash \diamond \rangle$ is derivable in $Ob_{\omega <:\mu}$ this judgment is $\langle E' \rangle, x:(E' \vdash A)$, $\langle E', x:A \vdash E'' \rangle \vdash \diamond$ by Lemma 21.5-1. By (Val x) we obtain $\langle E' \rangle, x:(E' \vdash A)$, $\langle E', x:A \vdash E'' \rangle \vdash x : (E' \vdash A)$. By Lemma 21.5-3, this judgment is $\langle E' \rangle, x:(E' \vdash A)$, $\langle E', x:A \vdash E'' \rangle \vdash x : \langle E', x:A, E'' \vdash A \rangle$; this is the translation of ϑ .

Case (Val Object)

We have $\vartheta \equiv E \vdash \text{object}(x:A) l_i = b_i \{x\}^{i \in 1..n} \text{ end} : A$, where $A \equiv \text{Object}(X)[l_i \cup_i B_i \{X\}_{i \in 1..n}]$, obtained from $E, x:A \vdash b_i \{x\} : B_i \{A\}$ for $i \in 1..n$. By induction hypothesis, the judgments $\langle E \rangle, x:(E \vdash A) \vdash \langle E, x:A \vdash b_i \{x\} : \langle E, x:A \vdash B_i \{A\} \rangle \rangle$ for $i \in 1..n$ are derivable in $Ob_{\omega <:\mu}$. These can be simplified to $\langle E \rangle, x:(E \vdash A) \vdash \langle E, x:A \vdash b_i \{x\} : \langle E \vdash B_i \{A\} \rangle \rangle$.

We have $\langle E \vdash A \rangle \equiv \langle E \vdash A \rangle^* \equiv \mu(X)(E \vdash A)(X)$ by Lemma 21.5-2, and $\langle E \vdash A \rangle(\langle E \vdash A \rangle) \equiv (\langle E \vdash A \rangle(X))\{X \leftarrow \langle E \vdash A \rangle\}$. From these equalities we obtain $\langle E \rangle, x:(E \vdash A)(\langle E \vdash A \rangle) \vdash \text{fold}(\langle E \vdash A \rangle, x') : \langle E \vdash A \rangle$. By weakening $\langle E \rangle, x:(E \vdash A) \vdash \langle E, x:A \vdash b_i \{x\} : \langle E \vdash B_i \{A\} \rangle \rangle$, we

obtain $\langle E \rangle, x':\langle E \vdash A \rangle(\langle E \vdash A \rangle), x:\langle E \vdash A \rangle \vdash \langle E, x:A \vdash b_i \rangle\{x\} : \langle E \vdash B_i\{A\} \rangle$. By Lemma 20.5-1(3) and renaming of x' to x , we obtain $\langle E \rangle, x:\langle E \vdash A \rangle(\langle E \vdash A \rangle) \vdash \langle E, x:A \vdash b_i \rangle\{fold(\langle E \vdash A \rangle, x)\} : \langle E \vdash B_i\{A\} \rangle$.

A few further steps are needed before we can form an object through (Val Object) in $Ob_{\omega < \mu}$. First we note that $\langle E \vdash A \rangle \equiv \lambda(X)[l_i v_i : \langle E, X <: \text{Top} \vdash B_i\{X\} \rangle^{i \in 1..n}]$; Lemma 21.5-4(1) yields $[l_i v_i : \langle E, X <: \text{Top} \vdash B_i\{X\} \rangle^{i \in 1..n}] \{X \leftarrow \langle E \vdash A \rangle\} \equiv [l_i v_i : \langle E \vdash B_i\{A\} \rangle^{i \in 1..n}]$, and hence $\langle E \rangle \vdash \langle E \vdash A \rangle(\langle E \vdash A \rangle) \leftrightarrow [l_i v_i : \langle E \vdash B_i\{A\} \rangle^{i \in 1..n}]$ through (Con Eval Beta). Now (Val Object) yields $\langle E \rangle \vdash [l_i = \zeta(x:\langle E \vdash A \rangle(\langle E \vdash A \rangle)) \langle E, x:A \vdash b_i \rangle\{fold(\langle E \vdash A \rangle, x)\}^{i \in 1..n}] : \langle E \vdash A \rangle(\langle E \vdash A \rangle)$. It follows that $\langle E \rangle \vdash fold(\langle E \vdash A \rangle, [l_i = \zeta(x:\langle E \vdash A \rangle(\langle E \vdash A \rangle)) \langle E, x:A \vdash b_i \rangle\{fold(\langle E \vdash A \rangle, x)\}^{i \in 1..n}]) : \langle E \vdash A \rangle$; this is the translation of ϑ .

Case (Val Select)

We have $\vartheta \equiv E \vdash a.l_j : B_j\{A'\}$, obtained from $E \vdash a : A'$ and $E \vdash A' <# \text{Object}(X)[l_i v_i : B_i\{X\}^{i \in 1..n}]$. By induction hypothesis, $\langle E \rangle \vdash \langle E \vdash a \rangle : \langle E \vdash A' \rangle$ and $\langle E \rangle \vdash \langle E \vdash A' \rangle <: \lambda(X)[l_i v_i : \langle E, X <: \text{Top} \vdash B_i \rangle^{i \in 1..n}]$ are derivable in $Ob_{\omega < \mu}$ and by Lemma 21.5-2 we have $\langle E \vdash A' \rangle \equiv \langle E \vdash A' \rangle^* \equiv \mu(X)\langle E \vdash A' \rangle(X)$. We obtain $\langle E \rangle \vdash unfold(\langle E \vdash a \rangle : \langle E \vdash A' \rangle(\langle E \vdash A' \rangle))$ and $\langle E \rangle \vdash \langle E \vdash A' \rangle(\langle E \vdash A' \rangle) <: [\lambda(X)[l_i v_i : \langle E, X <: \text{Top} \vdash B_i \rangle^{i \in 1..n}]](\langle E \vdash A' \rangle)$, hence $\langle E \rangle \vdash \langle E \vdash A' \rangle(\langle E \vdash A' \rangle) <: [l_i v_i : \langle E, X <: \text{Top} \vdash B_i \rangle\{X \leftarrow \langle E \vdash A' \rangle\}^{i \in 1..n}]$ through (Con Eval Beta) and $\langle E \rangle \vdash unfold(\langle E \vdash a \rangle) : [l_i v_i : \langle E, X <: \text{Top} \vdash B_i \rangle\{X \leftarrow \langle E \vdash A' \rangle\}^{i \in 1..n}]$ by subsumption. By (Val Select) and Lemma 21.5-4(1), we obtain $\langle E \rangle \vdash unfold(\langle E \vdash a \rangle).l_j : \langle E \vdash B_j\{X \leftarrow A'\} \rangle$; this is the translation of ϑ .

Case (Val Method Update)

We have $\vartheta \equiv E \vdash a.l_j := \text{method}(x:A')b \text{ end} : A'$, obtained from $E \vdash a : A'$, $E \vdash A' <# \text{Object}(X)[l_i v_i : B_i\{X\}^{i \in 1..n}]$, and $E, x:A' \vdash b : B_j\{A'\}$. By induction hypothesis, $\langle E \rangle \vdash \langle E \vdash a \rangle : \langle E \vdash A' \rangle$ and $\langle E \rangle \vdash \langle E \vdash A' \rangle <: \lambda(X)[l_i v_i : \langle E, X <: \text{Top} \vdash B_i \rangle^{i \in 1..n}]$ are derivable in $Ob_{\omega < \mu}$. By induction hypothesis, also, $\langle E \rangle, x:\langle E \vdash A' \rangle \vdash \langle E, x:A' \vdash b \rangle\{x\} : \langle E, x:A' \vdash B_j\{A'\} \rangle$ is derivable in $Ob_{\omega < \mu}$. We obtain $\langle E \rangle, x:\langle E \vdash A' \rangle(\langle E \vdash A' \rangle) \vdash \langle E, x:A' \vdash B_j\{A'\} \rangle\{fold(\langle E \vdash A' \rangle, x)\} : \langle E, x:A' \vdash B_j\{A'\} \rangle$ much as in the case of (Val Object) (by substitution and by Lemma 21.5-2, which says that $\langle E \vdash A' \rangle \equiv \langle E \vdash A' \rangle^* \equiv \mu(X)\langle E \vdash A' \rangle(X)$). As in the case of (Val Select), we obtain $\langle E \rangle \vdash unfold(\langle E \vdash a \rangle : \langle E \vdash A' \rangle(\langle E \vdash A' \rangle))$ and $\langle E \rangle \vdash \langle E \vdash A' \rangle(\langle E \vdash A' \rangle) <: [l_i v_i : \langle E, X <: \text{Top} \vdash B_i \rangle\{X \leftarrow \langle E \vdash A' \rangle\}^{i \in 1..n}]$. From Lemmas 21.5-3(1) and 21.5-4(1), we get that $\langle E, x:A' \vdash B_j\{X \leftarrow A'\} \rangle \equiv \langle E \vdash B_j\{X \leftarrow A'\} \rangle \equiv \langle E, X <: \text{Top} \vdash B_j \rangle\{X \leftarrow \langle E \vdash A' \rangle\}$. Hence we obtain $\langle E \rangle \vdash unfold(\langle E \vdash a \rangle).l_j = \zeta(x:\langle E \vdash A' \rangle(\langle E \vdash A' \rangle)) \langle E, x:A' \vdash b \rangle\{fold(\langle E \vdash A' \rangle, x)\} : \langle E \vdash A' \rangle(\langle E \vdash A' \rangle)$ by (Val Update); finally we obtain $\langle E \rangle \vdash fold(\langle E \vdash A' \rangle, unfold(\langle E \vdash a \rangle).l_j = \zeta(x:\langle E \vdash A' \rangle(\langle E \vdash A' \rangle))) \langle E, x:A' \vdash b \rangle\{fold(\langle E \vdash A' \rangle, x)\} : \langle E \vdash A' \rangle$, which is the translation of ϑ .

Case (Val New)

We have $\vartheta \equiv E \vdash \text{new } c : A$, obtained from $E \vdash c : \text{Class}(A)$. By induction hypothesis, $\langle E \rangle \vdash \langle E \vdash c \rangle : [\text{new}^+ : \langle E \vdash A \rangle, \dots]$ is derivable in $Ob_{\omega < \mu}$. Hence we obtain $\langle E \rangle \vdash \langle E \vdash c \rangle.\text{new} : \langle E \vdash A \rangle$; this is the translation of ϑ .

Case (Val Root)

Easy.

Case (Val Subclass)

We have $\emptyset \equiv E \vdash \text{subclass of } c' : \text{Class}(A') \text{ with } (x : X < \# A) l_i = b_i \underset{i \in 1..n+m}{\text{override}} l_i = b_i$ $\underset{i \in Ovr}{\text{end}}$ $\text{Class}(A)$, obtained from a number of hypotheses, which we will invoke in the course of the argument. First we set: $\underline{A} \triangleq \langle E \vdash A \rangle \equiv \lambda(X)[l_i v_i : \langle E, X < \# \text{Top} \vdash B_i \rangle \{X\} \underset{i \in 1..n}{\}]$, $A' \triangleq \langle E \vdash A' \rangle \equiv \lambda(X)[l_i v_i : \langle E, X < \# \text{Top} \vdash B_i' \rangle \{X\} \underset{i \in 1..n}{}]$, $\underline{A} \triangleq \langle E \vdash A \rangle$, $A' \triangleq \langle E \vdash A' \rangle$, $\underline{\underline{C}} \triangleq \langle E \vdash \text{Class}(A) \rangle \equiv [new^+ : \underline{A}, l_i^+ : \forall(X < \underline{A}) X^* \rightarrow \langle E, X < \# A \vdash B_i \rangle \{X\} \underset{i \in 1..n+m}{}]$, and $\underline{C} \triangleq \langle E \vdash \text{Class}(A') \rangle \equiv [new^+ : \underline{A}', l_i^+ : \forall(X < \underline{A}') X^* \rightarrow \langle E, X < \# A' \vdash B_i' \rangle \{X\} \underset{i \in 1..n}{}]$. By Lemma 21.5-2, $\underline{A} \equiv \underline{A}^*$ and $\underline{A}' \equiv \underline{A}'^*$.

By induction hypothesis, for $i \in Ovr \cup n+1..n+m$, $\langle E, X < \# A, x : X \vdash b_i : B_i \rangle \equiv \langle E \rangle, X < \underline{A}, x : X^* \vdash \langle E, X < \# A, x : X \vdash b_i \rangle : \langle E, X < \# A, x : X \vdash B_i \rangle$ is derivable in $Ob_{\omega <:\mu}$ hence $\langle E \rangle \vdash \lambda(X < \underline{A}) \lambda(x : X^*) \langle E, X < \# A, x : X \vdash b_i \rangle : \forall(X < \underline{A}) X^* \rightarrow \langle E, X < \# A \vdash B_i \rangle$ is derivable as well.

By induction hypothesis, for $i \in 1..n-Ovr$, $\langle E \rangle \vdash \underline{A} <: \underline{A}'$ and $\langle E \rangle, X < \underline{A} \vdash \langle E, X < \# A \vdash B_i \rangle$ are derivable in $Ob_{\omega <:\mu}$ and hence so is $\langle E \rangle \vdash \forall(X < \underline{A}) X^* \rightarrow \langle E, X < \# A \vdash B_i \rangle <: \forall(X < \underline{A}) X^* \rightarrow \langle E, X < \# A \vdash B_i \rangle$.

Moreover, also by induction hypothesis, $\langle E \rangle \vdash \langle E \vdash c' \rangle : \underline{C}$ is derivable in $Ob_{\omega <:\mu}$. We obtain $\langle E \rangle \vdash \langle E \vdash c' \rangle . l_i : \forall(X < \underline{A}) X^* \rightarrow \langle E, X < \# A \vdash B_i \rangle$ by (Val Select) and subsumption. Using the definition of \underline{C} and Lemma 21.5-2 we can prove that $\langle E \rangle, z : \underline{C}, s : \underline{\underline{A}}(A) \vdash z.l_i(\underline{A})(fold(\underline{A}, s)) : \langle E, X < \# A \vdash B_i \rangle \{X \leftarrow \underline{A}\}$; this is $\langle E \rangle, z : \underline{C}, s : \underline{\underline{A}}(A) \vdash z.l_i(\underline{A})(fold(\underline{A}, s)) : \langle E \vdash B_i[A] \rangle$ by Lemma 21.5-4(2). Hence, much as in the case of (Val Object), we obtain $\langle E \rangle, z : \underline{C} \vdash [l_i = \zeta(s : \underline{\underline{A}}(A)) z.l_i(\underline{A})(fold(\underline{A}, s))] \underset{i \in 1..n+m}{:} \underline{A}(A)$.

Combining these intermediate results, and using Lemma 21.5-2, we obtain that $\langle E \rangle \vdash [new = \zeta(z : \underline{C}) fold(\underline{A}, [l_i = \zeta(s : \underline{\underline{A}}(A)) z.l_i(\underline{A})(fold(\underline{A}, s))] \underset{i \in 1..n+m}{:})], l_i = \zeta(z : \underline{C}) \langle E \vdash c' \rangle . l_i \underset{i \in 1..n-Ovr}{:} l_i = \zeta(z : \underline{C}) \lambda(X < \underline{A}) \lambda(x : X^*) \langle E, X < \# A, x : X \vdash b_i \rangle \underset{i \in Ovr \cup n+1..n+m}{:} \underline{C}$ is derivable in $Ob_{\omega <:\mu}$. This is the translation of \emptyset .

Case (Val Class Select)

We have $\emptyset \equiv E \vdash c \wedge l_j(A', a) : B_j[A']$, obtained from $E \vdash a : A'$, $E \vdash A' < \# A$, and $E \vdash c : \text{Class}(A)$, with $A \equiv \text{Object}(X)[l_i v_i : B_i \{X\} \underset{i \in 1..n}{}]$. By induction hypothesis, $\langle E \rangle \vdash \langle E \vdash a \rangle : \langle E \vdash A' \rangle$, $\langle E \rangle \vdash \langle E \vdash A' \rangle <: \langle E \vdash A \rangle$, and $\langle E \rangle \vdash \langle E \vdash c \rangle : \langle E \vdash \text{Class}(A) \rangle$ are derivable in $Ob_{\omega <:\mu}$ where $\langle E \vdash \text{Class}(A) \rangle \equiv [new^+ : \langle E \vdash A \rangle, l_i^+ : \forall(X < \langle E \vdash A \rangle) X^* \rightarrow \langle E, X < \# A \vdash B_i \rangle \underset{i \in 1..n}{}]$. We obtain $\langle E \rangle \vdash \langle E \vdash c \rangle . l_j : \forall(X < \langle E \vdash A \rangle) X^* \rightarrow \langle E, X < \# A \vdash B_j \rangle$, hence $\langle E \rangle \vdash \langle E \vdash c \rangle . l_j(\langle E \vdash A' \rangle) : \langle E \vdash A' \rangle^* \rightarrow \langle E, X < \# A \vdash B_j \rangle \{X \leftarrow \langle E \vdash A' \rangle\}$. By Lemma 21.5-4(2), this is the same as $\langle E \rangle \vdash \langle E \vdash c \rangle . l_j(\langle E \vdash A' \rangle) : \langle E \vdash A' \rangle^* \rightarrow \langle E \vdash B_j[X \leftarrow A'] \rangle$. By Lemma 21.5-2, $\langle E \vdash A' \rangle \equiv \langle E \vdash A' \rangle^*$, hence $\langle E \rangle \vdash \langle E \vdash c \rangle . l_j(\langle E \vdash A' \rangle)(\langle E \vdash a \rangle) : \langle E \vdash B_j[X \leftarrow A'] \rangle$, which is the translation of \emptyset .

Case (Val Fun<#)

Easy.

Case (Val Appl <#>)

We have $\vartheta \equiv E \vdash b(A') : B\{A'\}$, obtained from $E \vdash b : \text{All}(X < \# A)B(X)$ and $E \vdash A' < \# A$. By induction hypothesis, $\langle\!\langle E \rangle\!\rangle \vdash \langle\!\langle E \vdash b \rangle\!\rangle : \forall(X < \langle\!\langle E \vdash A \rangle\!\rangle)(E, X < \# A \vdash B)$ and $\langle\!\langle E \rangle\!\rangle \vdash \langle\!\langle E \vdash A' \rangle\!\rangle <: \langle\!\langle E \vdash A \rangle\!\rangle$ are derivable in $Ob_{\omega < \mu}$. Therefore $\langle\!\langle E \rangle\!\rangle \vdash \langle\!\langle E \vdash b \rangle\!\rangle(\langle\!\langle E \vdash A' \rangle\!\rangle) : \langle\!\langle E, X < \# A \vdash B \rangle\!\rangle\{X \leftarrow \langle\!\langle E \vdash A' \rangle\!\rangle\}$ is derivable as well. By Lemma 21.5-4(2), this judgment is $\langle\!\langle E \rangle\!\rangle \vdash \langle\!\langle E \vdash b \rangle\!\rangle(\langle\!\langle E \vdash A' \rangle\!\rangle) : \langle\!\langle E \vdash B\{X \leftarrow A'\} \rangle\!\rangle$, the translation of ϑ .

□

Epilogue

At the start of this book we reviewed some of the central concepts in object-oriented programming. We hope to have clarified them in the course of Parts I, II, and III. In this Epilogue, we summarize our main conclusions.

The setting for our work has been a family of object calculi. These calculi have a tiny syntax with few orthogonal constructs, but they are remarkably expressive. The calculi have both functional and imperative operational semantics; these semantics correspond well to the behavior of common languages, yet are simple enough for use in proofs. In these respects, object calculi are analogous to λ -calculi. Unlike λ -calculi, however, object calculi are patently object-oriented, and enable us to express common object-oriented examples directly and naturally.

In our calculi, objects are primitive but classes are not. Classes can be encoded in terms of objects, easily and faithfully. The encoding is consistent with our typing rules. In fact, the subtyping rules serve in justifying inheritance, even in situations where inheritance is decoupled from subtyping (for example, in the presence of binary methods). Moreover, the encoding of classes lends itself to several variants; for instance, it can account for protected and private methods. As a result of this encoding, we obtain a satisfactory reduction of class-based languages to the simpler object-based languages. To our knowledge, such a reduction had never been defined precisely, despite the existence of many suggestive programming techniques and examples.

In addition to the encoding of classes, we found a translation from λ -calculi to object calculi that gives a simple method for expressing functions as objects. Thus we were able to reduce procedural languages to object-based languages. This reduction lends weight to the informal thesis that “everything is an object”, which is at the center of the Smalltalk philosophy. To our knowledge, this reduction had never been defined precisely either.

We have also considered the converse reduction, both by giving a denotational model for objects and by giving a translation of objects into a standard λ -calculus. In these interpretations, objects are records and methods are functions. The semantics of object types combines some intuitive ingredients (recursion, data abstraction) but is rather intricate, essentially because of the difficulty in accounting for object subtyping. Therefore the interpretations do not provide practical reductions of object-oriented programming to procedural programming. We conclude that it is best to accept objects as primitive, rather than try to explain them away.

In developing our calculi, we put emphasis on their typing rules and on the problem of type safety. One reason for this emphasis is that types serve to organize our thinking about objects. Another reason is that peculiar type systems are an important part of many object-oriented languages, and that these type systems are often unsafe.

In the course of the book, we formulated several type systems for our calculi and proved their soundness. We also defined three small but interesting programming languages O-1, O-2, and O-3, and we proved the soundness of their rules by translating them into calculi. This approach to proving soundness seems manageable and general.

Our investigation of typing rules has illuminated some delicate issues. One of those issues is the treatment of Self types. We found that a combination of simple object types, recursive types, and higher-order types can serve to explain Self types. The resulting rules for Self types are sound and prove useful for subtyping and inheritance. In particular, these rules permit the inheritance of methods that return self, without global flow analysis, dynamic typing, or retypechecking.

In our experience, object calculi are a good basis for studying object-oriented languages and for designing new ones. We used our calculi for guiding the design of O-1, O-2, and O-3, and for explaining and comparing their features. These languages include both class-based and object-based constructs. The inheritance mechanisms differ from language to language, and correspond, respectively, to those of the Simula family, the Eiffel family, and the emerging PolyTOIL/Theta family. O-1 is simple and convenient, but requires the use of typecase on a fairly regular basis. O-2 reduces the need for typecase by the introduction of Self types. Unfortunately, O-2 cannot deal well with binary methods. In contrast, O-3 can handle binary methods satisfactorily, without typecase. In order to handle binary methods, O-3 limits subsumption; one cannot always view instances of a subclass as instances of its superclass, as in O-1 and O-2. The trade-off between subsumption and inheritance of binary methods seems unavoidable. We do not know of a way of combining the advantages of O-1, O-2, and O-3 while eliminating all their limitations. The difficulty in reconciling the conflicting features of O-1, O-2, and O-3 is indicative of the difficulty of designing sound and flexible object-oriented languages.

In summary, we have found that objects are amenable to a precise, coherent understanding. As we have shown, we can cope with the complexity of object-oriented languages through the prism of object calculi. To temper our optimism, we should note that it is still hard to reason about objects. Our equational theories, though somewhat rudimentary, have brought into focus some interesting difficulties.

We do not expect our theory of objects to be definitive. Many variants and extensions are possible, and even desirable. In particular, our calculi do not yet address issues of concurrency, which seem worthy of study. We believe that our theory is a good starting point for further investigations, and that the development of object calculi will remain a fruitful approach to understanding object-oriented programming.

APPENDIX

Rules and Proofs

A FRAGMENTS

A.1 Simple-Objects Fragments

These are the typing and equational rules for simple objects.

$$\Delta_{\text{Ob}} \cup \Delta_{<:\text{Ob}}$$

(Type Object) (l_i distinct)

$$E \vdash B_i \quad \forall i \in 1..n$$

$$\frac{}{E \vdash [l_i; B_i]^{i \in 1..n}}$$

(Sub Object) (l_i distinct)

$$E \vdash B_i \quad \forall i \in 1..n+m$$

$$\frac{}{E \vdash [l_i; B_i]^{i \in 1..n+m} <: [l_i; B_i]^{i \in 1..n}}$$

(Val Object) (where $A \equiv [l_i; B_i]^{i \in 1..n}$)

$$E, x_i:A \vdash b_i : B_i \quad \forall i \in 1..n$$

$$\frac{}{E \vdash [l_i = \zeta(x_i:A)b_i]^{i \in 1..n} : A}$$

(Val Select)

$$E \vdash a : [l_i; B_i]^{i \in 1..n} \quad j \in 1..n$$

$$\frac{}{E \vdash a.l_j : B_j}$$

(Val Update) (where $A \equiv [l_i; B_i]^{i \in 1..n}$)

$$E \vdash a : A \quad E, x:A \vdash b : B_j \quad j \in 1..n$$

$$\frac{}{E \vdash a.l_j = \zeta(x:A)b : A}$$

$$\Delta_{=\text{Ob}} \cup \Delta_{=:\text{Ob}}$$

(Eq Object) (where $A \equiv [l_i; B_i]^{i \in 1..n}$)

$$E, x_i:A \vdash b_i \leftrightarrow b'_i : B_i \quad \forall i \in 1..n$$

$$\frac{}{E \vdash [l_i = \zeta(x_i:A)b_i]^{i \in 1..n} \leftrightarrow [l_i = \zeta(x_i:A)b'_i]^{i \in 1..n} : A}$$

(Eq Sub Object) (where $A \equiv [l_i; B_i]^{i \in 1..n}$, $A' \equiv [l_i; B_i]^{i \in 1..n+m}$)

$$E, x_i:A \vdash b_i : B_i \quad \forall i \in 1..n \quad E, x_i:A' \vdash b_j : B_j \quad \forall j \in n+1..n+m$$

$$\frac{}{E \vdash [l_i = \zeta(x_i:A)b_i]^{i \in 1..n} \leftrightarrow [l_i = \zeta(x_i:A')b_i]^{i \in 1..n+m} : A}$$

$\frac{(\text{Eq Select})}{E \vdash a \leftrightarrow a' : [l_i : B_i]^{i \in 1..n} \quad j \in 1..n}$ $\frac{}{E \vdash a.l_j \leftrightarrow a'.l_j : B_j}$	$\frac{(\text{Eq Update}) \quad (\text{where } A \equiv [l_i : B_i]^{i \in 1..n})}{E \vdash a \leftrightarrow a' : A \quad E, x:A \vdash b \leftrightarrow b' : B_j \quad j \in 1..n}$ $\frac{}{E \vdash a.l_j \not\equiv \zeta(x:A)b \leftrightarrow a'.l_j \not\equiv \zeta(x:A)b' : A}$
$(\text{Eval Select}) \quad (\text{where } A \equiv [l_i : B_i]^{i \in 1..n}, \quad a \equiv [l_i = \zeta(x_i : A')b_i]_{i \in 1..n+m})$ $\frac{E \vdash a : A \quad j \in 1..n}{E \vdash a.l_j \leftrightarrow b_j \{a\} : B_j}$	
$(\text{Eval Update}) \quad (\text{where } A \equiv [l_i : B_i]^{i \in 1..n}, \quad a \equiv [l_i = \zeta(x_i : A')b_i]^{i \in 1..n+m})$ $\frac{E \vdash a : A \quad E, x:A \vdash b : B_j \quad j \in 1..n}{E \vdash a.l_j \not\equiv \zeta(x:A)b \leftrightarrow [l_j = \zeta(x : A')b, l_i = \zeta(x_i : A')b_i]^{i \in (1..n+m) - \{j\}} : A}$	

A.2 Other Typing Fragments

These are other typing fragments for first-order and second-order calculi.

Δ_x

$\frac{(\text{Env } \emptyset)}{\emptyset \vdash \diamond}$	$\frac{(\text{Env } x)}{E \vdash A \quad x \notin \text{dom}(E)}$	$\frac{(\text{Val } x)}{E', x:A, E'' \vdash \diamond}$
$\emptyset \vdash \diamond$	$E, x:A \vdash \diamond$	$E', x:A, E'' \vdash x : A$

Δ_K

(Type Const)

$$\frac{}{E \vdash \diamond}$$

$$\frac{}{E \vdash K}$$

Δ_{\rightarrow}

(Type Arrow)	(Val Fun)	(Val Appl)
$E \vdash A \quad E \vdash B$	$E, x:A \vdash b : B$	$E \vdash b : A \rightarrow B \quad E \vdash a : A$
$E \vdash A \rightarrow B$	$E \vdash \lambda(x:A)b : A \rightarrow B$	$E \vdash b(a) : B$

Δ_{\leq}

(Sub Refl)	(Sub Trans)	(Val Subsumption)
$E \vdash A$	$E \vdash A <: B \quad E \vdash B <: C$	$E \vdash a : A \quad E \vdash A <: B$
$E \vdash A <: A$	$E \vdash A <: C$	$E \vdash a : B$

$$\frac{\begin{array}{c} (\text{Type Top}) \\ E \vdash \diamond \end{array}}{E \vdash \text{Top}} \quad \frac{\begin{array}{c} (\text{Sub Top}) \\ E \vdash A \end{array}}{E \vdash A <: \text{Top}}$$

 $\Delta_{<:\rightarrow}$

$$\frac{\begin{array}{c} (\text{Sub Arrow}) \\ E \vdash A' <: A \quad E \vdash B <: B' \end{array}}{E \vdash A \rightarrow B <: A' \rightarrow B'}$$

 Δ_X

$$\frac{\begin{array}{c} (\text{Env } X) \\ E \vdash \diamond \quad X \notin \text{dom}(E) \end{array}}{E, X \vdash \diamond} \quad \frac{\begin{array}{c} (\text{Type } X) \\ E', X, E'' \vdash \diamond \end{array}}{E', X, E'' \vdash X}$$

 $\Delta_{<:X}$

$$\frac{\begin{array}{c} (\text{Env } X <:) \\ E \vdash A \quad X \notin \text{dom}(E) \end{array}}{E, X <: A \vdash \diamond} \quad \frac{\begin{array}{c} (\text{Type } X <:) \\ E', X <: A, E'' \vdash \diamond \end{array}}{E', X <: A, E'' \vdash X} \quad \frac{\begin{array}{c} (\text{Sub } X) \\ E', X <: A, E'' \vdash \diamond \end{array}}{E', X <: A, E'' \vdash X <: A}$$

 Δ_μ

$$\frac{\begin{array}{c} (\text{Type Rec}) \\ E, X \vdash A \end{array}}{E \vdash \mu(X)A}$$

$$\frac{\begin{array}{c} (\text{Val Fold}) \quad (\text{where } A \equiv \mu(X)B\{X\}) \\ E \vdash b : B\{A\} \end{array}}{E \vdash \text{fold}(A,b) : A} \quad \frac{\begin{array}{c} (\text{Val Unfold}) \quad (\text{where } A \equiv \mu(X)B\{X\}) \\ E \vdash a : A \end{array}}{E \vdash \text{unfold}(a) : B\{A\}}$$

 $\Delta_{<:\mu}$

$$\frac{\begin{array}{c} (\text{Type Rec-} <) \\ E, X <: \text{Top} \vdash A \end{array}}{E \vdash \mu(X)A} \quad \frac{\begin{array}{c} (\text{Sub Rec}) \\ E \vdash \mu(X)A \quad E \vdash \mu(Y)B \quad E, Y <: \text{Top}, X <: Y \vdash A <: B \end{array}}{E \vdash \mu(X)A <: \mu(Y)B}$$

(Val Fold) (where $A \equiv \mu(X)B\{X\}$)

$$E \vdash b : B\{A\}$$

$$\frac{}{E \vdash fold(A,b) : A}$$

(Val Unfold) (where $A \equiv \mu(X)B\{X\}$)

$$E \vdash a : A$$

$$\frac{}{E \vdash unfold(a) : B\{A\}}$$

 Δ_V

(Type All)

$$E, X \vdash B$$

$$\frac{}{E \vdash \forall(X)B}$$

(Val Fun2)

$$E, X \vdash b : B$$

$$\frac{}{E \vdash \lambda(X)b : \forall(X)B}$$

(Val Appl2)

$$E \vdash b : \forall(X)B\{X\} \quad E \vdash A$$

$$\frac{}{E \vdash b(A) : B\{A\}}$$

 $\Delta_{<:V}$ (Type All $<:$)

$$E, X <: A \vdash B$$

$$\frac{}{E \vdash \forall(X <: A)B}$$

(Sub All)

$$E \vdash A' <: A \quad E, X <: A' \vdash B <: B'$$

$$\frac{}{E \vdash \forall(X <: A)B <: \forall(X <: A')B'}$$

(Val Fun2 $<:$)

$$E, X <: A \vdash b : B$$

$$\frac{}{E \vdash \lambda(X <: A)b : \forall(X <: A)B}$$

(Val Appl2 $<:$)

$$E \vdash b : \forall(X <: A)B\{X\} \quad E \vdash A' <: A$$

$$\frac{}{E \vdash b(A') : B\{A'\}}$$

 Δ_E

(Type Exists)

$$E, X \vdash B$$

$$\frac{}{E \vdash \exists(X)B}$$

(Val Pack)

$$E \vdash A \quad E \vdash b\{A\} : B\{A\}$$

$$\frac{}{E \vdash pack X=A \text{ with } b\{X\}:B\{X\} : \exists(X)B\{X\}}$$

(Val Open)

$$E \vdash c : \exists(X)B \quad E \vdash D \quad E, X, x:B \vdash d : D$$

$$\frac{}{E \vdash open c \text{ as } X,x:B \text{ in } d:D : D}$$

 $\Delta_{<:\exists}$ (Type Exists $<:$)

$$E, X <: A \vdash B$$

$$\frac{}{E \vdash \exists(X <: A)B}$$

(Sub Exists)

$$E \vdash A <: A' \quad E, X <: A \vdash B <: B'$$

$$\frac{}{E \vdash \exists(X <: A)B <: \exists(X <: A')B'}$$

(Val Pack $<:$)

$$\frac{E \vdash C <: A \quad E \vdash b\{C\} : B\{C\}}{E \vdash \text{pack } X <: A = C \text{ with } b\{X\}:B\{X\} : \exists(X <: A)B\{X\}}$$

(Val Open $<:$)

$$\frac{\begin{array}{c} E \vdash c : \exists(X <: A)B \quad E \vdash D \quad E, X <: A, x:B \vdash d : D \\ \hline E \vdash \text{open } c \text{ as } X <: A, x:B \text{ in } d:D : D \end{array}}{}$$

A.3 Other Equational Fragments

These are other equational fragments for first-order and second-order calculi.

 $\Delta_{=}$

$$\frac{\begin{array}{c} (\text{Eq Symm}) \quad (\text{Eq Trans}) \\ E \vdash a \leftrightarrow b : A \quad E \vdash a \leftrightarrow b : A \quad E \vdash b \leftrightarrow c : A \\ \hline E \vdash b \leftrightarrow a : A \quad E \vdash a \leftrightarrow c : A \end{array}}{}$$

 $\Delta_{=x}$

$$\frac{\begin{array}{c} (\text{Eq } x) \\ E', x:A, E'' \vdash \diamond \\ \hline E', x:A, E'' \vdash x \leftrightarrow x : A \end{array}}{}$$

 $\Delta_{=\rightarrow}$

$$\frac{\begin{array}{c} (\text{Eq Fun}) \quad (\text{Eq Appl}) \\ E, x:A \vdash b \leftrightarrow b' : B \quad E \vdash b \leftrightarrow b' : A \rightarrow B \quad E \vdash a \leftrightarrow a' : A \\ \hline E \vdash \lambda(x:A)b \leftrightarrow \lambda(x:A)b' : A \rightarrow B \quad E \vdash b(a) \leftrightarrow b'(a') : B \end{array}}{}$$

$$\frac{\begin{array}{c} (\text{Eval Beta}) \quad (\text{Eval Eta}) \\ E \vdash \lambda(x:A)b\{x\} : A \rightarrow B \quad E \vdash a : A \quad E \vdash b : A \rightarrow B \quad x \notin \text{dom}(E) \\ \hline E \vdash (\lambda(x:A)b\{x\})(a) \leftrightarrow b\{a\} : B \quad E \vdash \lambda(x:A)b(x) \leftrightarrow b : A \rightarrow B \end{array}}{}$$

 $\Delta_{=<:}$

$$\frac{\begin{array}{c} (\text{Eq Subsumption}) \quad (\text{Eq Top}) \\ E \vdash a \leftrightarrow a' : A \quad E \vdash A <: B \quad E \vdash a : A \quad E \vdash b : B \\ \hline E \vdash a \leftrightarrow a' : B \quad E \vdash a \leftrightarrow b : \text{Top} \end{array}}{}$$

$\Delta_{=\mu}$ (same as $\Delta_{=<:\mu}$)

$$\frac{\text{(Eq Fold)} \quad (\text{where } A \equiv \mu(X)B\{X\}) \quad E \vdash b \leftrightarrow b' : B\{A\}}{E \vdash fold(A,b) \leftrightarrow fold(A,b') : A} \quad \frac{\text{(Eq Unfold)} \quad (\text{where } A \equiv \mu(X)B\{X\}) \quad E \vdash a \leftrightarrow a' : A}{E \vdash unfold(a) \leftrightarrow unfold(a') : B\{A\}}$$

$$\frac{\text{(Eval Fold)} \quad (\text{where } A \equiv \mu(X)B\{X\}) \quad E \vdash a : A}{E \vdash fold(A,unfold(a)) \leftrightarrow a : A} \quad \frac{\text{(Eval Unfold)} \quad (\text{where } A \equiv \mu(X)B\{X\}) \quad E \vdash b : B\{A\}}{E \vdash unfold(fold(A,b)) \leftrightarrow b : B\{A\}}$$

 $\Delta_{=\forall}$

$$\frac{\text{(Eq Fun2)} \quad E, X \vdash b \leftrightarrow b' : B}{E \vdash \lambda(X)b \leftrightarrow \lambda(X)b' : \forall(X)B} \quad \frac{\text{(Eq Appl2)} \quad E \vdash b \leftrightarrow b' : \forall(X)B\{X\} \quad E \vdash A}{E \vdash b(A) \leftrightarrow b'(A) : B\{A\}}$$

$$\frac{\text{(Eval Beta2)} \quad E \vdash \lambda(X)b\{X\} : \forall(X)B\{X\} \quad E \vdash A}{E \vdash (\lambda(X)b\{X\})(A) \leftrightarrow b\{A\} : B\{A\}} \quad \frac{\text{(Eval Eta2)} \quad E \vdash b : \forall(X)B \quad X \notin \text{dom}(E)}{E \vdash \lambda(X)b(X) \leftrightarrow b : \forall(X)B}$$

 $\Delta_{=<:\forall}$

$$\frac{\text{(Eq Fun2<:)} \quad E, X <: A \vdash b \leftrightarrow b' : B}{E \vdash \lambda(X <: A)b \leftrightarrow \lambda(X <: A)b' : \forall(X <: A)B} \quad \frac{\text{(Eq Appl2<:)} \quad E \vdash b \leftrightarrow b' : \forall(X <: A)B\{X\} \quad E \vdash A' <: A}{E \vdash b(A') \leftrightarrow b'(A') : B\{A'\}}$$

$$\frac{\text{(Eval Beta2<:)} \quad E \vdash \lambda(X <: A)b\{X\} : \forall(X <: A)B\{X\} \quad E \vdash C <: A}{E \vdash (\lambda(X <: A)b\{X\})(C) \leftrightarrow b\{C\} : B\{C\}} \quad \frac{\text{(Eval Eta2<:)} \quad E \vdash b : \forall(X <: A)B \quad X \notin \text{dom}(E)}{E \vdash \lambda(X <: A)b(X) \leftrightarrow b : \forall(X <: A)B}$$

 $\Delta_{=\exists}$

$$\frac{\text{(Eq Pack)} \quad E \vdash A \quad E \vdash b\{A\} \leftrightarrow b'\{A\} : B\{A\}}{E \vdash \text{pack } X=A \text{ with } b\{X\}:B\{X\} \leftrightarrow \text{pack } X=A \text{ with } b'\{X\}:B\{X\} : \exists(X)B\{X\}}$$

$$\frac{\text{(Eq Open)} \quad E \vdash c \leftrightarrow c' : \exists(X)B \quad E \vdash D \quad E, X, x:B \vdash d \leftrightarrow d' : D}{E \vdash \text{open } c \text{ as } X,x:B \text{ in } d:D \leftrightarrow \text{open } c' \text{ as } X,x:B \text{ in } d':D : D}$$

(Eval Unpack $<:$) (where $c \equiv \text{pack } X=C \text{ with } b\{X\}:B\{X\}$)

$$\frac{E \vdash c : \exists(X)B\{X\} \quad E \vdash D \quad E, X, x:B\{X\} \vdash d\{X,x\} : D}{E \vdash \text{open } c \text{ as } X, x:B\{X\} \text{ in } d\{X,x\}:D \leftrightarrow d[\![C,b\{C\}]\!]:D}$$

(Eval Repack $<:$)

$$\frac{E \vdash b : \exists(X)B\{X\} \quad E, y:\exists(X)B\{X\} \vdash d\{y\} : D}{E \vdash \text{open } b \text{ as } X, x:B\{X\} \text{ in } d[\!\![\text{pack } X'=X \text{ with } x:B\{X'\}]\!]:D \leftrightarrow d[\![b]\!]:D}$$

$\Delta_{=<:\exists}$

(Eq Pack $<:$)

$$\frac{E \vdash C <: A' \quad E \vdash A' <: A \quad E, X <: A' \vdash B'\{X\} <: B\{X\} \quad E \vdash b\{C\} \leftrightarrow b'\{C\} : B'\{C\}}{E \vdash \text{pack } X <: A = C \text{ with } b\{X\}:B\{X\} \leftrightarrow \text{pack } X <: A' = C \text{ with } b'\{X\}:B'\{X\} : \exists(X <: A)B\{X\}}$$

(Eq Open $<:$)

$$\frac{E \vdash c \leftrightarrow c' : \exists(X <: A)B \quad E \vdash D \quad E, X <: A, x:B \vdash d \leftrightarrow d' : D}{E \vdash \text{open } c \text{ as } X <: A, x:B \text{ in } d:D \leftrightarrow \text{open } c' \text{ as } X <: A, x:B \text{ in } d':D : D}$$

(Eval Unpack $<:$) (where $c \equiv \text{pack } X <: A = C \text{ with } b\{X\}:B\{X\}$)

$$\frac{E \vdash c : \exists(X <: A)B\{X\} \quad E \vdash D \quad E, X <: A, x:B\{X\} \vdash d\{X,x\} : D}{E \vdash \text{open } c \text{ as } X <: A, x:B\{X\} \text{ in } d\{X,x\}:D \leftrightarrow d[\![C,b\{C}\!]\!]:D}$$

(Eval Repack $<:$)

$$\frac{E \vdash b : \exists(X <: A)B\{X\} \quad E, y:\exists(X <: A)B\{X\} \vdash d\{y\} : D}{E \vdash \text{open } b \text{ as } X <: A, x:B\{X\} \text{ in } d[\!\![\text{pack } X' <: A = X \text{ with } x:B\{X'\}]\!]:D \leftrightarrow d[\![b]\!]:D}$$

B SYSTEMS

B.1 The Ob_{\leq} Calculus

This is the first-order calculus with objects and subtyping.

Syntax of the Ob_{\leq} calculus

$A, B ::=$	types
K	ground type
Top	the biggest type
$[l_i : B_i]^{i \in 1..n}$	object type (l_i distinct)
$a, b ::=$	terms
x	variable
$[l_i = \zeta(x_i : A_i) b_i]^{i \in 1..n}$	object (l_i distinct)
$a.l$	method invocation
$a.l \Leftarrow \zeta(x : A)b$	method update

The fragments forming this calculus have been rearranged, so that the rules are grouped by judgment.

$$\frac{(\text{Env } \emptyset) \quad (\text{Env } x)}{\emptyset \vdash \diamond \quad \frac{E \vdash A \quad x \notin \text{dom}(E)}{E, x:A \vdash \diamond}}$$

$$\frac{(\text{Type Const}) \quad (\text{Type Top}) \quad (\text{Type Object}) \quad (l_i \text{ distinct})}{\begin{array}{c} E \vdash \diamond \\ E \vdash K \\ E \vdash \diamond \\ E \vdash \text{Top} \end{array} \quad \begin{array}{c} E \vdash \diamond \\ E \vdash B_i \quad \forall i \in 1..n \end{array} \quad \begin{array}{c} E \vdash [l_i : B_i]^{i \in 1..n} \end{array}}$$

$$\frac{(\text{Sub Refl}) \quad (\text{Sub Trans})}{\begin{array}{c} E \vdash A \\ E \vdash A <: A \end{array} \quad \begin{array}{c} E \vdash A <: B \quad E \vdash B <: C \\ E \vdash A <: C \end{array}}$$

<p>(Sub Top)</p> $\frac{E \vdash A}{E \vdash A <: \text{Top}}$	<p>(Sub Object) (l_i distinct)</p> $\frac{\begin{array}{c} E \vdash B_i \\ \forall i \in 1..n+m \end{array}}{E \vdash [l_i : B_i]^{i \in 1..n+m} <: [l_i : B_i]^{i \in 1..n}}$
<hr/>	
<p>(Val Subsumption)</p> $\frac{E \vdash a : A \quad E \vdash A <: B}{E \vdash a : B}$	<p>(Val x)</p> $\frac{E', x:A, E'' \vdash \diamond}{E', x:A, E'' \vdash x : A}$
<hr/>	
<p>(Val Object) (where $A \equiv [l_i : B_i]^{i \in 1..n}$)</p> $\frac{E, x_i : A \vdash b_i : B_i \quad \forall i \in 1..n}{E \vdash [l_i = \zeta(x_i : A) b_i]^{i \in 1..n} : A}$	
<p>(Val Select)</p> $\frac{E \vdash a : [l_i : B_i]^{i \in 1..n} \quad j \in 1..n}{E \vdash a.l_j : B_j}$	<p>(Val Update) (where $A \equiv [l_i : B_i]^{i \in 1..n}$)</p> $\frac{E \vdash a : A \quad E, x : A \vdash b : B_j \quad j \in 1..n}{E \vdash a.l_j \neq \zeta(x : A) b : A}$
<hr/>	
<p>(Eq Symm)</p> $\frac{E \vdash a \leftrightarrow b : A}{E \vdash b \leftrightarrow a : A}$	<p>(Eq Trans)</p> $\frac{E \vdash a \leftrightarrow b : A \quad E \vdash b \leftrightarrow c : A}{E \vdash a \leftrightarrow c : A}$
<p>(Eq Subsumption)</p> $\frac{E \vdash a \leftrightarrow a' : A \quad E \vdash A <: B}{E \vdash a \leftrightarrow a' : B}$	<p>(Eq Top)</p> $\frac{E \vdash a : A \quad E \vdash b : B}{E \vdash a \leftrightarrow b : \text{Top}}$
<hr/>	
<p>(Eq Object) (where $A \equiv [l_i : B_i]^{i \in 1..n}$)</p> $\frac{E, x_i : A \vdash b_i \leftrightarrow b'_i : B_i \quad \forall i \in 1..n}{E \vdash [l_i = \zeta(x_i : A) b_i]^{i \in 1..n} \leftrightarrow [l_i = \zeta(x_i : A) b'_i]^{i \in 1..n} : A}$	<p>(Eq x)</p> $\frac{E', x : A, E'' \vdash \diamond}{E', x : A, E'' \vdash x \leftrightarrow x : A}$
<hr/>	
<p>(Eq Sub Object) (where $A \equiv [l_i : B_i]^{i \in 1..n}$, $A' \equiv [l_i : B_i]^{i \in 1..n+m}$)</p> $\frac{E, x_i : A \vdash b_i : B_i \quad \forall i \in 1..n \quad E, x_j : A' \vdash b_j : B_j \quad \forall j \in n+1..n+m}{E \vdash [l_i = \zeta(x_i : A) b_i]^{i \in 1..n} \leftrightarrow [l_i = \zeta(x_i : A') b_i]^{i \in 1..n+m} : A}$	
<p>(Eq Select)</p> $\frac{E \vdash a \leftrightarrow a' : [l_i : B_i]^{i \in 1..n} \quad j \in 1..n}{E \vdash a.l_j \leftrightarrow a'.l_j : B_j}$	<p>(Eq Update) (where $A \equiv [l_i : B_i]^{i \in 1..n}$)</p> $\frac{E \vdash a \leftrightarrow a' : A \quad E, x : A \vdash b \leftrightarrow b' : B_j \quad j \in 1..n}{E \vdash a.l_j \neq \zeta(x : A) b \leftrightarrow a'.l_j \neq \zeta(x : A) b' : A}$

(Eval Select) (where $A \equiv [l_i:B_i]_{i \in 1..n}$, $a \equiv [l_i=\zeta(x_i:A')b_i]_{i \in 1..n+m}$)

$$\frac{E \vdash a : A \quad j \in 1..n}{E \vdash a.l_j \leftrightarrow b_j[a] : B_j}$$

(Eval Update) (where $A \equiv [l_i:B_i]_{i \in 1..n}$, $a \equiv [l_i=\zeta(x_i:A')b_i]_{i \in 1..n+m}$)

$$\frac{E \vdash a : A \quad E, x:A \vdash b : B_j \quad j \in 1..n}{E \vdash a.l_j=\zeta(x:A)b \leftrightarrow [l_i=\zeta(x_i:A')b_i]_{i \in (1..n+m)-\{j\}} : A}$$

B.2 The $F_{<:\mu}$ Calculus

This is the second-order calculus with subtyping and recursion. Objects may be added, using the same rules as for $\text{Ob}_{1<}$ (see also Section 13.2).

Syntax of the $F_{<:\mu}$ calculus

$A, B, C, D ::=$	types
X	type variable
Top	the biggest type
$A \rightarrow B$	function type
$\mu(X)A$	recursive type
$\forall(X <: A)B$	bounded universal type
$\exists(X <: A)B$	bounded existential type
$a, b, c, d ::=$	terms
x	variable
$\lambda(x:A)b$	function
$b(a)$	application
$\text{fold}(A, a)$	recursive fold
$\text{unfold}(a)$	recursive unfold
$\lambda(X <: A)b$	type abstraction
$b(A)$	type application
$\text{pack } X <: A = C \text{ with } b : B$	data abstraction packaging
$\text{open } c \text{ as } X <: A, x : B \text{ in } d : D$	data abstraction opening

The rules are grouped by judgment.

$$\frac{\begin{array}{c} (\text{Env } \emptyset) \quad (\text{Env } x) \\ \hline \emptyset \vdash \diamond \end{array} \quad \begin{array}{c} (\text{Env } x) \\ E \vdash A \quad x \notin \text{dom}(E) \\ \hline E, x:A \vdash \diamond \end{array} \quad \begin{array}{c} (\text{Env } X <) \\ E \vdash A \quad X \notin \text{dom}(E) \\ \hline E, X <: A \vdash \diamond \end{array}}{E, x:A \vdash \diamond}$$

$$\frac{\begin{array}{c} (\text{Type } X<:) \\ E', X<:A, E'' \vdash \diamond \end{array}}{E', X<:A, E'' \vdash X} \quad \frac{\begin{array}{c} (\text{Type Top}) \\ E \vdash \diamond \end{array}}{E \vdash \text{Top}} \quad \frac{\begin{array}{c} (\text{Type Arrow}) \\ E \vdash A \quad E \vdash B \end{array}}{E \vdash A \rightarrow B}$$

$$\frac{\begin{array}{c} (\text{Type Rec-} <:) \\ E, X<:\text{Top} \vdash A \end{array}}{E \vdash \mu(X)A} \quad \frac{\begin{array}{c} (\text{Type All-} <:) \\ E, X<:A \vdash B \end{array}}{E \vdash \forall(X<:A)B} \quad \frac{\begin{array}{c} (\text{Type Exists-} <:) \\ E, X<:A \vdash B \end{array}}{E \vdash \exists(X<:A)B}$$

$$\frac{\begin{array}{c} (\text{Sub Refl}) \\ E \vdash A \end{array}}{E \vdash A <: A} \quad \frac{\begin{array}{c} (\text{Sub Trans}) \\ E \vdash A <: B \quad E \vdash B <: C \end{array}}{E \vdash A <: C}$$

$$\frac{\begin{array}{c} (\text{Sub X}) \\ E', X<:A, E'' \vdash \diamond \end{array}}{E', X<:A, E'' \vdash X <: A} \quad \frac{\begin{array}{c} (\text{Sub Top}) \\ E \vdash A \end{array}}{E \vdash A <: \text{Top}} \quad \frac{\begin{array}{c} (\text{Sub Arrow}) \\ E \vdash A' <: A \quad E \vdash B <: B' \end{array}}{E \vdash A \rightarrow B <: A' \rightarrow B'}$$

$$\frac{\begin{array}{c} (\text{Sub Rec}) \\ E \vdash \mu(X)A \quad E \vdash \mu(Y)B \quad E, Y<:\text{Top}, X<:Y \vdash A <: B \end{array}}{E \vdash \mu(X)A <: \mu(Y)B}$$

$$\frac{\begin{array}{c} (\text{Sub All}) \\ E \vdash A' <: A \quad E, X<:A' \vdash B <: B' \end{array}}{E \vdash \forall(X<:A)B <: \forall(X<:A')B'} \quad \frac{\begin{array}{c} (\text{Sub Exists}) \\ E \vdash A <: A' \quad E, X<:A \vdash B <: B' \end{array}}{E \vdash \exists(X<:A)B <: \exists(X<:A')B'}$$

$$\frac{\begin{array}{c} (\text{Val Subsumption}) \\ E \vdash a : A \quad E \vdash A <: B \end{array}}{E \vdash a : B} \quad \frac{\begin{array}{c} (\text{Val } x) \\ E', x:A, E'' \vdash \diamond \end{array}}{E', x:A, E'' \vdash x : A}$$

$$\frac{\begin{array}{c} (\text{Val Fun}) \\ E, x:A \vdash b : B \end{array}}{E \vdash \lambda(x:A)b : A \rightarrow B} \quad \frac{\begin{array}{c} (\text{Val Appl}) \\ E \vdash b : A \rightarrow B \quad E \vdash a : A \end{array}}{E \vdash b(a) : B}$$

$$\frac{\begin{array}{c} (\text{Val Fold}) \quad (\text{where } A \equiv \mu(X)B\{X\}) \\ E \vdash b : B\{A\} \end{array}}{E \vdash \text{fold}(A,b) : A} \quad \frac{\begin{array}{c} (\text{Val Unfold}) \quad (\text{where } A \equiv \mu(X)B\{X\}) \\ E \vdash a : A \end{array}}{E \vdash \text{unfold}(a) : B\{A\}}$$

(Val Fun2<:)	$E, X <: A \vdash b : B$	(Val Appl2<:)	$E \vdash b : \forall(X <: A)B\{X\}$	$E \vdash A' <: A$	$E \vdash b(A') : B\{A'\}$
$E \vdash \lambda(X <: A)b : \forall(X <: A)B$					
(Val Pack<:)	$E \vdash C <: A \quad E \vdash b\{C\} : B\{C\}$				
$E \vdash pack X <: A = C \text{ with } b\{X\}:B\{X\} : \exists(X <: A)B\{X\}$					
(Val Open<:)	$E \vdash c : \exists(X <: A)B \quad E \vdash D \quad E, X <: A, x:B \vdash d : D$				
$E \vdash open c \text{ as } X <: A, x:B \text{ in } d:D : D$					
(Eq Symm)	$E \vdash a \leftrightarrow b : A$	(Eq Trans)	$E \vdash a \leftrightarrow b : A \quad E \vdash b \leftrightarrow c : A$		
$E \vdash b \leftrightarrow a : A$			$E \vdash a \leftrightarrow c : A$		
(Eq Subsumption)	$E \vdash a \leftrightarrow a' : A \quad E \vdash A <: B$	(Eq Top)	$E \vdash a : A \quad E \vdash b : B$	(Eq x)	$E', x:A, E'' \vdash \diamond$
$E \vdash a \leftrightarrow a' : B$		$E \vdash a \leftrightarrow b : Top$		$E', x:A, E'' \vdash x \leftrightarrow x : A$	
(Eq Fun)	$E, x:A \vdash b \leftrightarrow b' : B$	(Eq Appl)	$E \vdash b \leftrightarrow b' : A \rightarrow B \quad E \vdash a \leftrightarrow a' : A$		
$E \vdash \lambda(x:A)b \leftrightarrow \lambda(x:A)b' : A \rightarrow B$			$E \vdash b(a) \leftrightarrow b'(a') : B$		
(Eq Fold)	$E \vdash b \leftrightarrow b' : B\{A\}$	(Eq Unfold)	$E \vdash a \leftrightarrow a' : A$		
$E \vdash fold(A,b) \leftrightarrow fold(A,b') : A$			$E \vdash unfold(a) \leftrightarrow unfold(a') : B\{A\}$		
(Eq Fun2<:)	$E, X <: A \vdash b \leftrightarrow b' : B$	(Eq Appl2<:)	$E \vdash b \leftrightarrow b' : \forall(X <: A)B\{X\}$	$E \vdash A' <: A$	$E \vdash b(A') \leftrightarrow b'(A') : B\{A'\}$
$E \vdash \lambda(X <: A)b \leftrightarrow \lambda(X <: A)b' : \forall(X <: A)B$					
(Eq Pack<:)	$E \vdash C <: A' \quad E \vdash A' <: A \quad E, X <: A' \vdash B'\{X\} <: B\{X\} \quad E \vdash b\{C\} \leftrightarrow b'\{C\} : B'\{C\}$				
$E \vdash pack X <: A = C \text{ with } b\{X\}:B\{X\} \leftrightarrow pack X <: A' = C \text{ with } b'\{X\}:B'\{X\} : \exists(X <: A)B\{X\}$					

(Eq Open $\langle:\rangle$)

$$E \vdash c \leftrightarrow c' : \exists(X \langle : A)B \quad E \vdash D \quad E, X \langle : A, x:B \vdash d \leftrightarrow d' : D$$

$$\boxed{E \vdash \text{open } c \text{ as } X \langle : A, x:B \text{ in } d:D \leftrightarrow \text{open } c' \text{ as } X \langle : A, x:B \text{ in } d':D : D}$$

(Eval Beta)

$$E \vdash \lambda(x:A)b\{x\} : A \rightarrow B \quad E \vdash a : A$$

$$\boxed{E \vdash (\lambda(x:A)b\{x\})(a) \leftrightarrow b\{a\} : B}$$

(Eval Eta)

$$E \vdash b : A \rightarrow B \quad x \notin \text{dom}(E)$$

$$\boxed{E \vdash \lambda(x:A)b(x) \leftrightarrow b : A \rightarrow B}$$

(Eval Fold) (where $A \equiv \mu(X)B\{X\}$)

$$E \vdash a : A$$

$$\boxed{E \vdash \text{fold}(A, \text{unfold}(a)) \leftrightarrow a : A}$$

(Eval Unfold) (where $A \equiv \mu(X)B\{X\}$)

$$E \vdash b : B\{A\}$$

$$\boxed{E \vdash \text{unfold}(\text{fold}(A, b)) \leftrightarrow b : B\{A\}}$$

(Eval Beta2 $\langle:\rangle$)

$$E \vdash \lambda(X \langle : A)b\{X\} : \forall(X \langle : A)B\{X\} \quad E \vdash C \langle : A$$

$$\boxed{E \vdash (\lambda(X \langle : A)b\{X\})(C) \leftrightarrow b\{C\} : B\{C\}}$$

(Eval Eta2 $\langle:\rangle$)

$$E \vdash b : \forall(X \langle : A)B \quad X \notin \text{dom}(E)$$

$$\boxed{E \vdash \lambda(X \langle : A)b(X) \leftrightarrow b : \forall(X \langle : A)B}$$

(Eval Unpack $\langle:\rangle$) (where $c \equiv \text{pack } X \langle : A = C \text{ with } b\{X\}:B\{X\}$)

$$E \vdash c : \exists(X \langle : A)B\{X\} \quad E \vdash D \quad E, X \langle : A, x:B\{X\} \vdash d\{X, x\} : D$$

$$\boxed{E \vdash \text{open } c \text{ as } X \langle : A, x:B\{X\} \text{ in } d\{X, x\}:D \leftrightarrow d\{C, b\{C\}\} : D}$$

(Eval Repack $\langle:\rangle$)

$$E \vdash b : \exists(X \langle : A)B\{X\} \quad E, y : \exists(X \langle : A)B\{X\} \vdash d\{y\} : D$$

$$\boxed{E \vdash \text{open } b \text{ as } X \langle : A, x:B\{X\} \text{ in } d[\text{pack } X' \langle : A = X \text{ with } x:B\{X'\}]:D \leftrightarrow d\{b\} : D}$$

B.3 The ζ Ob Calculus

This is the second-order calculus with object types and the Self quantifier.

Syntax of the ζ Ob calculus

$$A, B ::=$$

$$X$$

$$\text{Top}$$

$$[l_i; B_i]^{i \in 1..n}$$

$$\zeta(X)B$$

types

type variable

the biggest type

object type (l_i distinct)

Self quantified type

$$a, b ::=$$

$$x$$

$$[l_i = \zeta(x_i; A_i); b_i]^{i \in 1..n}$$

terms

variable

object (l_i distinct)

<i>a.l</i>	method invocation
<i>a.l</i> $\Leftarrow \zeta(x)b$	method update
<i>wrap(X <: A = B) b</i>	wrap
<i>use a as X <: A, y : B in b : D</i>	use

The rules are grouped by judgment.

$$\frac{(\text{Env } \emptyset) \quad (\text{Env } x) \quad (\text{Env } X <:) \quad \phi \vdash \diamond}{\phi \vdash \diamond \quad E \vdash A \quad x \notin \text{dom}(E) \quad E \vdash A \quad X \notin \text{dom}(E) \quad E, x:A \vdash \diamond \quad E, X <:A \vdash \diamond}$$

$$\frac{(\text{Type } X <) \quad E', X <: A, E'' \vdash \diamond \quad (\text{Type Top}) \quad E \vdash \diamond}{E', X <: A, E'' \vdash X \quad E \vdash \text{Top}}$$

$$\frac{(\text{Type Object}) \quad (l_i \text{ distinct}) \quad E \vdash B_i \quad \forall i \in 1..n \quad (\text{Type Self}) \quad E, X <: \text{Top} \vdash B}{E \vdash [l_i; B_i]_{i \in 1..n} \quad E \vdash \zeta(X)B}$$

$$\frac{(\text{Sub Refl}) \quad (\text{Sub Trans}) \quad E \vdash A \quad E \vdash A <: B \quad E \vdash B <: C}{E \vdash A <: A \quad E \vdash A <: C}$$

$$\frac{(\text{Sub X}) \quad (\text{Sub Top}) \quad E', X <: A, E'' \vdash \diamond \quad E \vdash A}{E', X <: A, E'' \vdash X <: A \quad E \vdash A <: \text{Top}}$$

$$\frac{(\text{Sub Object}) \quad (l_i \text{ distinct}) \quad E \vdash B_i \quad \forall i \in 1..n+m \quad (\text{Sub Self}) \quad E, X <: \text{Top} \vdash B <: B' \quad E \vdash [l_i; B_i]_{i \in 1..n+m} <: [l_i; B_i]_{i \in 1..n} \quad E \vdash \zeta(X)B <: \zeta(X)B'}{E \vdash \zeta(X)B <: \zeta(X)B'}$$

$$\frac{(\text{Val Subsumption}) \quad (\text{Val } x) \quad E \vdash a : A \quad E \vdash A <: B \quad E' \vdash a : B \quad E' \vdash x : A \quad E' \vdash x : A \quad E' \vdash x : A}{E' \vdash x : A}$$

(Val Object) (where $A \equiv [l_i; B_i]^{i \in 1..n}]$)

$$E, x_i:A \vdash b_i : B_i \quad \forall i \in 1..n$$

$$\frac{}{E \vdash [l_i = \zeta(x_i; A) b_i]^{i \in 1..n} : A}$$

(Val Select)

$$E \vdash a : [l_j; B_j]^{j \in 1..n}$$

$$\frac{}{E \vdash a.l_j : B_j}$$

(Val Update) (where $A \equiv [l_i; B_i]^{i \in 1..n}]$)

$$E \vdash a : A \quad E, x:A \vdash b : B_j \quad j \in 1..n$$

$$\frac{}{E \vdash a.l_j = \zeta(x; A) b : A}$$

(Val Wrap) (where $A \equiv \zeta(X)B\{X\}$)

$$E \vdash C <: A \quad E \vdash b[C] : B[C]$$

$$\frac{}{E \vdash \text{wrap}(Y <: A = C) b[Y] : A}$$

(Val Use) (where $A \equiv \zeta(X)B\{X\}$)

$$E \vdash c : A \quad E \vdash D \quad E, Y <: A, y:B[Y] \vdash d : D$$

$$\frac{}{E \vdash \text{use } c \text{ as } Y <: A, y:B[Y] \text{ in } d : D : D}$$

(Eq Symm)

$$E \vdash a \leftrightarrow b : A$$

(Eq Trans)

$$E \vdash a \leftrightarrow b : A \quad E \vdash b \leftrightarrow c : A$$

$$\frac{}{E \vdash b \leftrightarrow a : A}$$

$$\frac{}{E \vdash a \leftrightarrow c : A}$$

(Eq Subsumption)

$$E \vdash a \leftrightarrow a' : A$$

(Eq Top)

$$E \vdash A <: B$$

(Eq x)

$$E \vdash a : A \quad E \vdash b : B$$

(Eq Top)

$$E', x:A, E'' \vdash \diamond$$

$$\frac{}{E \vdash a \leftrightarrow a' : B}$$

$$\frac{}{E \vdash a \leftrightarrow b : \text{Top}}$$

$$\frac{}{E', x:A, E'' \vdash x \leftrightarrow x : A}$$

(Eq Object) (where $A \equiv [l_i; B_i]^{i \in 1..n}]$)

$$E, x_i:A \vdash b_i \leftrightarrow b'_i : B_i \quad \forall i \in 1..n$$

$$\frac{}{E \vdash [l_i = \zeta(x_i; A) b_i]^{i \in 1..n} \leftrightarrow [l_i = \zeta(x_i; A) b'_i]^{i \in 1..n} : A}$$

(Eq Sub Object) (where $A \equiv [l_i; B_i]^{i \in 1..n}], A' \equiv [l_i; B_i]^{i \in 1..n+m}]$)

$$E, x_i:A \vdash b_i : B_i \quad \forall i \in 1..n$$

$$E, x_j:A' \vdash b_j : B_j \quad \forall j \in n+1..n+m$$

$$\frac{}{E \vdash [l_i = \zeta(x_i; A) b_i]^{i \in 1..n} \leftrightarrow [l_i = \zeta(x_i; A') b_i]^{i \in 1..n+m} : A}$$

(Eq Select)

$$E \vdash a \leftrightarrow a' : [l_j; B_j]^{j \in 1..n}$$

(Eq Update) (where $A \equiv [l_i; B_i]^{i \in 1..n}]$)

$$j \in 1..n$$

$$\frac{}{E \vdash a.l_j \leftrightarrow a'.l_j : B_j}$$

$$E \vdash a \leftrightarrow a' : A \quad E, x:A \vdash b \leftrightarrow b' : B_j \quad j \in 1..n$$

$$\frac{}{E \vdash a.l_j = \zeta(x; A) b \leftrightarrow a'.l_j = \zeta(x; A) b' : A}$$

(Eq Wrap) (where $A \equiv \zeta(X)B\{X\}, A' \equiv \zeta(X)B'\{X\}$)

$$E \vdash C <: A'$$

$$E \vdash A \quad E, Y \vdash B'\{Y\} <: B\{Y\}$$

$$E \vdash b[C] \leftrightarrow b'[C] : B'\{C\}$$

$$E \vdash b'[C] : B'\{C\}$$

$$\frac{}{E \vdash \text{wrap}(Y <: A = C) b[Y] \leftrightarrow \text{wrap}(Y <: A' = C) b'[Y] : A}$$

(Eq Use) (where $A \equiv \zeta(X)B\{X\}$)

$$E \vdash c \leftrightarrow c' : A \quad E \vdash D \quad E, Y <: A, y:B\{Y\} \vdash d \leftrightarrow d' : D$$

$$\frac{}{E \vdash \text{use } c \text{ as } Y <: A, y:B\{Y\} \text{ in } d:D \leftrightarrow \text{use } c' \text{ as } Y <: A, y:B\{Y\} \text{ in } d':D : D}$$

(Eval Select) (where $A \equiv [l_i:B_i]_{i \in 1..n}$, $a \equiv [l_i=\zeta(x_i:A')b_i]_{i \in 1..n+m}$)

$$E \vdash a : A \quad j \in 1..n$$

$$E \vdash a.l_j \leftrightarrow b_j\{a\} : B_j$$

(Eval Update) (where $A \equiv [l_i:B_i]_{i \in 1..n}$, $a \equiv [l_i=\zeta(x_i:A')b_i]_{i \in 1..n+m}$)

$$E \vdash a : A \quad E, x:A \vdash b : B_j \quad j \in 1..n$$

$$E \vdash a.l_j=\zeta(x:A)b \leftrightarrow [l_j=\zeta(x:A')b, l_i=\zeta(x_i:A')b_i]_{i \in (1..n+m)-\{j\}} : A$$

(Eval Unwrap) (where $A \equiv \zeta(X)B\{X\}$, $c \equiv \text{wrap}(Z <: A = C)b\{Z\}$)

$$E \vdash c : A \quad E \vdash D \quad E, Y <: A, y:B\{Y\} \vdash d\{Y,y\} : D$$

$$E \vdash \text{use } c \text{ as } Y <: A, y:B\{Y\} \text{ in } d\{Y,y\}:D \leftrightarrow d\{C,b\{C\}\} : D$$

(Eval Rewrap) (where $A \equiv \zeta(X)B\{X\}$)

$$E \vdash b : A \quad E, y:A \vdash d\{y\} : D$$

$$E \vdash \text{use } b \text{ as } Y <: A, y:B\{Y\} \text{ in } d\{\text{wrap}(Y <: A = Y)y\}:D \leftrightarrow d\{b\} : D$$

C PROOFS

C.1 Proof of the Variance Lemma from Section 13.3

Our goal in this appendix is to prove technical lemmas that guarantee that positive occurrence is a sufficient condition for covariance. As a first step, we state two lemmas.

Lemma C.1-1 (Bound weakening)

If $E, X<:A, E' \vdash \mathfrak{S}$ and $E \vdash A' <: A$, then $E, X<:A', E' \vdash \mathfrak{S}$.

□

Lemma C.1-2 (Substitution)

If $E, X<:A, E'\{X\} \vdash \mathfrak{S}\{X\}$ and $E \vdash A' <: A$, then $E, E'\{A'\} \vdash \mathfrak{S}\{A'\}$.

□

The following table defines the sets of free variables of parts of environments:

Scoping for (incomplete) environments

$FV(\emptyset)$	$\triangleq \{\}$
$FV(E, x:A)$	$\triangleq FV(E) \cup (FV(A) - \text{dom}(E))$
$FV(E, X<:A)$	$\triangleq FV(E) \cup (FV(A) - \text{dom}(E))$

In order to manipulate positive and negative occurrences of a variable, we prove a general lemma about the separation of a set of occurrences into two sets:

Lemma C.1-3 (Variable separation)

- (1) Let Y and Z be different variables that may occur free in an environment $E'\{Y,Z\}$. Assume $X \notin FV(E')$ and $Y, Z \notin \text{dom}(E, E')$. If $E, X<:C, E'\{X,X\} \vdash \diamond$, then $E, Y<:C, Z<:C, E'\{Y,Z\} \vdash \diamond$.
- (2) Let Y and Z be different variables that may occur free in an environment $E'\{Y,Z\}$ and in a type $D\{Y,Z\}$. Assume $X \notin FV(E') \cup FV(D)$, and $Y, Z \notin \text{dom}(E, E')$. If $E, X<:C, E'\{X,X\} \vdash D\{X,X\}$, then $E, Y<:C, Z<:C, E'\{Y,Z\} \vdash D\{Y,Z\}$.

Proof

The proof is by simultaneous induction on the derivations:

(1)

Case (Env \emptyset)

Not applicable.

Case (Env x)

In this case $E\{X,X\} \equiv (E_1\{X,X\}, z:D\{X,X\})$ and we have $E, X<:C, E_1\{X,X\} \vdash D\{X,X\}$. By induction hypothesis (2), $E, Y<:C, Z<:C, E_1\{Y,Z\} \vdash D\{Y,Z\}$. Hence by (Env x), $E, Y<:C, Z<:C, E_1\{Y,Z\}, z:D\{Y,Z\} \vdash \diamond$.

Case (Env $X<:$)

If $E' \equiv \emptyset$, then we have $E \vdash \diamond$ and $E \vdash C$. By (Env $X<:$), $E, Y<:C \vdash \diamond$. By weakening, $E, Y<:C \vdash C$. By (Env $X<:$) again, $E, Y<:C, Z<:C \vdash \diamond$.

If $E'\{X,X\} \equiv (E_1\{X,X\}, W<:D\{X,X\})$, then we have $E, X<:C, E_1\{X,X\} \vdash D\{X,X\}$. By induction hypothesis (2), $E, Y<:C, Z<:C, E_1\{Y,Z\} \vdash D\{Y,Z\}$. By (Env $X<:$) $E, Y<:C, Z<:C, E_1\{Y,Z\}, W<:D\{Y,Z\} \vdash \diamond$.

(2)

Case (Type $X<:$)

If $D\{X,X\} \equiv X$, then either $D\{Y,Z\} \equiv Y$ or $D\{Y,Z\} \equiv Z$ (since $X \notin FV(D)$). We have $E, X<:C, E'\{X,X\} \vdash \diamond$. By induction hypothesis (1), $E, Y<:C, Z<:C, E'\{Y,Z\} \vdash \diamond$, and by (Type $X<:$) both $E, Y<:C, Z<:C, E'\{Y,Z\} \vdash Y$ and $E, Y<:C, Z<:C, E'\{Y,Z\} \vdash Z$.

If $D\{X,X\} \equiv W \not\equiv X$, then $W \in dom(E, E')$ and $W \not\equiv Y, W \not\equiv Z$, so $D\{Y,Z\} \equiv W$. By induction hypothesis (1), $E, Y<:C, Z<:C, E'\{Y,Z\} \vdash \diamond$, and by (Type $X<:$) $E, Y<:C, Z<:C, E'\{Y,Z\} \vdash W$.

Case (Type Top)

In this case, $D\{X,X\} \equiv Top \equiv D\{Y,Z\}$. We have $E, X<:C, E'\{X,X\} \vdash \diamond$. By induction hypothesis (1), $E, Y<:C, Z<:C, E'\{Y,Z\} \vdash \diamond$, and by (Type Top) $E, Y<:C, Z<:C, E'\{Y,Z\} \vdash Top$.

Case (Type Object)

In this case, $D\{X,X\} \equiv [l_i:D_i]_{i \in 1..n}$. Hence $D\{Y,Z\} \equiv [l_i:D_i\{Y,Z\}]_{i \in 1..n}$ for some D_i such that $D'_i \equiv D_i\{X,X\}$, for $i \in 1..n$. We have $E, X<:C, E'\{X,X\} \vdash D_1\{X,X\}$. By induction hypothesis (2), $E, Y<:C, Z<:C, E'\{Y,Z\} \vdash D_1\{Y,Z\}$. By (Type Object), $E, Y<:C, Z<:C, E'\{Y,Z\} \vdash [l_i:D_i\{Y,Z\}]_{i \in 1..n}$.

Case (Type Arrow)

In this case, $D\{X,X\} \equiv D' \rightarrow D''$. Hence $D\{Y,Z\} \equiv D_1\{Y,Z\} \rightarrow D_2\{Y,Z\}$ for some D_1 and D_2 such that $D' \equiv D_1\{X,X\}$ and $D'' \equiv D_2\{X,X\}$. We have $E, X<:C, E'\{X,X\} \vdash D_1\{X,X\}$ and $E, X<:C, E'\{X,X\} \vdash D_2\{X,X\}$. By induction hypothesis (2), $E, Y<:C, Z<:C, E'\{Y,Z\} \vdash D_1\{Y,Z\}$ and $E, Y<:C, Z<:C, E'\{Y,Z\} \vdash D_2\{Y,Z\}$. By (Type Arrow), $E, Y<:C, Z<:C, E'\{Y,Z\} \vdash D_1\{Y,Z\} \rightarrow D_2\{Y,Z\}$.

Case (Type Rec $<:$)

In this case, $D\{X,X\} \equiv \mu(W)D'$ with $W \notin dom(E) \cup \{X,Y,Z\}$. Hence $D\{Y,Z\} \equiv \mu(W)(D_1\{Y,Z\})$ for some D_1 such that $D' = D_1\{X,X\}$. We have $E, X<:C, E'\{X,X\}$,

$W <: \text{Top} \vdash D_1\{X, X\}$. By induction hypothesis (2), $E, Y <: C, Z <: C, E'\{Y, Z\}, W <: \text{Top} \vdash D_1\{Y, Z\}$. By (Type Rec $<:$), $E, Y <: C, Z <: C, E'\{Y, Z\} \vdash \mu(W)(D_1\{Y, Z\})$.

Case (Type All $<:$)

In this case, $D\{X, X\} \equiv \forall(W <: D')D''$ with $W \notin \text{dom}(E) \cup \{X, Y, Z\}$. Hence $D\{Y, Z\} \equiv \forall(W <: D_1\{Y, Z\})D_2\{Y, Z\}$ for some D_1 and D_2 such that $D' \equiv D_1\{X, X\}$ and $D'' \equiv D_2\{X, X\}$. We have $E, X <: C, E'\{X, X\} \vdash D_1\{X, X\}$ and $E, X <: C, E'\{X, X\}, W <: D_1\{X, X\} \vdash D_2\{X, X\}$. By induction hypothesis (2), $E, Y <: C, Z <: C, E'\{Y, Z\} \vdash D_1\{Y, Z\}$ and $E, Y <: C, Z <: C, E'\{Y, Z\}, W <: D_1\{Y, Z\} \vdash D_2\{Y, Z\}$. By (Type All $<:$), $E, Y <: C, Z <: C, E'\{Y, Z\} \vdash \forall(W <: D_1\{Y, Z\})D_2\{Y, Z\}$.

Case (Type Exists $<:$)

Similar to case (Type All $<:$).

□

Corollary C.1-4

Under the same assumptions, we can also conclude:

- (1) $E, Y <: C, Z <: Y, E'\{Y, Z\} \vdash \diamond$
- (2) $E, Y <: C, Z <: Y, E'\{Y, Z\} \vdash D\{Y, Z\}$

Proof

By Lemma C.1-1, since $E, Y <: C \vdash Y <: C$.

□

The following general Variance Lemma establishes that positive occurrence is a sufficient condition for covariance, and negative occurrence for contravariance.

Lemma C.1-5 (Variance)

Assume $E'\{V^+, U^-\}$ and $A\{V^+, U^-\}$ with $Y \notin \text{dom}(E, X <: C, E')$ and $X \notin FV(E') \cup FV(A)$. If $E, X <: C, E'\{X, X\} \vdash A\{X, X\}$, then $E, X <: C, Y <: X, E'\{Y, X\} \vdash A\{Y, X\} <: A\{X, Y\}$.

Proof

The proof is by induction on the size of $A\{V^+, U^-\}$. Without loss of generality, we assume that U and V are different from all the variables bound in A .

Case A $\{V^+, U^-\} \equiv W$

If $W \equiv V$, then $A\{Y, X\} \equiv Y$ and $A\{X, Y\} \equiv X$. We have $E, X <: C, E'\{X, X\} \vdash \diamond$, so $E, U <: C, V <: U, E'\{V^+, U^-\} \vdash \diamond$ by Corollary C.1-4(1), and $E, X <: C, Y <: X, E'\{Y, X\} \vdash \diamond$ by renaming. Then $E, X <: C, Y <: X, E'\{Y, X\} \vdash Y <: X$ by (Sub X).

If $W \not\equiv V$, then $A\{Y, X\} \equiv W \equiv A\{X, Y\}$ and $W \in \text{dom}(E, E')$. We have $E, X <: C, E'\{X, X\} \vdash \diamond$, so $E, U <: C, V <: U, E'\{V^+, U^-\} \vdash \diamond$ by Corollary C.1-4(1), and $E, X <: C, Y <: X, E'\{Y, X\} \vdash \diamond$ by renaming. Then $E, X <: C, Y <: X, E'\{Y, X\} \vdash W$ by (Type X $<:$), and $E, X <: C, Y <: X, E'\{Y, X\} \vdash W <: W$ by (Sub Refl).

Case A $\{V^+, U^-\} \equiv Top$

Then $A\{Y, X\} \equiv Top \equiv A\{X, Y\}$. We have $E, X <: C, E'\{X, X\} \vdash \diamond$, so $E, U <: C, V <: U, E'\{V^+, U^-\} \vdash \diamond$ by Corollary C.1-4(1), and $E, X <: C, Y <: X, E'\{Y, X\} \vdash \diamond$ by renaming. Then $E, X <: C, Y <: X, E'\{Y, X\} \vdash Top$ by (Type Top), and $E, X <: C, Y <: X, E'\{Y, X\} \vdash Top <: Top$ by (Sub Refl).

Case A $\{V^+, U^-\} \equiv [l_i : A_i]_{i \in 1..n}$

Then U and V cannot occur in any of the A_i . We have $E, X <: C, E'\{X, X\} \vdash A_i$. Hence $E, U <: C, V <: U, E'\{V^+, U^-\} \vdash A_i$ by Corollary C.1-4(2), and $E, X <: C, Y <: X, E'\{Y, X\} \vdash A_i$ by renaming. Then $E, X <: C, Y <: X, E'\{Y, X\} \vdash A <: A$ by (Sub Object).

Case A $\{V^+, U^-\} \equiv A_1\{U^+, V^-\} \rightarrow A_2\{V^+, U^-\}$

We have $E, X <: C, E'\{X, X\} \vdash A_1\{X, X\}$ and $E, X <: C, E'\{X, X\} \vdash A_2\{X, X\}$. To apply the induction hypothesis we need to consider $A'_1 \triangleq A_1\{V, U\}$, so $A'_1\{V^+, U^-\} \vdash A'_1\{X, X\}$ and $A'_1\{D', D''\} \equiv A_1\{D', D''\}$ for any D' and D'' . We have $E, X <: C, E'\{X, X\} \vdash A'_1\{X, X\}$. By induction hypothesis, $E, X <: C, Y <: X, E'\{Y, X\} \vdash A'_1\{Y, X\} <: A'_1\{X, Y\}$; that is, $E, X <: C, Y <: X, E'\{Y, X\} \vdash A_1\{Y, X\} <: A_1\{X, Y\}$. Also by induction hypothesis, $E, X <: C, Y <: X, E'\{Y, X\} \vdash A_2\{Y, X\} <: A_2\{X, Y\}$. By (Sub Arrow), $E, X <: C, Y <: X, E'\{Y, X\} \vdash A_1\{X, Y\} \rightarrow A_2\{Y, X\} <: A_1\{Y, X\} \rightarrow A_2\{X, Y\}$. That is, $E, X <: C, Y <: X, E'\{Y, X\} \vdash (A_1 \rightarrow A_2)\{Y, X\} <: (A_1 \rightarrow A_2)\{X, Y\}$.

Case A $\{V^+, U^-\} \equiv (\mu(Z)A_1\{Z\})\{V^+, U^-\}$

Then either V and U do not occur free in $A_1\{Z\}$, or $A_1\{Z^+\}\{V^+, U^-\}$, by definition of variant occurrence.

In the former case, we have $E, X <: C, E'\{X, X\}, Z <: Top \vdash A_1\{Z\}$. By Corollary C.1-4(2) $E, X <: C, Y <: X, E'\{Y, X\}, Z <: Top \vdash A_1\{Z\}$, so by (Type Rec) $E, X <: C, Y <: X, E'\{Y, X\} \vdash \mu(Z)(A_1\{Z\})$, and by (Sub Refl) $E, X <: C, Y <: X, E'\{Y, X\} \vdash \mu(Z)(A_1\{Z\}) <: \mu(Z)(A_1\{Z\})$.

In the latter case, we have $E, X <: C, E'\{X, X\}, Z <: Top \vdash A_1\{Z^+\}\{X, X\}$, so by induction hypothesis $E, X <: C, Y <: X, E'\{Y, X\}, Z <: Top \vdash A_1\{Z^+\}\{Y, X\} <: A_1\{Z^+\}\{X, Y\}$. By Corollary C.1-4(2) $E, X <: C, Y <: X, E'\{Y, X\}, Z <: Top \vdash A_1\{Z^+\}\{Y, X\}$. Since $A_1\{Z^+\}\{Y, X\}$ is smaller than A , we use the induction hypothesis again with $W \notin dom(E, X <: C, Y <: X, E'\{Y, X\}, Z <: Top)$, obtaining $E, X <: C, Y <: X, E'\{Y, X\}, Z <: Top, W <: Z \vdash A_1\{W\}\{Y, X\} <: A_1\{Z\}\{Y, X\}$. Therefore by weakening and (Sub Trans), $E, X <: C, Y <: X, E'\{Y, X\}, Z <: Top, W <: Z \vdash A_1\{W\}\{Y, X\} <: A_1\{Z\}\{X, Y\}$, and by (Sub Rec) $E, X <: C, Y <: X, E'\{Y, X\} \vdash \mu(W)(A_1\{W\}\{Y, X\}) <: \mu(Z)(A_1\{Z\}\{X, Y\})$. That is, $E, X <: C, Y <: X, E'\{Y, X\} \vdash (\mu(Z)A_1\{Z\})\{Y, X\} <: (\mu(Z)A_1\{Z\})\{X, Y\}$.

Case A $\{V^+, U^-\} \equiv \forall(Z <: A_1\{U^+, V^-\})A_2\{V^+, U^-\}$

We have $E, X <: C, E'\{X, X\} \vdash A_1\{X, X\}$ and $E, X <: C, E'\{X, X\}, Z <: A_1\{X, X\} \vdash A_2\{X, X\}$.

To apply the induction hypothesis we need to consider $A'_1 \triangleq A_1\{V, U\}$, so $A'_1\{V^+, U^-\}$ and $A'_1\{D', D''\} \equiv A_1\{D', D''\}$ for any D' and D'' . We have $E, X <: C, E'\{X, X\} \vdash A'_1\{X, X\}$ and $E, X <: C, E'\{X, X\}, Z <: A'_1\{X, X\} \vdash A_2\{X, X\}$. By induction

hypothesis, $E, X <: C, Y <: X, E'\{Y, X\} \vdash A'_1\{Y, X\} <: A'_1\{X, Y\}$; that is, $E, X <: C, Y <: X, E'\{Y, X\} \vdash A_1\{Y, X\} <: A_1\{X, Y\}$. Also by induction hypothesis, $E, X <: C, Y <: X, E'\{Y, X\}, Z <: A'_1\{Y, X\} \vdash A_2\{Y, X\} <: A_2\{X, Y\}$, that is $E, X <: C, Y <: X, E'\{Y, X\}, Z <: A_1\{Y, X\} \vdash A_2\{Y, X\} <: A_2\{X, Y\}$. By (Sub All $<$), $E, X <: C, Y <: X, E'\{Y, X\} \vdash \forall(Z <: A_1\{X, Y\})A_2\{Y, X\} <: \forall(Z <: A_1\{X, Y\})A_2\{X, Y\}$. That is, $E, X <: C, Y <: X, E'\{Y, X\} \vdash (\forall(Z <: A_1)A_2)\{Y, X\} <: (\forall(Z <: A_1)A_2)\{X, Y\}$.

Case A $\{V^+, U^-\} \equiv \exists(Z <: A_1\{V^+, U^-\})A_2\{V^+, U^-\}$

We have $E, X <: C, E'\{X, X\} \vdash A_1\{X, X\}$ and $E, X <: C, E'\{X, X\}, Z <: A_1\{X, X\} \vdash A_2\{X, X\}$. By induction hypothesis, $E, X <: C, Y <: X, E'\{Y, X\} \vdash A_1\{Y, X\} <: A_1\{X, Y\}$ and $E, X <: C, Y <: X, E'\{Y, X\}, Z <: A_1\{Y, X\} \vdash A_2\{Y, X\} <: A_2\{X, Y\}$. By (Sub Exists $<$), $E, X <: C, Y <: X, E'\{Y, X\} \vdash \exists(Z <: A_1\{Y, X\})A_2\{Y, X\} <: \exists(Z <: A_1\{X, Y\})A_2\{X, Y\}$. That is, $E, X <: C, Y <: X, E'\{Y, X\} \vdash (\exists(Z <: A_1)A_2)\{Y, X\} <: (\exists(Z <: A_1)A_2)\{X, Y\}$.

□

We are now ready to prove the two lemmas stated in Section 13.3.

Lemma C.1-6 (Lemma 13.3-1)

Assume $E'\{Y^+, X^-\}$ and $B\{X^+\}$.

If $E, X <: A, E'\{X, X\} \vdash B\{X\}$, then $E, X <: A, E'\{X, A\} \vdash B\{X\} <: B\{A\}$.

Proof

The assumption $E, X <: A, E'\{X, X\} \vdash B\{X\}$ gives $E, X <: A, Y <: X, E'\{Y, X\} \vdash B\{Y\} <: B\{X\}$ (for a suitable Y) by Lemma C.1-5, and then $E, Y <: A, E'\{Y, A\} \vdash B\{Y\} <: B\{A\}$ by Lemma C.1-2. We conclude by renaming.

□

Lemma C.1-7 (Lemma 13.3-2)

Assume $B\{Y^+, X^-\}$.

If $E, X <: A \vdash B\{X, X\}$ and $E \vdash C_1 <: C_2$ and $E \vdash C_2 <: A$, then $E \vdash B\{C_1, C_2\} <: B\{C_2, C_1\}$.

Proof

The assumption $E, X <: A \vdash B\{X, X\}$ gives $E, X <: A, Y <: X \vdash B\{Y, X\} <: B\{X, Y\}$ (for a suitable Y) by Lemma C.1-5, then $E, Y <: C_2 \vdash B\{Y, C_2\} <: B\{C_2, Y\}$ by Lemma C.1-2, and $E \vdash B\{C_1, C_2\} <: B\{C_2, C_1\}$ by Lemma C.1-2 again.

□

C.2 Proof of the Variance Lemma from Section 16.4

We extend the results of Appendix C.1 to the object types with Self of Chapter 16.

Lemma C.2-1 (Variance)

Assume $E'\{V^+, U^-\}$ and $A\{V^+, U^-\}$ with $Y \notin \text{dom}(E, X <: C, E')$ and $X \notin \text{FV}(E') \cup \text{FV}(A)$.

If $E, X <: C, E'\{X, X\} \vdash A\{X, X\}$, then $E, X <: C, Y <: X, E'\{Y, X\} \vdash A\{Y, X\} <: A\{X, Y\}$.

Proof

The proof is by induction on the size of $A\{V^+, U^-\}$, and is similar to the proof of Lemma C.1-5. We detail only the case for objects.

Case $A\{V^+, U^-\} \equiv Obj(Z)[l; v_i; B_i^{i \in 1..n}]$

Without loss of generality, we assume that U and V are different from Z .

If $v_i \equiv ^0$, then $U, V \notin FV(B_i)$. We have $E, X <: C, E'\{X, X\}, Z <: Top \vdash B_i\{Z^+\}$, and $E, X <: C, E'\{X, X\}, Z <: A\{X, X\} \vdash B_i\{Z^+\}$ by Lemma 16.4-1. By induction hypothesis $E, X <: C, Y <: X, E'\{Y, X\}, Z <: A\{Y, X\} \vdash B_i\{Z\} <: B_i\{Z\}$. Therefore $E, X <: C, Y <: X, E'\{Y, X\}, Z <: A\{Y, X\} \vdash {}^0 B_i\{Z\} <: {}^0 B_i\{Z\}$.

If $v_i \equiv ^+$, then $B_i\{Z^+\}\{V^+, U^-\}$. We have $E, X <: C, E'\{X, X\}, Z <: Top \vdash B_i\{Z\}\{X, X\}$, and $E, X <: C, E'\{X, X\}, Z <: A\{X, X\} \vdash B_i\{Z\}\{X, X\}$ by Lemma 16.4-1. By induction hypothesis $E, X <: C, Y <: X, E'\{Y, X\}, Z <: A\{Y, X\} \vdash B_i\{Z\}\{Y, X\} <: B_i\{Z\}\{X, Y\}$. Therefore $E, X <: C, Y <: X, E'\{Y, X\}, Z <: A\{Y, X\} \vdash {}^+ B_i\{Z\}\{Y, X\} <: {}^+ B_i\{Z\}\{X, Y\}$.

If $v_i \equiv ^-$, then $B_i\{Z^+\}\{U^-, V^-\}$. We have $E, X <: C, E'\{X, X\}, Z <: Top \vdash B_i\{Z\}\{X, X\}$, and $E, X <: C, E'\{X, X\}, Z <: A\{X, X\} \vdash B_i\{Z\}\{X, X\}$ by Lemma 16.4-1. To apply the induction hypothesis we need to consider $B'_i \triangleq B_i\{Z\}\{V, U\}$, so $B'_i\{Z^+\}\{V^+, U^-\}$ and $B_i\{Z\}\{D', D''\} \equiv B'_i\{Z\}\{D', D''\}$ for any D' and D'' . We have $E, X <: C, E'\{X, X\}, Z <: A\{X, X\} \vdash B'_i\{Z\}\{X, X\}$. By induction hypothesis, $E, X <: C, Y <: X, E'\{Y, X\}, Z <: A\{Y, X\} \vdash B'_i\{Z\}\{Y, X\} <: B'_i\{Z\}\{X, Y\}$; that is, $E, X <: C, Y <: X, E'\{Y, X\}, Z <: A\{Y, X\} \vdash B_i\{Z\}\{Y, X\} <: B_i\{Z\}\{X, Y\}$. Therefore $E, X <: C, Y <: X, E'\{Y, X\}, Z <: A\{Y, X\} \vdash {}^- B_i\{Z\}\{X, Y\} <: {}^- B_i\{Z\}\{Y, X\}$.

Then $E, X <: C, Y <: X, E'\{Y, X\} \vdash A\{Y, X\} <: A\{X, Y\}$ follows by (Sub Object).

□

Lemma C.2-2 (Lemma 16.4-6)

Assume $B\{Y^+, X^-\}$.

If $E, X <: A \vdash B\{X, X\}$ and $E \vdash C_1 <: C_2$ and $E \vdash C_2 <: A$, then $E \vdash B\{C_1, C_2\} <: B\{C_2, C_1\}$.

Proof

As for Lemma C.1-7.

□

C.3 Deriving the Rules for ζ -Objects from Section 15.1.2

We verify some of the derived rules for ζ -objects in detail. We omit the easier treatment of other rules; in particular (Sub ζ Object) follows from (Sub Self).

(Val ζ Select) (where $A \equiv \zeta(X)[l; B_i\{X^+\}^{i \in 1..n}]$)

The derivation relies on the covariance of B_i in X , and subsumption.

$E, Y <: A \vdash B_j\{Y\}$ from the assumption of (Val ζ Select)

$E, Y <: A \vdash B_j\{Y\} <: B_j\{A\}$ by Lemma 13.3-1
 $E, Y <: A, y:A(Y) \vdash B_j\{Y\} <: B_j\{A\}$ by weakening
 $E, Y <: A, y:A(Y) \vdash y.l_j : B_j\{Y\}$ by (Val Select)
 $E, Y <: A, y:A(Y) \vdash y.l_j : B_j\{A\}$ by (Val Subsumption)
 $E \vdash (\text{use } a \text{ as } Y <: A, y:A(Y) \text{ in } y.l_j : B_j\{A\}) : B_j\{A\}$ by (Val Use) since $E \vdash a : A$
 $E \vdash a_{A,l_j} : B_j\{A\}$ by definition

(**Val ζ Update**) (where $A \equiv \zeta(X)[l_i; B_i\{X^+\}_{i \in 1..n}]$)

The derivation relies on the full power of (Val Wrap). Note that $y : A(Y)$ does not imply $y : A(A)$, even though $Y <: A$. Otherwise b would only need to have type $B_i\{A\}$, and this would permit unsound updates.

$E, Y <: A, y:A(Y), x:A(Y) \vdash b\{Y,y,x\} : B_j\{Y\}$ by assumption
 $E, Y <: A, y:A(Y) \vdash y.l_j \Leftarrow \zeta(x:A(Y))b\{Y,y,x\} : A(Y)$ by (Val Update)
 $E, Y <: A, y:A(Y) \vdash \text{wrap}(Z <: A = Y) y.l_j \Leftarrow \zeta(x:A(Z))b\{Z,y,x\} : A$ by (Val Wrap)
 $E \vdash (\text{use } a \text{ as } Y <: A, y:A(Y) \text{ in } \text{wrap}(Z <: A = Y) y.l_j \Leftarrow \zeta(x:A(Z))b\{Z,y,x\} : A) : A$
 by (Val Use) since $E \vdash a : A$
 $E \vdash a_{A,l_j} : (Y <: A, y:A(Y))\zeta(x:A(Y))b : A$ by definition

(**Eq ζ Object**) (where $A \equiv \zeta(X)[l_i; B_i\{X^+\}_{i \in 1..n}]$, $A' \equiv \zeta(X)[l_i; B_i\{X^+\}_{i \in 1..n+m}]$)

$E \vdash A' <: A$ by (Sub Self) and (Sub Object)
 $E \vdash b\{C\} \leftrightarrow b'\{C\} : A'(C)$ by assumption
 $E \vdash \text{pack } Y <: A = C \text{ with } b\{Y\}:A(Y)$
 $\leftrightarrow \text{pack } Y <: A' = C \text{ with } b'\{Y\}:A'(Y) : \exists(X <: A)A(X)$
 by (Eq Pack<:) since $E \vdash C <: A'$
 $E \vdash \text{fold}(A, \text{pack } Y <: A = C \text{ with } b\{Y\}:A(Y))$
 $\leftrightarrow \text{fold}(A, \text{pack } Y <: A' = C \text{ with } b'\{Y\}:A'(Y)) : A$ by (Eq Fold)

 $E, Y, X <: Y \vdash \exists(Z <: X)A'(Z) <: \exists(Z <: Y)A(Z)$ as for (Sub Self)
 $E \vdash b'\{C\} \leftrightarrow b'\{C\} : A'(C)$ by (Eq Symm) and (Eq Trans)
 $E \vdash \text{pack } Y <: A' = C \text{ with } b'\{Y\}:A'(Y)$
 $\leftrightarrow \text{pack } Y <: A' = C \text{ with } b'\{Y\}:A'(Y) : \exists(X <: A')A'(X)$
 by (Eq Pack<:) since $E \vdash C <: A'$
 $E \vdash \text{fold}(A, \text{pack } Y <: A' = C \text{ with } b'\{Y\}:A'(Y))$
 $\leftrightarrow \text{fold}(A', \text{pack } Y <: A' = C \text{ with } b'\{Y\}:A'(Y)) : A$ by (Eq Fold<: Lemma1)

 $E \vdash \text{fold}(A, \text{pack } Y <: A = C \text{ with } b\{Y\}:A(Y))$
 $\leftrightarrow \text{fold}(A', \text{pack } Y <: A' = C \text{ with } b'\{Y\}:A'(Y)) : A$ by (Eq Trans)
 $E \vdash \text{wrap}(Y <: A = C)b\{Y\} \leftrightarrow \text{wrap}(Y <: A' = C)b'\{Y\} : A$ by definition

(**Eq ζ Sub ζ Object**) (where $A \equiv \zeta(X)[l_i; B_i\{X^+\}_{i \in 1..n}]$, $A' \equiv \zeta(X)[l_i; B_i\{X^+\}_{i \in 1..n+m}]$)

$E \vdash [l_i \Leftarrow \zeta(x_i : A(C))b_i\{C\}_{i \in 1..n}] \leftrightarrow [l_i \Leftarrow \zeta(x_i : A'(C))b_i\{C\}_{i \in 1..n+m}] : A(C)$
 by (Eq Sub Object)
 $E, x_i : A'(C) \vdash b_i\{C\} : B_i\{C\}_{i \in 1..n+m}$

by a bound weakening lemma, using $A'(C) \triangleleft A(C)$

$E \vdash [l_i = \zeta(x_i; A'(C)) b_i \{ C \}^{i \in 1..n+m}] : A'(C)$ by (Val Object)

$E \vdash \text{wrap}(Y \triangleleft A = C)[l_i = \zeta(x_i; A(Y)) b_i \{ Y \}^{i \in 1..n}]$

$\leftrightarrow \text{wrap}(Y \triangleleft A' = C)[l_i = \zeta(x_i; A'(Y)) b_i \{ Y \}^{i \in 1..n+m}] : A$ by (Eq Wrap Lemma)

(Eval ζ Select) (where $A \equiv \zeta(X)[l_i; B_i \{ X \}^{i \in 1..n}]$, $c \equiv \text{wrap}(Z \triangleleft A = C) a \{ Z \}$)

$E, Y \triangleleft A, y: A(Y) \vdash y.l_j : B_j \{ A \}$ as for (Val ζ Select)

$E \vdash (\text{use } c \text{ as } Y \triangleleft A, y: A(Y) \text{ in } y.l_j; B_j \{ A \}) \leftrightarrow a \{ C \}.l_j : B_j \{ A \}$ by (Eval Unwrap)

$E \vdash c.a.l_j \leftrightarrow a \{ C \}.l_j : B_j \{ A \}$ by definition

(Eval ζ Update) (where $A \equiv \zeta(X)[l_i; B_i \{ X \}^{i \in 1..n}]$, $c \equiv \text{wrap}(Z \triangleleft A = C) a \{ Z \}$)

$E, Y \triangleleft A, y: A(Y) \vdash \text{wrap}(Z \triangleleft A = Y) y.l_j = \zeta(x: A(Z)) b \{ Z, y, x \} : A$ as for (Val ζ Update)

$E \vdash (\text{use } c \text{ as } Y \triangleleft A, y: A(Y) \text{ in } \text{wrap}(Z \triangleleft A = Y) y.l_j = \zeta(x: A(Z)) b \{ Z, y, x \}; A)$

$\leftrightarrow \text{wrap}(Z \triangleleft A = C) a \{ C \}.l_j = \zeta(x: A(Z)) b \{ Z, a \{ C \}, x \}$ by (Eval Unwrap)

$\leftrightarrow \text{wrap}(Z \triangleleft A = C) a \{ Z \}.l_j = \zeta(x: A(Z)) b \{ Z, a \{ Z \}, x \} : A$ by (Eq Wrap)

$E \vdash c.l_j = (Y \triangleleft A, y: A(Y)) \zeta(x: A(Y)) b \{ Y, y, x \}$

$\leftrightarrow \text{wrap}(Z \triangleleft A = C) a \{ Z \}.l_j = \zeta(x: A(Z)) b \{ Z, a \{ Z \}, x \} : A$ by definition

C.4 Denotational Soundness of Equational Rules

In this appendix we prove the soundness of the equational rules of $\mathbf{FOb}_{\triangleleft_{\mu}}$ in the denotational model of Chapter 14.

First we state some basic properties of records:

Proposition C.4-1

- (1) Let $I \cap K = \emptyset$. If $(x_i, x'_i) \in T_i$ for all $i \in I$, then $(\langle\langle m_i = x_i \rangle\rangle^{i \in I}, \langle\langle m_i = x'_i \rangle\rangle^{i \in I}, \langle\langle m_k = y_k \rangle\rangle^{k \in K}) \in \langle\langle m_i; T_i \rangle\rangle^{i \in I}$.
- (2) If $(x, x') \in \langle\langle m_i; T_i \rangle\rangle^{i \in I}$ and $(y, y') \in T_k$ for some $k \in I$, then $(x(m_k \leftarrow y), x'(m_k \leftarrow y')) \in \langle\langle m_i; T_i \rangle\rangle^{i \in I}$.

□

We have analogous properties for objects. The next group of results concerns the equality of explicitly given objects and the semantic properties of method invocation and update.

Proposition C.4-2

Let $I \cap K = \emptyset$. If $(x_j, x'_j) \in \langle\langle m_j; T_j \rangle\rangle^{j \in I} \rightarrow T_j$ for all $j \in I$, then $(\langle\langle m_i = x_i \rangle\rangle^{i \in I}, \langle\langle m_i = x'_i \rangle\rangle^{i \in I}, \langle\langle m_k = y_k \rangle\rangle^{k \in K}) \in \mu(S)(\langle\langle m_i; S \rightarrow T_i \rangle\rangle^{i \in I})$, and hence $(\langle\langle m_i = x_i \rangle\rangle^{i \in I}, \langle\langle m_i = x'_i \rangle\rangle^{i \in I}, \langle\langle m_k = y_k \rangle\rangle^{k \in K}) \in \langle\langle m_i; T_i \rangle\rangle^{i \in I}$.

Proof

Let $T = \langle\langle m_i; T_i \rangle\rangle^{i \in I}$ and let $R = \mu(S)(\langle\langle m_i; S \rightarrow T_i \rangle\rangle^{i \in I})$. Note that $R \subseteq T$. The hypothesis and Proposition C.4-1 yield that $(\langle\langle m_i = x_i \rangle\rangle^{i \in I}, \langle\langle m_i = x'_i \rangle\rangle^{i \in I}, \langle\langle m_k = y_k \rangle\rangle^{k \in K}) \in \langle\langle m_i; T \rightarrow T_i \rangle\rangle^{i \in I}$. Since $R \subseteq T$, \rightarrow is antimonotonic, and $\langle\langle \dots \rangle\rangle$ is monotonic, it follows that $(\langle\langle m_i = x_i \rangle\rangle^{i \in I})$,

$\langle\langle m_i=x_i^{i \in I}, m_k=y_k^{k \in K} \rangle\rangle \in \langle\langle m_i:R \rightarrow T_i^{i \in I} \rangle\rangle$. By folding, it follows that $\langle\langle m_i=x_i^{i \in I} \rangle\rangle, \langle\langle m_i=x_i^{i \in I}, m_k=y_k^{k \in K} \rangle\rangle \in R$, and hence $\langle\langle m_i=x_i^{i \in I} \rangle\rangle, \langle\langle m_i=x_i^{i \in I}, m_k=y_k^{k \in K} \rangle\rangle \in T$.

□

Proposition C.4-3

If $(x, x') \in [[m_i; T_i^{i \in I}]]$, then either $x = x' = \perp$ or $x, x' \in (L \rightarrow D)$.

Proof

Since $[[m_i; T_i^{i \in I}]] \triangleq \sqcup \{\mu(S)F(S) \mid F \in \mathbf{Gen}, F \leq \lambda(S)\langle\langle m_i: S \rightarrow T_i^{i \in I} \rangle\rangle\}$, we prove that if $(x, x') \in \mu(S)F(S)$ and $F \leq \lambda(S)\langle\langle m_i: S \rightarrow T_i^{i \in I} \rangle\rangle$ then $(x, x') \in \{(\perp, \perp)\} \cup (L \rightarrow D) \times (L \rightarrow D)$. From this it follows that $(\lambda(x)x, \lambda(x)x) \in \mu(S)F(S) \rightarrow (\{(\perp, \perp)\} \cup (L \rightarrow D) \times (L \rightarrow D))$ for each $F \leq \lambda(S)\langle\langle m_i: S \rightarrow T_i^{i \in I} \rangle\rangle$ and, by Proposition 14.2-11, that $(\lambda(x)x, \lambda(x)x) \in [[m_i; T_i^{i \in I}]] \rightarrow (\{(\perp, \perp)\} \cup (L \rightarrow D) \times (L \rightarrow D))$.

If $F \leq \lambda(S)\langle\langle m_i: S \rightarrow T_i^{i \in I} \rangle\rangle$, then $F(S)$ has the form $\langle\langle m_j: S \rightarrow T_j^{j \in J} \rangle\rangle$. If $(x, x') \in \mu(S)F(S)$, unfolding yields $(x, x') \in \langle\langle m_j: (\mu(S)F(S)) \rightarrow T_j^{j \in J} \rangle\rangle$. By definition of $\langle\langle \dots \rangle\rangle$, we have $(x, x') \in \{(\perp, \perp)\} \cup (L \rightarrow D) \times (L \rightarrow D)$.

□

Proposition C.4-4

If $(x, x') \in [[m_i; T_i^{i \in I}]]$, then $x(m_k), x'(m_k) \in (D \rightarrow D)$ and $(x(m_k)(x), x'(m_k)(x')) \in T_k$ for all $k \in I$.

Proof

Again Proposition 14.2-11 applies. It suffices to prove that if $(x, x') \in \mu(S)F(S)$, $F \leq \lambda(S)\langle\langle m_i: S \rightarrow T_i^{i \in I} \rangle\rangle$, and $k \in I$, then $(x(m_k), x'(m_k)) \in (D \rightarrow D) \times (D \rightarrow D)$ and $(x(m_k)(x), x'(m_k)(x')) \in T_k$.

If $F \leq \lambda(S)\langle\langle m_i: S \rightarrow T_i^{i \in I} \rangle\rangle$, then $F(S)$ has the form $\langle\langle m_k: S \rightarrow T_k, \dots \rangle\rangle$. If $(x, x') \in \mu(S)F(S)$, unfolding yields $(x, x') \in \langle\langle m_k: (\mu(S)F(S)) \rightarrow T_k, \dots \rangle\rangle$. By Proposition C.4-3, either $(x, x') = (\perp, \perp)$ or $x, x' \in (L \rightarrow D)$. If $(x, x') = (\perp, \perp)$, then the result is trivial. So we assume that $x, x' \in (L \rightarrow D)$. An application yields $(x(m_k), x'(m_k)) \in (\mu(S)F(S)) \rightarrow T_k$, hence $(x(m_k), x'(m_k)) \in (D \rightarrow D) \times (D \rightarrow D)$; a second application yields $(x(m_k)(x), x'(m_k)(x')) \in T_k$.

□

Proposition C.4-5

If $(x, x') \in [[m_i; T_i^{i \in I}]]$ and $(y, y') \in [[m_i; T_i^{i \in I}]] \rightarrow T_k$ for some $k \in I$, then $(x(m_k \leftarrow y), x'(m_k \leftarrow y')) \in [[m_i; T_i^{i \in I}]]$.

Proof

Again Proposition 14.2-11 applies. It suffices to prove that if $(x, x') \in \mu(S)F(S)$, $F \leq \lambda(S)\langle\langle m_i: S \rightarrow T_i^{i \in I} \rangle\rangle$, $k \in I$, and $(y, y') \in [[m_i; T_i^{i \in I}]] \rightarrow T_k$, then $(x(m_k \leftarrow y), x'(m_k \leftarrow y')) \in [[m_i; T_i^{i \in I}]]$.

If $F \leq \lambda(S)(\langle m_i : S \rightarrow T_i \rangle^{ieI})$, then $F(S)$ has the form $\langle (m_k : S \rightarrow T_k, \dots) \rangle$. If $(x, x') \in \mu(S)F(S)$, unfolding yields $(x, x') \in \langle (m_k : (\mu(S)F(S)) \rightarrow T_k, \dots) \rangle$. Now $(y, y') \in \langle (m_i : T_i \rangle^{ieI}) \rightarrow T_k \rangle$ implies $(y, y') \in (\mu(S)F(S)) \rightarrow T_k$. So by Proposition C.4-1 $(x(m_k \leftarrow y), x'(m_k \leftarrow y')) \in \langle (m_i : (\mu(S)F(S)) \rightarrow T_i \rangle^{ieI})$. By folding, $(x(m_k \leftarrow y), x'(m_k \leftarrow y')) \in \mu(S)F(S) \subseteq \langle (m_i : T_i \rangle^{ieI})$.

□

Next we state a substitution lemma.

Lemma C.4-6 (Substitution)

- (1) $\llbracket B \rrbracket_{\eta(X \leftarrow [C]_{\eta})} = \llbracket B[C] \rrbracket_{\eta}$
- (2) $\llbracket b[X] \rrbracket_{\rho} = \llbracket b[C] \rrbracket_{\rho}$
- (3) $\llbracket b[X] \rrbracket_{\rho(X \leftarrow [c]_{\rho})} = \llbracket b[c] \rrbracket_{\rho}$

□

Finally, we check the equational rules one by one.

Lemma C.4-7

The equational rules are sound.

Proof

We treat most of the rules, omitting only the trivial (Eq x), (Eq Symm), and (Eq Trans). We treat even the basic rules from the λ -calculus, because our induction is superficially different to those of Amadio and Cardone, for example.

$$\begin{array}{c} \text{(Eq Subsumption)} \\ \frac{E \vdash a \leftrightarrow a' : A \quad E \vdash A \subset B}{E \vdash a \leftrightarrow a' : B} \end{array}$$

This rule is immediately sound, since subtyping is simply inclusion.

$$\begin{array}{c} \text{(Eq Top)} \\ \frac{E \vdash a : A \quad E \vdash b : B}{E \vdash a \leftrightarrow b : \text{Top}} \end{array}$$

From the hypotheses we know that $\llbracket a \rrbracket_{\rho}$ and $\llbracket b \rrbracket_{\rho'}$ are not $*$. Hence $(\llbracket a \rrbracket_{\rho}, \llbracket b \rrbracket_{\rho'}) \in \llbracket \text{Top} \rrbracket_{\eta}$.

$$\begin{array}{c} \text{(Eq Fun)} \\ \frac{E, x:A \vdash b \leftrightarrow b' : B}{E \vdash \lambda(x:A)b \leftrightarrow \lambda(x:A)b' : A \rightarrow B} \end{array}$$

Consider η and (ρ, ρ') consistent with E , and $(v, v') \in \llbracket A \rrbracket_{\eta}$. Let $\theta = \rho(x \leftarrow v)$ and $\theta' = \rho'(x \leftarrow v')$. We have that $(\llbracket \lambda(x:A)b \rrbracket_{\rho})v = (\lambda(u)\llbracket b \rrbracket_{\rho(x \leftarrow u)})(v)$, and hence $\llbracket \lambda(x:A)b \rrbracket_{\rho}(v) = \llbracket b \rrbracket_{\rho}$. Similarly, $\llbracket \lambda(x:A)b \rrbracket_{\rho'}(v') = \llbracket b \rrbracket_{\rho'}$. Since η and (θ, θ') are consistent with $E, x:A$, the hypothesis yields $(\llbracket b \rrbracket_{\theta}, \llbracket b' \rrbracket_{\theta'}) \in \llbracket B \rrbracket_{\eta}$, so $(\llbracket \lambda(x:A)b \rrbracket_{\rho}(v), \llbracket \lambda(x:A)b' \rrbracket_{\rho'}(v')) \in \llbracket B \rrbracket_{\eta}$, and $(\llbracket \lambda(x:A)b \rrbracket_{\rho}, \llbracket \lambda(x:A)b' \rrbracket_{\rho'}) \in \llbracket A \rrbracket_{\eta} \rightarrow \llbracket B \rrbracket_{\eta}$. That is, $(\llbracket \lambda(x:A)b \rrbracket_{\rho}, \llbracket \lambda(x:A)b' \rrbracket_{\rho'}) \in \llbracket A \rightarrow B \rrbracket_{\eta}$.

(Eq Appl)

$$\frac{E \vdash b \leftrightarrow b' : A \rightarrow B \quad E \vdash a \leftrightarrow a' : A}{E \vdash b(a) \leftrightarrow b'(a') : B}$$

The first hypothesis yields $(\llbracket b \rrbracket_\rho, \llbracket b' \rrbracket_{\rho'}) \in \llbracket A \rightarrow B \rrbracket_\eta$, so $(\llbracket b \rrbracket_\rho, \llbracket b' \rrbracket_{\rho'}) \in \llbracket A \rrbracket_\eta \rightarrow \llbracket B \rrbracket_\eta$. Hence $\llbracket b \rrbracket_\rho, \llbracket b' \rrbracket_{\rho'} \in (D \rightarrow D)$. We obtain $(\llbracket a \rrbracket_\rho, \llbracket a' \rrbracket_{\rho'}) \in \llbracket A \rrbracket_\eta$ from the second hypothesis. Hence $(\llbracket b \rrbracket_\rho(\llbracket a \rrbracket_\rho), \llbracket b' \rrbracket_{\rho'}(\llbracket a' \rrbracket_{\rho'})) \in \llbracket B \rrbracket_\eta$, with $\llbracket b(a) \rrbracket_\rho = \llbracket b \rrbracket_\rho(\llbracket a \rrbracket_\rho)$ and $\llbracket b'(a') \rrbracket_{\rho'} = \llbracket b' \rrbracket_{\rho'}(\llbracket a' \rrbracket_{\rho'})$. Finally, we derive $(\llbracket b(a) \rrbracket_\rho, \llbracket b'(a') \rrbracket_{\rho'}) \in \llbracket B \rrbracket_\eta$.

(Eq Fun2<:)

$$\frac{E, X <: A \vdash b \leftrightarrow b' : B}{E \vdash \lambda(X <: A)b \leftrightarrow \lambda(X <: A)b' : \forall(X <: A)B}$$

Consider η and (ρ, ρ') consistent with E , and $R \in \text{CUPER}$ such that $R \subseteq \llbracket A \rrbracket_\eta$. Let $\vartheta = \eta(X \leftarrow R)$. We have that $\llbracket \lambda(X <: A)b \rrbracket_\rho = \llbracket b \rrbracket_\rho$. Similarly, $\llbracket \lambda(X <: A)b' \rrbracket_{\rho'} = \llbracket b' \rrbracket_{\rho'}$. Since ϑ and (ρ, ρ') are consistent with E , $X <: A$, the hypothesis applies, yielding $(\llbracket b \rrbracket_\rho, \llbracket b' \rrbracket_{\rho'}) \in \llbracket B \rrbracket_\vartheta$, hence $(\llbracket \lambda(X <: A)b \rrbracket_\rho, \llbracket \lambda(X <: A)b' \rrbracket_{\rho'}) \in \llbracket B \rrbracket_\vartheta$. Since this is true independently of the choice of R , $(\llbracket \lambda(X <: A)b \rrbracket_\rho, \llbracket \lambda(X <: A)b' \rrbracket_{\rho'}) \in \llbracket \forall(X <: A)B \rrbracket_\eta$.

(Eq Appl2<:)

$$\frac{E \vdash b \leftrightarrow b' : \forall(X <: A)B[X] \quad E \vdash C <: A}{E \vdash b(C) \leftrightarrow b'(C) : B[C]}$$

Since $\llbracket C \rrbracket_\eta \subseteq \llbracket A \rrbracket_\eta$, we have that $\llbracket \forall(X <: A)B \rrbracket_\eta \subseteq \llbracket B \rrbracket_{\eta(X \leftarrow \llbracket C \rrbracket_\eta)}$, while $\llbracket B \rrbracket_{\eta(X \leftarrow \llbracket C \rrbracket_\eta)} = \llbracket B[C] \rrbracket_\eta$ by Lemma C.4-6. Moreover, $\llbracket b \rrbracket_\rho = \llbracket b(C) \rrbracket_\rho$ and $\llbracket b' \rrbracket_{\rho'} = \llbracket b'(C) \rrbracket_{\rho'}$. The hypothesis yields $(\llbracket b \rrbracket_\rho, \llbracket b' \rrbracket_{\rho'}) \in \llbracket \forall(X <: A)B \rrbracket_\eta$, and hence $(\llbracket b(C) \rrbracket_\rho, \llbracket b'(C) \rrbracket_{\rho'}) \in \llbracket B[C] \rrbracket_\eta$.

(Eq Pack<:)

$$\frac{\begin{array}{c} E \vdash C <: A' \quad E \vdash A' <: A \quad E, X <: A' \vdash B'[X] <: B[X] \\ E \vdash b\{C\} \leftrightarrow b'\{C\} : B'\{C\} \end{array}}{\begin{array}{c} E \vdash \text{pack } X <: A = C \text{ with } b\{X\}; B\{X\} \leftrightarrow \\ \text{pack } X <: A' = C \text{ with } b'\{X\}; B'\{X\} : \exists(X <: A)B[X] \end{array}}$$

This case is similar to that for (Eq Appl2<:)), noting that $\llbracket B' \rrbracket_{\eta(X \leftarrow \llbracket C \rrbracket_\eta)} \subseteq \llbracket \exists(X <: A)B \rrbracket_\eta$, $\llbracket b\{C\} \rrbracket_\rho = \llbracket \text{pack } X <: A = C \text{ with } b\{X\}; B \rrbracket_\rho$, and $\llbracket b'\{C\} \rrbracket_{\rho'} = \llbracket \text{pack } X <: A' = C \text{ with } b'\{X\}; B' \rrbracket_{\rho'}$.

(Eq Open<:)

$$\frac{\begin{array}{c} E \vdash c \leftrightarrow c' : \exists(X <: A)B \quad E \vdash D \quad E, X <: A, x: B \vdash d \leftrightarrow d' : D \\ E \vdash \text{open } c \text{ as } X <: A, x: B \text{ in } d: D \leftrightarrow \text{open } c' \text{ as } X <: A, x: B \text{ in } d': D : D \end{array}}{E \vdash \text{open } c \text{ as } X <: A, x: B \text{ in } d: D \leftrightarrow \text{open } c' \text{ as } X <: A, x: B \text{ in } d': D : D}$$

By the first hypothesis, we have that $(\llbracket c \rrbracket_\rho, \llbracket c' \rrbracket_{\rho'}) \in \sqcup_{R \in \text{CUPER}, R \subseteq \llbracket A \rrbracket_\eta} \llbracket B \rrbracket_{\eta(X \leftarrow R)}$, and we wish to derive that $(\llbracket d \rrbracket_{\rho(x \leftarrow \llbracket c \rrbracket_\rho)}, \llbracket d' \rrbracket_{\rho'(x \leftarrow \llbracket c' \rrbracket_{\rho'})}) \in \llbracket D \rrbracket_\eta$. By Proposition 14.2-11, it suffices

to prove that if $(v, v') \in \llbracket B \rrbracket_{\eta(X \leftarrow R)}$ for some $R \subseteq \llbracket A \rrbracket_{\eta}$ then $(\llbracket d \rrbracket_{\rho(x \leftarrow v)}, \llbracket d' \rrbracket_{\rho'(x \leftarrow v')}) \in \llbracket D \rrbracket_{\eta}$. If η and (ρ, ρ') are consistent with E , then $\eta(X \leftarrow R)$ and $(\rho(x \leftarrow v), \rho'(x \leftarrow v'))$ are consistent with E , $X <: A$, $x : B$. The third hypothesis yields $(\llbracket d \rrbracket_{\rho(x \leftarrow v)}, \llbracket d' \rrbracket_{\rho'(x \leftarrow v')}) \in \llbracket D \rrbracket_{\eta(X \leftarrow R)}$, and since X cannot occur in D (because $E \vdash D$) we obtain $(\llbracket d \rrbracket_{\rho(x \leftarrow v)}, \llbracket d' \rrbracket_{\rho'(x \leftarrow v')}) \in \llbracket D \rrbracket_{\eta}$.

$$\text{(Eq Object) (where } A \equiv [l_i : B_i]_{i \in 1..n} \text{)} \\ \frac{E, x_i : A \vdash b_i \leftrightarrow b'_i : B_i \quad \forall i \in 1..n}{E \vdash [l_i = \zeta(x_i : A) b_i]_{i \in 1..n} \leftrightarrow [l_i = \zeta(x_i : A) b'_i]_{i \in 1..n} : A}$$

The first hypothesis yields $(\llbracket \lambda(x_i : A) b_i \rrbracket_{\rho}, \llbracket \lambda(x_i : A) b_i \rrbracket_{\rho'}) \in \llbracket A \rightarrow B_i \rrbracket_{\eta}$. (The argument is a special case of the one for (Eq Fun).) According to the definition of the semantics, this means that $(\llbracket \lambda(x_i : A) b_i \rrbracket_{\rho}, \llbracket \lambda(x_i : A) b_i \rrbracket_{\rho'}) \in ([m_i : \llbracket B_i \rrbracket_{\eta}]^{i \in 1..n}) \rightarrow \llbracket B_i \rrbracket_{\eta}$ for all $i \in 1..n$. Proposition C.4-2 yields $(\langle \langle l_i = \llbracket \lambda(x_i : A) b_i \rrbracket_{\rho} \rangle \rangle, \langle \langle l_i = \llbracket \lambda(x_i : A) b_i \rrbracket_{\rho'} \rangle \rangle) \in ([l_i : \llbracket B_i \rrbracket_{\eta}]^{i \in 1..n})$. That is, $(\llbracket l_i = \zeta(x_i : A) b_i \rrbracket_{\rho}, \llbracket l_i = \zeta(x_i : A) b_i \rrbracket_{\rho'}) \in \llbracket [l_i : B_i]_{i \in 1..n} \rrbracket_{\eta}$.

$$\text{(Eq Sub Object) (where } A \equiv [l_i : B_i]_{i \in 1..n}, A' \equiv [l_i : B_i]_{i \in 1..n+m} \text{)} \\ \frac{E, x_i : A \vdash b_i : B_i \quad \forall i \in 1..n \quad E, x_j : A' \vdash b_j : B_j \quad \forall j \in n+1..n+m}{E \vdash [l_i = \zeta(x_i : A) b_i]_{i \in 1..n} \leftrightarrow [l_i = \zeta(x_i : A') b_i]_{i \in 1..n+m} : A}$$

The first hypothesis yields $(\llbracket \lambda(x_i : A) b_i \rrbracket_{\rho}, \llbracket \lambda(x_i : A) b_i \rrbracket_{\rho'}) \in \llbracket A \rightarrow B_i \rrbracket_{\eta}$. (The argument is a special case of the one for (Eq Fun).) According to the definition of the semantics, this means that $(\llbracket \lambda(x_i : A) b_i \rrbracket_{\rho}, \llbracket \lambda(x_i : A) b_i \rrbracket_{\rho'}) \in ([m_i : \llbracket B_i \rrbracket_{\eta}]^{i \in 1..n}) \rightarrow \llbracket B_i \rrbracket_{\eta}$ for all $i \in 1..n$. Proposition C.4-2 yields $(\langle \langle l_i = \llbracket \lambda(x_i : A) b_i \rrbracket_{\rho} \rangle \rangle, \langle \langle l_i = \llbracket \lambda(x_i : A) b_i \rrbracket_{\rho'} \rangle \rangle, l_j = \llbracket \lambda(x_j : A') b_j \rrbracket_{\rho'}, j \in n+1..n+m) \rangle \rangle \in ([l_i : \llbracket B_i \rrbracket_{\eta}]^{i \in 1..n})$. That is, $(\llbracket l_i = \zeta(x_i : A) b_i \rrbracket_{\rho}, \llbracket l_i = \zeta(x_i : A') b_i \rrbracket_{\rho}, l_j = \zeta(x_j : A') b_j \rrbracket_{\rho'}, j \in n+1..n+m) \rrbracket_{\rho'} \in \llbracket [l_i : B_i]_{i \in 1..n} \rrbracket_{\eta}$. (Note that the second hypothesis is not used.)

$$\text{(Eq Select)} \\ \frac{E \vdash a \leftrightarrow a' : [l_i : B_i]_{i \in 1..n} \quad j \in 1..n}{E \vdash a.l_j \leftrightarrow a'.l_j : B_j}$$

The hypothesis yields $(\llbracket a \rrbracket_{\rho}, \llbracket a' \rrbracket_{\rho'}) \in \llbracket [l_i : B_i]_{i \in 1..n} \rrbracket_{\eta}$, so $(\llbracket a \rrbracket_{\rho}, \llbracket a' \rrbracket_{\rho'}) \in ([l_i : \llbracket B_i \rrbracket_{\eta}]^{i \in 1..n})$. By Proposition C.4-3, $\llbracket a \rrbracket_{\rho} = \llbracket a' \rrbracket_{\rho'} = \perp$ or $\llbracket a \rrbracket_{\rho}, \llbracket a' \rrbracket_{\rho'} \in (L \rightarrow D)$. If $\llbracket a \rrbracket_{\rho} = \llbracket a' \rrbracket_{\rho'} = \perp$, then immediately $\llbracket a.l_j \rrbracket_{\rho} = \llbracket a'.l_j \rrbracket_{\rho'} = \perp$ so $(\llbracket a.l_j \rrbracket_{\rho}, \llbracket a'.l_j \rrbracket_{\rho'}) \in \llbracket B_j \rrbracket_{\eta}$. Otherwise, $\llbracket a \rrbracket_{\rho}, \llbracket a' \rrbracket_{\rho'} \in (L \rightarrow D)$. By Proposition C.4-4, $\llbracket a \rrbracket_{\rho}(l_j), \llbracket a' \rrbracket_{\rho'}(l_j) \in (D \rightarrow D)$ and $(\llbracket a \rrbracket_{\rho}(l_j)(\llbracket a \rrbracket_{\rho}), \llbracket a' \rrbracket_{\rho'}(l_j)(\llbracket a' \rrbracket_{\rho'})) \in \llbracket B_j \rrbracket_{\eta}$. Since $\llbracket a \rrbracket_{\rho}, \llbracket a' \rrbracket_{\rho'} \in (L \rightarrow D)$ and $\llbracket a \rrbracket_{\rho}(l_j), \llbracket a' \rrbracket_{\rho'}(l_j) \in (D \rightarrow D)$, this last result means $(\llbracket a.l_j \rrbracket_{\rho}, \llbracket a'.l_j \rrbracket_{\rho'}) \in \llbracket B_j \rrbracket_{\eta}$.

$$\text{(Eq Update) (where } A \equiv [l_i : B_i]_{i \in 1..n} \text{)} \\ \frac{E \vdash a \leftrightarrow a' : A \quad E, x : A \vdash b \leftrightarrow b' : B_j \quad j \in 1..n}{E \vdash a.l_j \zeta(x : A) b \leftrightarrow a'.l_j \zeta(x : A) b' : A}$$

One hypothesis yields $(\llbracket a \rrbracket_{\rho}, \llbracket a' \rrbracket_{\rho'}) \in \llbracket [l_i : B_i]_{i \in 1..n} \rrbracket_{\eta}$, that is, $(\llbracket a \rrbracket_{\rho}, \llbracket a' \rrbracket_{\rho'}) \in ([l_i : \llbracket B_i \rrbracket_{\eta}]^{i \in 1..n})$. The second one yields $(\llbracket \lambda(x : A) b \rrbracket_{\rho}, \llbracket \lambda(x : A) b' \rrbracket_{\rho'}) \in \llbracket [l_i : B_i]_{i \in 1..n} \rrbracket \rightarrow C_j \rrbracket_{\eta}$, that is, $(\llbracket \lambda(x : A) b \rrbracket_{\rho},$

$\llbracket \lambda(x:A)b' \rrbracket_{\rho'} \in \llbracket [l_i : \llbracket B_i \rrbracket_{\eta}^{i \in 1..n}] \rightarrow \llbracket B_j \rrbracket_{\eta} \}$. (The argument is a special case of the one for (Eq Fun).) By Proposition C.4-3, $\llbracket a \rrbracket_{\rho} = \llbracket a' \rrbracket_{\rho'} = \perp$ or $\llbracket a \rrbracket_{\rho}, \llbracket a' \rrbracket_{\rho'} \in (L \rightarrow D)$. If $\llbracket a \rrbracket_{\rho} = \llbracket a' \rrbracket_{\rho'} = \perp$, then immediately $\llbracket a, l_i \not\models \zeta(x:A)b \rrbracket_{\rho} = \llbracket a', l_i \not\models \zeta(x:A)b' \rrbracket_{\rho'} = \perp$, so $(\llbracket a, l_i \not\models \zeta(x:A)b \rrbracket_{\rho}, \llbracket a', l_i \not\models \zeta(x:A)b' \rrbracket_{\rho'}) \in \llbracket [l_i : B_i]_{\eta}^{i \in 1..n} \rrbracket_{\eta}$. Otherwise, $\llbracket a \rrbracket_{\rho}, \llbracket a' \rrbracket_{\rho'} \in (L \rightarrow D)$. Proposition C.4-5 yields $(\llbracket a \rrbracket_{\rho}(l_j \leftarrow \llbracket \lambda(x:A)b \rrbracket_{\rho}), \llbracket a' \rrbracket_{\rho'}(l_j \leftarrow \llbracket \lambda(x:A)b' \rrbracket_{\rho'})) \in \llbracket [l_i : B_i]_{\eta}^{i \in 1..n} \rrbracket_{\eta}$. Since $\llbracket a \rrbracket_{\rho}, \llbracket a' \rrbracket_{\rho'} \in (L \rightarrow D)$, this last result means $(\llbracket a, l_i \not\models \zeta(x:A)b \rrbracket_{\rho}, \llbracket a', l_i \not\models \zeta(x:A)b' \rrbracket_{\rho'}) \in \llbracket [l_i : B_i]_{\eta}^{i \in 1..n} \rrbracket_{\eta}$.

$$\frac{(\text{Eq Fold}) \quad (\text{where } A \equiv \mu(X)B\{X\})}{E \vdash b \leftrightarrow b' : B\{A\}}$$

$$\frac{}{E \vdash \text{fold}(A, b) \leftrightarrow \text{fold}(A, b') : A}$$

According to the hypothesis, $(\llbracket b \rrbracket_{\rho}, \llbracket b' \rrbracket_{\rho'}) \in \llbracket B\{A\} \rrbracket_{\eta}$, that is, $(\llbracket b \rrbracket_{\rho}, \llbracket b' \rrbracket_{\rho'}) \in \llbracket B \rrbracket_{\eta(X \leftarrow \llbracket A \rrbracket_{\eta})}$ by Lemma C.4-6. Therefore $(\lambda(v)v\llbracket b \rrbracket_{\rho}, \lambda(v')v'\llbracket b' \rrbracket_{\rho'}) \in \text{Univ} \rightarrow \llbracket B \rrbracket_{\eta(X \leftarrow \llbracket A \rrbracket_{\eta})}$, that is, $(\llbracket \text{fold}(A, b) \rrbracket_{\rho}, \llbracket \text{fold}(A, b') \rrbracket_{\rho'}) \in \text{Univ} \rightarrow \llbracket B \rrbracket_{\eta(X \leftarrow \llbracket A \rrbracket_{\eta})}$. Since $\text{Univ} \rightarrow \llbracket B \rrbracket_{\eta(X \leftarrow \llbracket A \rrbracket_{\eta})} = \llbracket A \rrbracket_{\eta}$, we obtain that $(\llbracket \text{fold}(A, b) \rrbracket_{\rho}, \llbracket \text{fold}(A, b') \rrbracket_{\rho'}) \in \llbracket A \rrbracket_{\eta}$.

$$\frac{(\text{Eq Unfold}) \quad (\text{where } A \equiv \mu(X)B\{X\})}{E \vdash a \leftrightarrow a' : A}$$

$$\frac{}{E \vdash \text{unfold}(a) \leftrightarrow \text{unfold}(a') : B\{A\}}$$

According to the hypothesis, $(\llbracket a \rrbracket_{\rho}, \llbracket a' \rrbracket_{\rho'}) \in \llbracket A \rrbracket_{\eta}$. Since $\llbracket A \rrbracket_{\eta} = \text{Univ} \rightarrow \llbracket B \rrbracket_{\eta(X \leftarrow \llbracket A \rrbracket_{\eta})}$ and $(\perp, \perp) \in \text{Univ}$, we obtain that $(\llbracket a \rrbracket_{\rho}(\perp), \llbracket a' \rrbracket_{\rho'}(\perp)) \in \llbracket B \rrbracket_{\eta(X \leftarrow \llbracket A \rrbracket_{\eta})}$. By Lemma C.4-6, this means that $(\llbracket a \rrbracket_{\rho}(\perp), \llbracket a' \rrbracket_{\rho'}(\perp)) \in \llbracket B\{A\} \rrbracket_{\eta}$. Since $\llbracket a \rrbracket_{\rho}(\perp) = \llbracket \text{unfold}(a) \rrbracket_{\rho}$ and $\llbracket a' \rrbracket_{\rho}(\perp) = \llbracket \text{unfold}(a') \rrbracket_{\rho}$, we obtain that $(\llbracket \text{unfold}(a) \rrbracket_{\rho}, \llbracket \text{unfold}(a') \rrbracket_{\rho}) \in \llbracket B\{A\} \rrbracket_{\eta}$.

$$\frac{(\text{Eval Beta})}{E \vdash \lambda(x:A)b\{x\} : A \rightarrow B \quad E \vdash a : A}$$

$$\frac{}{E \vdash (\lambda(x:A)b\{x\})(a) \leftrightarrow b\{a\} : B}$$

Consider η and (ρ, ρ') consistent with E . As in the case of application, the hypotheses yield $(\llbracket (\lambda(x:A)b)(a) \rrbracket_{\rho}, \llbracket (\lambda(x:A)b)(a) \rrbracket_{\rho'}) \in \llbracket B \rrbracket_{\eta}$. Because of the form of $\lambda(x:A)b$, we have that $\llbracket \lambda(x:A)b \rrbracket_{\rho'} \in (D \rightarrow D)$ and hence $\llbracket (\lambda(x:A)b)(a) \rrbracket_{\rho'} = \llbracket b \rrbracket_{\rho'(\alpha \leftarrow \llbracket a \rrbracket_{\rho})}$. Lemma C.4-6 yields $\llbracket (\lambda(x:A)b\{x\})(a) \rrbracket_{\rho'} = \llbracket b\{a\} \rrbracket_{\rho'}$. Therefore $(\llbracket (\lambda(x:A)b\{x\})(a) \rrbracket_{\rho}, \llbracket b\{a\} \rrbracket_{\rho'}) \in \llbracket B \rrbracket_{\eta}$.

$$\frac{(\text{Eval Eta})}{E \vdash b : A \rightarrow B \quad x \notin \text{dom}(E)}$$

$$\frac{}{E \vdash \lambda(x:A)b(x) \leftrightarrow b : A \rightarrow B}$$

Let $(v, v') \in \llbracket A \rrbracket_{\eta}$. The hypothesis yields that $\llbracket b \rrbracket_{\rho}, \llbracket b \rrbracket_{\rho'} \in (D \rightarrow D)$. We obtain that $\llbracket \lambda(x:A)b(x) \rrbracket_{\rho} = \lambda(u)\llbracket b(x) \rrbracket_{\rho(x \leftarrow u)}$ and hence $\llbracket \lambda(x:A)b(x) \rrbracket_{\rho} = \lambda(u)\llbracket b \rrbracket_{\rho(x \leftarrow u)}(\llbracket x \rrbracket_{\rho(x \leftarrow u)})$ and, since x is assumed not to occur free in b , $\llbracket \lambda(x:A)b(x) \rrbracket_{\rho} = \lambda(u)\llbracket b \rrbracket_{\rho}(u)$. The hypothesis also gives $(\llbracket b \rrbracket_{\rho}(v), \llbracket b' \rrbracket_{\rho'}(v')) \in \llbracket B \rrbracket_{\eta}$, hence $(\llbracket \lambda(x:A)b(x) \rrbracket_{\rho}(v), \llbracket b' \rrbracket_{\rho'}(v')) \in \llbracket B \rrbracket_{\eta}$. Finally, we have $(\llbracket \lambda(x:A)b(x) \rrbracket_{\rho}, \llbracket b' \rrbracket_{\rho'}) \in \llbracket A \rrbracket_{\eta} \rightarrow \llbracket B \rrbracket_{\eta}$. That is, $(\llbracket \lambda(x:A)b(x) \rrbracket_{\rho}, \llbracket b' \rrbracket_{\rho'}) \in \llbracket A \rightarrow B \rrbracket_{\eta}$.

$$\begin{array}{c}
 (\text{Eval Beta2}::) \\
 \dfrac{E \vdash \lambda(X <: A)b[X] : \forall(X <: A)B[X] \quad E \vdash C <: A}{E \vdash (\lambda(X <: A)b)(C) \leftrightarrow b[C] : B[C]}
 \end{array}$$

As in the case of second-order application, we use that $\llbracket \forall(X <: A)B[X] \rrbracket_{\eta} \subseteq \llbracket B[C] \rrbracket_{\eta}$. Moreover, $\llbracket (\lambda(X <: A)b)(C) \rrbracket_{\rho} = \llbracket \lambda(X <: A)b \rrbracket_{\rho}$ and $\llbracket b[C] \rrbracket_{\rho'} = \llbracket \lambda(X <: A)b \rrbracket_{\rho'}$. Hence, using the hypothesis, we obtain $(\llbracket (\lambda(X <: A)b)(C) \rrbracket_{\rho}, \llbracket b[C] \rrbracket_{\rho'}) \in \llbracket B[C] \rrbracket_{\eta}$.

$$\begin{array}{c}
 (\text{Eval Eta2}::) \\
 \dfrac{E \vdash b : \forall(X <: A)B \quad X \notin \text{dom}(E)}{E \vdash \lambda(X <: A)b(X) \leftrightarrow b : \forall(X <: A)B}
 \end{array}$$

This case is trivial, since $\llbracket \lambda(X <: A)b(X) \rrbracket_{\rho} = \llbracket b \rrbracket_{\rho}$.

$$\begin{array}{c}
 (\text{Eval Unpack}::) \quad (\text{where } c \equiv \text{pack } X <: A = C \text{ with } b[X]; B[X]) \\
 \dfrac{E \vdash c : \exists(X <: A)B[X] \quad E \vdash D \quad E, X <: A, x: B[X] \vdash d[X, x] : D}{E \vdash \text{open } c \text{ as } X <: A, x: B[X] \text{ in } d[X, x]; D \leftrightarrow d[C, b[C]] : D}
 \end{array}$$

By the first hypothesis, we have that $(\llbracket c \rrbracket_{\rho}, \llbracket c \rrbracket_{\rho'}) \in \sqcup_{R \in \text{CUPER}, R \subseteq A} \llbracket B \rrbracket_{\eta(X \leftarrow R)}$. We first prove that $(\llbracket d \rrbracket_{\rho(x \leftarrow [c]_{\rho})}, \llbracket d \rrbracket_{\rho'(x \leftarrow [c]_{\rho'})}) \in \llbracket D \rrbracket_{\eta}$. By Proposition 14.2-11, it suffices to prove that if $(v, v') \in \llbracket B \rrbracket_{\eta(X \leftarrow R)}$ for some $R \subseteq \llbracket A \rrbracket_{\eta}$, then $(\llbracket d \rrbracket_{\rho(x \leftarrow v)}, \llbracket d' \rrbracket_{\rho'(x \leftarrow v')}) \in \llbracket D \rrbracket_{\eta}$. If η and (ρ, ρ') are consistent with E , then $\eta(X \leftarrow R)$ and $(\rho(x \leftarrow v), \rho'(x \leftarrow v'))$ are consistent with $E, X <: A, x: B$. The third hypothesis yields $(\llbracket d \rrbracket_{\rho(x \leftarrow v)}, \llbracket d' \rrbracket_{\rho'(x \leftarrow v')}) \in \llbracket D \rrbracket_{\eta(X \leftarrow R)}$. Since X cannot occur in D (because $E \vdash D$) we have $(\llbracket d \rrbracket_{\rho(x \leftarrow v)}, \llbracket d' \rrbracket_{\rho'(x \leftarrow v)}) \in \llbracket D \rrbracket_{\eta}$. Thus we obtain that $(\llbracket d \rrbracket_{\rho(x \leftarrow [c]_{\rho})}, \llbracket d \rrbracket_{\rho'(x \leftarrow [c]_{\rho'})}) \in \llbracket D \rrbracket_{\eta}$. By definition $\llbracket \text{open } (\text{pack } X <: A = C \text{ with } b[X]; B[X]) \text{ as } X <: A, x: B[X] \text{ in } d[X, x]; D \rrbracket_{\rho} = \llbracket d \rrbracket_{\rho(x \leftarrow [c]_{\rho})}$; in addition, since $\llbracket b \rrbracket_{\rho'} = \llbracket c \rrbracket_{\rho'}$, $\llbracket d[C, b[C]] \rrbracket_{\rho'} = \llbracket d \rrbracket_{\rho'(x \leftarrow [c]_{\rho})}$ by Lemma C.4-6. Hence we obtain that $(\llbracket \text{open } (\text{pack } X <: A = C \text{ with } b[X]; B[X]) \text{ as } X <: A, x: B[X] \text{ in } d[X, x]; D \rrbracket_{\rho}, \llbracket d[C, b[C]] \rrbracket_{\rho'}) \in \llbracket D \rrbracket_{\eta}$.

$$\begin{array}{c}
 (\text{Eval Repack}::) \\
 \dfrac{E \vdash b : \exists(X <: A)B[X] \quad E, y: \exists(X <: A)B[X] \vdash d[y] : D}{E \vdash \text{open } b \text{ as } X <: A, x: B[X] \text{ in } d[\text{pack } X' <: A = X \text{ with } x: B[X']] ; D \leftrightarrow d[b] : D}
 \end{array}$$

We have that $\llbracket \text{open } b \text{ as } X <: A, x: B[X] \text{ in } d[\text{pack } X' <: A = X \text{ with } x: B[X']] ; D \rrbracket_{\rho} = \llbracket d \rrbracket_{\rho(x \leftarrow [b]_{\rho})}$. The result thus follows from the hypothesis, by Lemma C.4-6.

$$\begin{array}{c}
 (\text{Eval Select}) \quad (\text{where } A \equiv [l_i: B_i]^{i \in 1..n}, a \equiv [l_i = \zeta(x_i: A') b_i | x_i]^{i \in 1..n+m}) \\
 \dfrac{E \vdash a : A \quad j \in 1..n}{E \vdash a.l_j \leftrightarrow b_j[a] : B_j}
 \end{array}$$

The hypothesis yields $(\llbracket a \rrbracket_{\rho}, \llbracket a \rrbracket_{\rho'}) \in \llbracket A \rrbracket_{\eta}$, that is, $(\langle l_i = \llbracket \lambda(x_i: A') b_i \rrbracket_{\rho} \rangle, \langle l_i = \llbracket \lambda(x_i: A') b_i \rrbracket_{\rho'} \rangle)$ $\in \llbracket [l_i: B_i]_{\eta} \rrbracket$. The argument for (Eq Select) gives $(\llbracket a.l_j \rrbracket_{\rho}, \llbracket a.l_j \rrbracket_{\rho'}) \in \llbracket B_j \rrbracket_{\eta}$. In addition, $\llbracket a \rrbracket_{\rho'}$

$\in (L \rightarrow D)$ and $[\lambda(x_j:A')b_j]_{\rho'} \in (D \rightarrow D)$. Hence $[a.l_j]_{\rho'} = [\lambda(x_j:A')b_j]_{\rho'}([a]_{\rho'})$, so $[a.l_j]_{\rho'} = [b_j]_{\rho'(x_j \leftarrow [a]_{\rho'})}$. Lemma C.4-6 yields $[a.l_j]_{\rho'} = [b_j[a]]_{\rho'}$, hence $([a.l_j]_{\rho'}, [b_j[a]]_{\rho'}) \in [B_j]_{\eta}$.

$$\text{(Eval Update) (where } A \equiv [l_i; B_i \text{ } i \in 1..n], \text{ } a \equiv [l_i = \zeta(x_i:A')b_i \text{ } i \in 1..n+m]) \\ \frac{E \vdash a : A \quad E, x:A \vdash b : B_j \quad j \in 1..n}{E \vdash a.l_i = \zeta(x:A)b \leftrightarrow [l_i = \zeta(x:A')b, l_i = \zeta(x_i:A')b_i \text{ } i \in (1..n+m) - \{j\}] : A}$$

The first hypothesis yields $([a]_{\rho}, [a]_{\rho'}) \in [A]_{\eta}$. That is, $((l_i = [\lambda(x_i:A)b_i]_{\rho}), ([l_i = [\lambda(x_i:A)b_i]_{\rho'}])) \in ([l_i; [B_i]_{\eta}])$. The argument for (Eq Update) gives $([a.l_i = \zeta(x:A)b]_{\rho}, [a.l_i = \zeta(x:A)b]_{\rho'}) \in [A]_{\eta}$. In addition, $[a]_{\rho'} \in (L \rightarrow D)$. Hence $[a.l_i = \zeta(x:A)b]_{\rho'} = [[l_i = \zeta(x:A')b, l_i = \zeta(x_i:A')b_i \text{ } i \in (1..n+m) - \{j\}]]_{\rho'}$ and $([a.l_i = \zeta(x:A)b]_{\rho}, [[l_i = \zeta(x:A')b, l_i = \zeta(x_i:A')b_i \text{ } i \in (1..n+m) - \{j\}]]_{\rho'}) \in [A]_{\eta}$.

$$\text{(Eval Fold) (where } A \equiv \mu(X)B\{X\}) \\ \frac{E \vdash a : A}{E \vdash \text{fold}(A, \text{unfold}(a)) \leftrightarrow a : A}$$

Since $[\text{fold}(A, \text{unfold}(a))]_{\rho} = \lambda(v)([a]_{\rho}(\perp))$, we prove that $(\lambda(v)([a]_{\rho}(\perp)), [a]_{\rho'}) \in [A]_{\eta}$ from the hypothesis that $([a]_{\rho}, [a]_{\rho'}) \in [A]_{\eta}$. Since $[A]_{\eta} = \text{Univ} \rightarrow [B]_{\eta(X \leftarrow [A]_{\eta})}$, we prove that if $(v, v') \in \text{Univ}$ then $((\lambda(v)([a]_{\rho}(\perp))(v), [a]_{\rho'}(v')) \in [B]_{\eta(X \leftarrow [A]_{\eta})}$, that is, that $([a]_{\rho}(\perp), [a]_{\rho'}(v')) \in [A]_{\eta}$. This follows immediately from $(\perp, v') \in \text{Univ}$ and $([a]_{\rho}, [a]_{\rho'}) \in [A]_{\eta}$.

$$\text{(Eval Unfold) (where } A \equiv \mu(X)B\{X\}) \\ \frac{E \vdash b : B\{A\}}{E \vdash \text{unfold}(\text{fold}(A, b)) \leftrightarrow b : B\{A\}}$$

Since $[\text{unfold}(\text{fold}(A, b))]_{\rho} = [b]_{\rho}$, we obtain that $([\text{unfold}(\text{fold}(A, b))]_{\rho}, [b]_{\rho'}) \in [B\{A\}]_{\eta}$ directly from the hypothesis that $([b]_{\rho}, [b]_{\rho'}) \in [B\{A\}]_{\eta}$.

□

List of Figures

2-1. Naive storage model.	12
2-2. Method suites.	13
2-3. Hierarchical method suites.	16
2-4. Collapsed method suites.	17
4-1. Embedding.	39
4-2. Delegation.	42
4-3. Reparenting.	46
4-4. Traits.	48
20-1. Binary trees.	294

List of Tables

2 Class-Based Languages

Argument for the covariance of $A \times B$	20
Argument for the co/contravariance of $A \rightarrow B$	21
Argument for the invariance of $A * B$	21

6 Untyped Calculi

Objects and methods	57
Terminology	58
Primitive semantics	59
Syntax of the ζ -calculus	60
Object scoping	61
Object substitution	61
Equational theory	63
Operational semantics	64
Relations between terms	66
Translation of the untyped λ -calculus	66
Translation of default parameters	67
Translation of call-by-keyword	68
Self-application semantics	76
Recursive-record semantics	77
Recursive-record semantics (update)	77

7 First-Order Calculi

Syntax fragment for Δ_{Ob}	80
Δ_{Ob}	81
Δ_x	82
Δ_K	82
Δ_{\rightarrow}	83
Syntax of the FOb_1 calculus	83
Operational semantics	86
$\Delta_=_$	89
$\Delta_{=x}$	89
$\Delta_{=\text{Ob}}$	89
$\Delta_{= \rightarrow}$	90
Translation of the first-order λ -calculus	91

8 Subtyping

$\Delta_{<}$	93
$\Delta_{<\rightarrow}$	94
$\Delta_{<;\text{Ob}}$	94
Modified rules for $\text{MinOb}_{1;<}$	96
$\Delta_{=;<}$	99
$\Delta_{=<;\text{Ob}}$	99
Δ_{Rcd}	106
$\Delta_{=\text{Rcd}}$	106
Rules for elder	108
Object types with variance annotations	110
Subtyping with variance annotations	110
Typing with variance annotations	111

9 Recursion

Δ_X	114
Δ_μ	114
$\Delta_{=\mu}$	114
Typed translation of the untyped λ -calculus	115
$\Delta_{<;X}$	116
$\Delta_{<;\mu}$	116
$\Delta_{=<;\mu}$	116
Syntax of the $\text{FOb}_{1;<;\mu}$ calculus	117
Scoping for the $\text{FOb}_{1;<;\mu}$ calculus	118
Restricted rules for $\text{rOb}_{1;<;\mu}$	119
Modified rule for <i>fold</i> for $\text{rMinOb}_{1;<;\mu}$	119
Operational semantics for <i>fold</i> and <i>unfold</i>	120
Environment substitution	120
Typing rule for <i>typecase</i>	126
Operational semantics for <i>typecase</i>	127

10 Untyped Imperative Calculi

Syntax of the $\text{imp}\varsigma$ -calculus	129
Syntax of the $\text{imp}\varsigma_f$ -calculus	130
Translation of $\text{imp}\varsigma_f$ into $\text{imp}\varsigma$	131
Syntax of the $\text{imp}\lambda$ -calculus	131
Translation of the $\text{imp}\lambda$ -calculus into the $\text{imp}\varsigma_f$ -calculus	132
Syntax of the $\text{imp}\lambda\varsigma_f$ -calculus	133
Operational semantics	136

11 First-Order Imperative Calculi

Typing rules	141
--------------	-----

Store typing	147
12 A First-Order Language	
Syntax of O-1 types	154
Syntax of O-1 terms	155
Judgments	159
Environments	159
Types	160
Subtyping	160
Terms	161
Translation of O-1 types	163
Translation of O-1 environments	163
Preliminary translation of O-1 terms	164
Translation of O-1 terms	164
13 Second-Order Calculi	
Δ_A	170
$\Delta=\!A$	170
$\Delta_{<\!A}$	171
$\Delta_{=<\!A}$	171
$\Delta_{<\!E}$	173
$\Delta_{=<\!E}$	174
Syntax of the $\text{FOb}_{<\!:\mu}$ calculus	176
Scoping for the $\text{FOb}_{<\!:\mu}$ calculus	176
Variant occurrences	177
Translation of the first-order λ -calculus with subtyping	179
The Self quantifier	181
Δ_ζ	182
$\Delta_{=\!\zeta}$	182
Syntax of the ζOb calculus	183
15 Definable Covariant Self Types	
ζ -Object operations	202
$\Delta_{\zeta+}$	203
$\Delta_{=\!\zeta+}$	203
Objects, with preliminary structural assumptions	217
Objects, with structural assumptions	218
16 Primitive Covariant Self Types	
Syntax of S types	223
Variant occurrences	223
Syntax of S terms	224
Operational semantics	224

Judgments	225
Environments, types, and subtypes	225
Terms with typing annotations	226
Syntax of type parameterization	228
Operational semantics for type parameterization	228
Quantifier rules	228
Variant occurrences for quantifiers	229
17 Imperative Calculi with Self Types	
Syntax of the imp_ζ -calculus	241
Operational semantics of method update	242
Terms	243
Syntax of the $\text{imp}_{\zeta\forall}$ -calculus	244
Operational semantics for polymorphism	244
Quantifier rules	244
Substitution of type stacks	247
Store typing	248
18 Interpretations of Object Calculi	
Syntax for separate fields and methods	258
Syntax with Self types	259
Untyped self-application interpretation	259
Untyped imperative self-application interpretation	260
Untyped recursive-record interpretation without update	260
Untyped recursive-record interpretation with internal field update	260
Untyped state-application interpretation	261
Untyped state-application interpretation (continued)	261
Untyped cyclic-record interpretation	262
Untyped split-method interpretation	263
Self-application interpretation	264
Recursive-record interpretation, without update	265
Recursive-record interpretation with internal field update	265
State-application interpretation	266
Cyclic-record interpretation	267
Split-method interpretation (initial version)	268
Split-method interpretation	269
Split-method interpretation (with Self types)	270
Imperative self-application interpretation	271
19 A Second-Order Language	
Syntax of O-2 types	274
Syntax of O-2 terms	275
Judgments	279

Environments	279
Types	279
Subtyping	279
Terms	280
Translation of O-2 types	282
Translation of O-2 environments	282
Translation of O-2 terms	283

20 A Higher-Order Calculus

Syntax of $Ob_{\omega<\mu}$	288
Results	289
Operational semantics	289
Judgments for $Ob_{\omega<\mu}$	290
Environment formation	291
Kind formation	291
Constructor formation	291
Constructor equivalence	291
Constructor inclusion	292
Term typing	293
Object-oriented binary trees	294
Object-oriented binary-tree operator	295
Object-oriented binary-tree class type	296
Object-oriented binary-tree class	297
Least upper bounds and normal forms	299
Judgments for the normal system	299
Normal constructor inclusion	300

21 A Language with Matching

Syntax of O-3 types	306
Syntax of O-3 terms	306
Judgments	311
Environments	311
Types	312
Object types	312
Subtyping	312
Matching	313
Terms	313
Translation summary	316
Translations for O-3	317
Translation of O-3 types	318
Translation of O-3 environments	318
Translation of O-3 terms	318
Translation of O-3 judgments	319

A Fragments

$\Delta_{\text{Ob}} \cup \Delta_{<:\text{Ob}}$	329
$\Delta_{=\text{Ob}} \cup \Delta_{=;<:\text{Ob}}$	329
Δ_x	330
Δ_K	330
Δ_\rightarrow	330
$\Delta_<$	330
$\Delta_{<:\rightarrow}$	331
Δ_X	331
$\Delta_{<:X}$	331
Δ_μ	331
$\Delta_{<:\mu}$	331
Δ_\forall	332
$\Delta_{<:\forall}$	332
Δ_\exists	332
$\Delta_{<:\exists}$	332
Δ_\neg	333
$\Delta_{=x}$	333
$\Delta_{=\rightarrow}$	333
$\Delta_{=;<}$	333
$\Delta_{=\mu}$ (same as $\Delta_{=;<:\mu}$)	334
$\Delta_{=\forall}$	334
$\Delta_{=;<:\forall}$	334
$\Delta_{=\exists}$	334
$\Delta_{=;<:\exists}$	335

B Systems

Syntax of the $\text{Ob}_{1<}$ calculus	337
Syntax of the $F_{<:\mu}$ calculus	339
Syntax of the ζOb calculus	342

C Proofs

Scoping for (incomplete) environments	347
---------------------------------------	-----

List of Notations

The main notations in this book are listed in the order in which they are introduced. Each chapter may reuse notation introduced in previous chapters; this notation is not always repeated, unless its meaning changes considerably.

2 Class-Based Languages

class	generator of objects	11
var	field attribute or variable.....	11
method	method of an object	11
self	object self-reference	11
new	new object from a class	12
procedure	procedure.....	13
<i>InstanceOf</i>	type of the instances of a class	12
subclass	subclass	15
super	reference to a superclass during override.....	15
override	used to replace a class attribute in a subclass ..	15
<:	subtype relation	18
typecase	discriminating on a type	19
<i>AxB</i>	product type	20
<i>(a,b)</i>	pair	20
<i>fst</i>	first (i.e., left) element of a pair.....	20
<i>snd</i>	second (i.e., right) element of a pair	20
<i>A→B</i>	function type	21
<i>A*B</i>	mutable product type	21
Self	the type of self	24

3 Advanced Class-Based Features

ObjectType	object type	26
<i>ObjectTypeOf</i>	type of an object	26
ObjectOperator	object type operator	29
type	type	29
<:	suboperator relation.....	32

4 Object-Based Languages

object	object	35
clone	imperative shallow copy of an object	36
<i>DescendentOf</i>	relation between an object and its clones	37
embed	inclusion of a method from another object.....	40

copied from
extends
override
child of
delegate
reparent

shorthand for embed	40
embedding relation between objects	41
used to replace an attribute in a derived object	41
delegation relation between objects	43
indirection to a method of another object	44
dynamic change in the child of relation	47

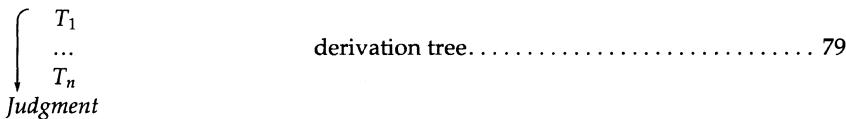
6 Untyped Calculi

=
 \equiv
 \triangleq
 $::=$
 $\Phi_i^{i \in 1..n}$
 ζ
 a, b, c
 x, y, z
 $b\{x \leftarrow c\}$
 l
 $[l_i = \zeta(x_i)b_i \text{ } i \in 1..n]$
 $\zeta(x)b$
 $a.l$
 $a.l \Leftarrow \zeta(y)b$
 $FV(a)$
 \gg
 \rightarrow
 $\rightarrow\rightarrow$
 $b\{x\}$
 $b\{c\}$
 $[..., l=b, ...]$
 $a.l := b$
 $\vdash \mathcal{S}$
 $\vdash a \leftrightarrow b$
 v
 $\vdash a \rightsquigarrow v$
 $\lambda(x)b$
 $b(a)$
 $\langle \rangle$
 fix
 $\mu(x)b$
 $\langle l_i = a_i \text{ } i \in 1..n \rangle$
 $a.l$
 $a.l := b$

equal (informal)	66
syntactically identical	59
equal by definition	59
syntax definition	60
sequence Φ_1, \dots, Φ_n	58
self binder	57
terms	60
term variables	60
substitution: c replaces the free x 's in term b	59
label	60
object	60
method	60
field selection or method invocation	60
method update	60
free variables of a	61
top-level one-step reduction	61
one-step reduction	61
many-step reduction	61
x may occur free in b	61
$b\{x \leftarrow c\}$, when x is clear from context	61
field	58
field update	58
judgment	62
equivalence judgment	63
result	64
weak-reduction judgment	64
function abstraction	66
function application	66
syntactic translation	66
fixpoint operator	68
recursive term	69
record	76
record field selection	76
record field update	76

7 First-Order Calculi

\emptyset	empty typing environment.....	79
$E \vdash \mathfrak{S}$	judgment	79
(Rule name) (Annotations)		
$\frac{E_1 \vdash \mathfrak{S}_1 \quad \dots \quad E_n \vdash \mathfrak{S}_n}{E \vdash \mathfrak{S}}$	rule	79



$E \vdash \diamond$	environment judgment	79
A, B, C	types	80
$[l_i:B_i]^{i \in 1..n}$	object type	80
$E \vdash A$	type judgment	81
$E \vdash a : A$	typing judgment	81
$[l_i=\zeta(x_i:A_i)b_i]^{i \in 1..n}$	object (with typing)	81
$a.l \Leftarrow \zeta(x:A)b$	method update (with typing)	81
$[..., l, m : B, ...]$	set of variables defined in environment E	82
$dom(E)$	the first-order typed ζ -calculus	83
\mathbf{Ob}_1	the first-order typed λ -calculus	83
\mathbf{F}_1	the first-order typed $\lambda\zeta$ -calculus	83
\mathbf{FOb}_1	boolean type	83
$Bool$	natural number type	83
Nat	integer number type	83
Int	real number type	83
$Real$	$x : A \triangleq a$ and $E \vdash a : A$ for appropriate E	84
$x : A \triangleq a$	equivalence judgment	89
$E \vdash b \leftrightarrow c : A$	fixpoint operator at type A	92
fix_A	recursive term (with typing).	92
$\mu(x:A)b$		

8 Subtyping

Top	the biggest type.....	93
$E \vdash A <: B$	subtyping judgment.....	93
$\mathbf{Ob}_{1<:}$	first-order ζ -calculus with subtyping	94
$\mathbf{F}_{1<:}$	first-order λ -calculus with subtyping	94
$\mathbf{FOb}_{1<:}$	first-order $\lambda\zeta$ -calculus with subtyping	94
$E \vdash a \leftrightarrow a' : A$	typed equivalence judgment	99
$Class(A)$	class type for object type A	100

$\text{Class}(A)_{\text{Ins}, \text{Sub}}$
 $(l_i; B_i \mid_{i \in 1..n})$
 $a.l_i = (y; B_j) \zeta(x; A)b$
 $[l_i; v_i; B_i \mid_{i \in 1..n}]$
 $+$
 $-$
 \circ
 $E \vdash v B <: v' B'$

class type with visibility restrictions	102
record type	106
method update with elder	108
object type with variance annotations v_i	110
covariant (read-only) variance annotation	110
contravariant (write-only) variance annotation	110
invariant (read-write) variance annotation	110
subtyping judgment with variance annotation	110

9 Recursion

$B[X \leftarrow C]$
 $b[X \leftarrow C]$
 $B(X)$
 $b(X)$
 $B\{C\}$
 $b\{C\}$
 $\mu(X)B$
 $\text{fold}(A, a)$
 $\text{unfold}(a)$
 $\mathbf{Ob}_{1\mu}$
 $\mathbf{F}_{1\mu}$
 $\mathbf{FOb}_{1\mu}$
 $\mathbf{Ob}_{1<\mu}$
 $\mathbf{F}_{1<\mu}$
 $\mathbf{FOb}_{1<\mu}$
 $A + B$
 inl_{AB}
 inr_{AB}
 if_{ABC}
 Unit
 unit
 $\text{typecase } a \mid (x; A)d_1 \mid d_2$

substitution: C replaces the free X 's in type B	113
substitution: C replaces the free X 's in term b	113
X may occur free in type B	113
X may occur free in term b	113
$B[X \leftarrow C]$, when X is clear from context	113
$b[X \leftarrow C]$, when X is clear from context	113
recursive type	113
isomorphism into a recursive type	113
isomorphism out of a recursive type	113
first-order ζ -calculus with recursion	114
first-order λ -calculus with recursion	114
first-order $\lambda\zeta$ -calculus with recursion	114
$\mathbf{Ob}_{1\mu}$ with subtyping	117
$\mathbf{F}_{1\mu}$ with subtyping	117
$\mathbf{FOb}_{1\mu}$ with subtyping	117
sum (disjoint union) type	122
left injection into a sum type	122
right injection into a sum type	122
discrimination on elements of a sum type	122
unit type	122
the element of Unit	122
typecase construct	126

10 Untyped Imperative Calculi

$\text{imp}\zeta$
 $\text{clone}(a)$
 $\text{let } x=a \text{ in } b$
 $a;b$
 $\text{imp}\zeta_f$
 $\text{imp}\lambda$
 $\text{imp}\lambda\zeta_f$
 \emptyset
 $\rho\{y \leftarrow a\}$

the imperative untyped ζ -calculus	129
cloning (shallow copy)	129
local definition of an immutable identifier	129
sequential evaluation	130
the imperative ζ -calculus with fields	130
the imperative λ -calculus	131
the imperative $\lambda\zeta$ -calculus with fields	132
empty translation environment	132
translation environment extension	132

<i>var</i> $x=a$ <i>in b</i>	local definition of a mutable identifier	132
$x:=a$	assignment to a mutable identifier	131
<i>let</i> $x=a;$	top-level definition	133
ι	store location	136
$v ::= [l_i := l_i^{i \in 1..n}]$	object result	136
$\sigma ::= \iota \mapsto (\zeta(x_i)b_i, S_i)^{i \in 1..n}$	store	136
$\sigma.\iota \leftarrow m$	store update	136
$S ::= x_i \mapsto v_i^{i \in 1..n}$	stack	136
$\sigma \vdash \diamond$	well-formed store judgment	136
$\sigma.S \vdash \diamond$	well-formed stack judgment	136
$\sigma.S \vdash a \rightsquigarrow v.\sigma'$	term reduction judgment	136
$\text{dom}(\sigma)$	set of location defined in a store	136

11 First-Order Imperative Calculi

$M ::= [l_i:B_i^{i \in 1..n}] \Rightarrow B_j$	method type	147
$\Sigma ::= \iota \mapsto M_i^{i \in 1..n}$	store type	147
$\Sigma(\iota)$	method type for location ι	147
$\Sigma_1(\iota)$	self type of method type at ι	147
$\Sigma_2(\iota)$	result type of method type at ι	147
$\vdash M \in \text{Meth}$	well-formed method type judgment	147
$\Sigma \models \diamond$	well-formed store type judgment	147
$\Sigma \models v : A$	result typing judgment	147
$\Sigma \models S : E$	stack typing judgment	147
$\Sigma \models \sigma$	store typing judgment	147
$\Sigma' \succcurlyeq \Sigma$	Σ' is an extension of Σ	148

12 A First-Order Language

Top	the biggest type	154
Object (X) $[l_i:v_i:B_i^{i \in 1..n}]$	object type	154
Class (A)	class type	154
object ($x:A$) $l_i := b_i^{i \in 1..n}$ end	object	155
$a.l := \text{method}(x:A) b$ end	method update	155
new c	new object from a class	155
root	root class	155
subclass of $c:C$ with ($x:A$) $l_i = b_i^{i \in n+1..n+m}$	subclass	155
override $l_i = b_i^{i \in Ovr \subseteq 1..n}$		
end		
$c^l(a)$	class selection	155
typecase a	typecase	155
when ($x:A$) b_1 else b_2 end		
fun ($x:A$) b end	function	156

13 Second-Order Calculi

$\forall(X)B$	universally quantified type.....	169
$\lambda(X)b$	type abstraction	169
$b(A)$	type application	169
$\forall(X <: A)B$	bounded universally quantified type	171
$\lambda(X <: A)b$	bounded type abstraction	171
$\exists(X <: A)B$	bounded existentially quantified type	173
$\langle A', b \rangle$	element of an existential type (informal)	173
$pack X <: A = A' with b : B$	element of an existential type.....	173
$open c as X <: A, x : B$ in $d : D$	using an element of an existential type.....	173
F	second-order λ -calculus without \exists	170
Ob	second-order ζ -calculus without \exists	170
FOb	second-order $\lambda\zeta$ -calculus without \exists	170
F_μ	F with recursion	170
Ob_μ	Ob with recursion.....	170
FOb_μ	FOb with recursion	170
$F_<$	F with subtyping	172
$Ob_<$	Ob with subtyping	172
$FOb_<$	FOb with subtyping	172
$F_{<\mu}$	F with recursion and subtyping.....	172
$Ob_{<\mu}$	Ob with recursion and subtyping	172
$FOb_{<\mu}$	FOb with recursion and subtyping	172
F	second-order λ -calculus.....	175
Ob	second-order ζ -calculus.....	175
FOb	second-order $\lambda\zeta$ -calculus.....	175
F_μ	F with recursion	175
Ob_μ	Ob with recursion	175
FOb_μ	FOb with recursion	175
$F_<$	F with subtyping	175
$Ob_<$	Ob with subtyping	175
$FOb_<$	FOb with subtyping	175
$F_{<\mu}$	F with recursion and subtyping.....	175
$Ob_{<\mu}$	Ob with recursion and subtyping	175
$FOb_{<\mu}$	FOb with recursion and subtyping	175
$A(X^+)$	X occurs positively in A	177
$A(X^-)$	X occurs negatively in A	177
$A(X^0)$	X occurs neither positively nor negatively in A . 177	
$A(X^+, Y^-)$	both $A[X^+]$ and $A[Y^-]$	178
$A \rightarrow^{\forall\exists} B$	encoding of variant function types	178
$A x^{\exists\exists} B$	encoding of variant product types.....	179
$\zeta(X)B$	Self-quantified type	180
$\langle C, c \rangle$	element of Self-quantified type (informal).....	180
$(\zeta(X)B(X))(C)$	unfolding $B\{C\}$ of a Self-quantified type	180

<i>wrap</i> ($Y <: A = C$) <i>b</i>	181
<i>use c as</i> $Y <: A$, $y : B$ <i>in d : D</i>	181
<i>wrap</i> (A, c)	182
<i>wrap</i> ($X = A$) <i>c</i>	182

14 A Semantics

{...}	185
\cup	188
\cap	188
$-$	187
D	185
\sqsubseteq	186
\perp	186
\sqcup	188
\sqcap	188
*	185
\rightarrow	185
D_\perp	186
L	185
m	185
L_i	186
W	186
+	186
\sqsubseteq	198
\times	187
D_i	186
e	186
r	186
p	186
;	186
\uparrow	186
$\langle m_1=x_1, \dots, m_n=x_n \rangle$	186
$f l \leftarrow x$	186
(x, y)	186
$x P y$	188
$\langle x_i \rangle$	186
CUPER	187
$Univ$	187
$C(P)$	188
$distance(R, T)$	188
$\mu(S)F(S)$	188
Gen	189
\preccurlyeq	189
set	185
set union	188
set intersection	188
set difference	187
the domain (cpo) of values	185
partial order on a domain	186
least element of a domain	186
join	186
meet	188
the error value	185
continuous function space	185
lifting	186
the countable set of labels	185
label	185
the finite subset $\{m_0, \dots, m_i\}$ of L	186
error domain, $\{\perp, *\}$	186
coalesced sum	186
coalesced sum membership	198
cartesian product	187
approximant of D	186
embedding	186
retraction	186
projection	186
function composition	186
function restriction	186
element of $L \rightarrow D$	187
function extension	187
pair	186
$(x, y) \in P$	187
sequence	186
complete uniform pers	187
the largest cuper	187
least cuper containing P	188
distance between two cupers	188
unique fixpoint	188
the generators on cupers	189
extension relation on generators	189

$[(m_i; T_i)^{i \in I}]$	
P°	semantic object types 189
$T(P)$	restriction of the relation P to finite elements 191
$U(P)$	transitive closure of P 191
NUSR	completion of P 191
NUPER	nonempty uniform symmetric binary relations 191
TV	nonempty uniform pers 191
TE	set of type variables 196
η	set of type expressions 196
$\llbracket A \rrbracket_{\mathbb{M}}$	type environment 197
ρ	semantics of a type 197
$\llbracket a \rrbracket_{\rho}$	value environment 197
	semantics of a term 197

15 Definable Covariant Self Types

$\zeta(X)[l_i; B_i X^+]^{i \in 1..n}$	ζ -object type 201
$a_A.l_j$	ζ -object selection 202
$a.l \Leftarrow (Y <: A, y:A(Y))\zeta(x:A(Y))b$	ζ -object update 202
$Class(A)$	class type (for ζ -objects) 210

16 Primitive Covariant Self Types

$Obj(X)[l_i; B_i X^+]^{i \in 1..n}$	structural object type 216
$obj(X=C)[l_i = \zeta(x_i; X)b_i X^+]^{i \in 1..n}$	structural object 216
$a.l$	structural method invocation 216
$a.l \Leftarrow (Y <: A, y:Y)\zeta(x:Y)b$	structural method update 216
$C(D)$	unfolding of structural object type 216
$a(D)$	unfolding of structural object 216
S	calculus with Self types and structural rules 222
$Obj(X)[l_i; v_i; B_i X^+]^{i \in 1..n}$	structural object type with variance 223
$B\{X^+\}$	positive occurrence 223
$B\{X^-\}$	negative occurrence 223
S_V	S with universally quantified types 228
$Class(A)$	class type (for primitive Self types) 237
$Class(A)_{Ins, Sub}$	class type with visibility restrictions 239

17 Imperative Calculi with Self Types

$a.l \Leftarrow (y, z=c)\zeta(x)b$	method update (generalized) 241
$\lambda()b$	dummy type abstraction 243
$a()$	dummy type application 243
imp ζ_V	the imperative ζ -calculus with $\lambda()b$ and $a()$ 244
$(\lambda()b, S)$	type abstraction result 244
$[l_i; B_i X^+]^{i \in 1..n}$	object type 245
$a.l \Leftarrow \zeta(x)b$	method update 245
<i>let</i> $x=a$ <i>in</i> b	local definition 245

$[l_i = b_i]_{i \in 1..n}$	object with fields	245
$a.l := b$	field update	245
$a.l \Leftarrow (y)\zeta(x)b$	method update	245
$T ::= X_i \rightarrow A_i$	type stack	247
$\{\leftarrow T\}$	type stack substitution	247
$M ::= Obj(X)[l_i; B_i]_{i \in 1..n} \Rightarrow j$	method type	248
$\Sigma ::= t_i \mapsto M_i$	store type	248
$\Sigma_1(t)$	self type of method type at t	248
$\Sigma_2(A, t)$	result type of method type at t	248
$\vdash M \in Meth$	well-formed method type judgment	248
$\Sigma \vdash \diamond$	well-formed store type judgment	248
$\Sigma \vdash v : A$	result typing judgment	248
$\Sigma \vdash S \cdot T : E$	stack typing judgment	248
$\Sigma \vdash \sigma$	store typing judgment	248
$dom(S)$	set of variables defined in stack S	248

18 Interpretations of Object Calculi

$[f_k: B_k]_{k \in 1..m} \mid l_i: B_i]_{i \in 1..n}]$	object type with m fields and n methods	258
$[f_k = b_k]_{k \in 1..m} \mid l_i = \zeta(x_i; A) b_i]_{i \in 1..n}]$	object with m fields and n methods	258
$o_A.f$	field selection	258
$o_A.f := b$	field update	258
$o_A.l$	method invocation	258
$Obj(X)[f_k: B_k]_{k \in 1..m} \mid l_i: B_i[X^+]]_{i \in 1..n}]$	object type with Self, m fields and n methods ..	259
$obj(X=A)[f_k = b_k]_{k \in 1..m} \mid l_i = \zeta(x_i; X) b_i]_{i \in 1..n}]$	object with Self, m fields and n methods	259
$nil(C)$	uninitialized value of type C	267

19 A Second-Order Language

$All(X <: A)B$	bounded universal type	274
$object(x: X = A) l_i = b_i]_{i \in 1..n} \text{ end}$	object	275
$a.l := \text{method}(x: X <: A) b \text{ end}$	method update	275
$\text{subclass of } c: C \text{ with}(x: X <: A)$	subclass	275
$l_i = b_i]_{i \in 1..n+m}$		
$\text{override } l_i = b_i]_{i \in Ovr \subseteq 1..n} \text{ end}$		
$c \wedge l(A, a)$	class selection	275
$\text{fun}(X <: A) b \text{ end}$	type abstraction	275
$b(A)$	type application	275

20 A Higher-Order Calculus

$Ob_{\omega <: \mu}$	higher-order ζ -calculus	287
Ty	the kind of all types	287
$K \Rightarrow L$	the kind of the operators from kind K to kind L	287

$A :: K$	constructor A has kind K	287
$X <: A :: K$	X subconstructor of A at kind K	287
$\forall(X <: A :: K)B$	universally quantified type	288
$\lambda(X <: A :: K)b$	constructor abstraction	288
$b(A)$	constructor application	288
$\lambda(X :: K)B$	operator	288
$\lambda(X)B$	$\lambda(X :: Ty)B$	288
$B(A)$	operator application	288
$E \vdash K \text{ kind}$	well-formed kind judgment	290
$E \vdash A :: K$	constructor kind judgment	290
$E \vdash A \leftrightarrow B :: K$	constructor equivalence judgment	290
$E \vdash A <: B :: K$	constructor inclusion judgment	290
$[K]$	maximum constructor of a kind	290
$X :: K$	stands for $X <: [K] :: K$ (in environments)	290
$X <: A$	stands for $X <: A :: Ty$ (in environments)	290
X	stands for $X <: Top :: Ty$ (in environments)	290
$E \vdash A$	stands for $E \vdash A :: Ty$	290
$E \vdash A \leftrightarrow B$	stands for $E \vdash A \leftrightarrow B :: Ty$	290
$E \vdash A <: B$	stands for $E \vdash A <: B :: Ty$	290
Op	stands for $Ty \Rightarrow Ty$	295
$A <: B$	stands for $A <: B :: Op$	295
A^*	fixpoint of the operator A	295
$lub_E(A)$	least upper bound of a constructor	298
A^n	normal form of a constructor	298
$E \vdash^n A <: B :: K$	normal subconstructor judgment	299
$E \vdash^n \nu A <: \nu' B$	normal subconstructor judgment with variance	299

21 A Language with Matching

object ($x:A$) $l_i=b_i$ $i \in 1..n$ end	object	306
$a.l :=$ method ($x:A$) b end	method update	306
subclass of $c:C$ with ($x:X <: \# A$) sub class	subclass	306
$l_i=b_i$ $i \in n+1..n+m$		
override $l_i=b_i$ $i \in Ovr \subseteq 1..n$ end		
fun ($X <: \# A$) b end	match-bound type abstraction	306
$E \vdash A :: Obj$	object type judgment	311
$E \vdash A <: \# B$	matching judgment	311

List of Languages

The following is a list of the languages mentioned in the text, with some bibliographic references. There are many other object-oriented languages that we do not discuss explicitly.

Act1	[78, 80]	Oaklisp	[76]
Beta	[86]	Oberon	[107]
C++	[115]	Obliq	[41]
Cecil	[49]	Omega	[26, 27]
CLOS	[96]	PolyTOIL	[37]
Eiffel	[88, 89]	Rapide	[75]
Ellie	[21]	Sather	[116]
Emerald	[105]	School	[109]
Garnet	[94]	Self	[12, 121, 122]
Java	[23]	Simula	[25, 57]
Kevo	[117]	Smalltalk	[66]
Modula-3	[95]	Theta	[58]
NewtonScript	[22]	Trellis/Owl	[110]

Bibliography

- [1] Abadi, M. 1994. *Baby Modula-3 and a theory of objects*. Journal of Functional Programming 4(2); 249-283.
- [2] Abadi, M. and Cardelli, L. 1994. *A semantics of object types*. In Proceedings of the Ninth Annual IEEE Symposium on Logic in Computer Science, 332-341.
- [3] Abadi, M. and Cardelli, L. 1994. *A theory of primitive objects: Second-order systems*. In Proceedings of the European Symposium on Programming. Lecture Notes in Computer Science 788, 1-25. Springer-Verlag.
- [4] Abadi, M. and Cardelli, L. 1995. *On subtyping and matching*. In Proceedings of the European Conference on Object-Oriented Programming. Lecture Notes in Computer Science 952, 145-167. Springer-Verlag.
- [5] Abadi, M. and Cardelli, L. 1995. *A theory of primitive objects: Second-order systems*. Science of Computer Programming 25(2-3); 81-116.
- [6] Abadi, M. and Cardelli, L. 1995. *An imperative object calculus*. Theory and Practice of Object Systems 1(3); 151-166.
- [7] Abadi, M. and Cardelli, L. 1996. *A theory of primitive objects: Untyped and first-order systems*. Information and Computation 125(2); 78-102.
- [8] Abadi, M., Cardelli, L., Pierce, B., and Plotkin, G. D. 1991. *Dynamic typing in a statically typed language*. ACM Transactions on Programming Languages and Systems 13(2); 237-268.
- [9] Abadi, M., Cardelli, L., and Viswanathan, R. 1996. *An interpretation of objects and object types*. In Proceedings of the 23rd Annual ACM Symposium on Principles of Programming Languages, 396-409.
- [10] Abadi, M. and Plotkin, G. 1990. *A per model of polymorphism and recursive types*. In Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science, 355-365.
- [11] Adams, N. and Rees, J. 1988. *Object-oriented programming in Scheme*. In Proceedings of the 1988 ACM Conference on Lisp and Functional Programming, 277-288.
- [12] Agesen, O., Bak, L., Chambers, C., Chang, B.-W., Hölzle, U., Maloney, J., Smith, R. B., Ungar, D., and Wolczko, M. 1993. *The Self 3.0 programmer's reference manual*. Sun Microsystems.
- [13] Alagic, S., Sunderraman, R., and Bagai, R. 1994. *Declarative object-oriented programming: Inheritance, subtyping, and prototyping*. In Proceedings of the European

- Conference on Object-Oriented Programming. Lecture Notes in Computer Science 821, 236-259. Springer-Verlag.
- [14] Albano, A., Bergamini, R., Ghelli, G., and Orsini, R. 1993. *An object data model with roles*. In Proceedings of the 19th International Conference on Very Large Data Bases, 39-51.
 - [15] Albano, A., Cardelli, L., and Orsini, R. 1985. *Galileo: A strongly typed, interactive conceptual language*. ACM Transactions on Database Systems 10(2); 230-260.
 - [16] Albano, A., Ghelli, G., and Orsini, R. 1995. *Fibonacci: A programming language for object databases*. The Very Large Data Bases Journal 4(3); 403-444.
 - [17] Amadio, R. M. 1991. *Recursion over realizability structures*. Information and Computation 91(1); 55-85.
 - [18] Amadio, R. M. and Cardelli, L. 1993. *Subtyping recursive types*. ACM Transactions on Programming Languages and Systems 15(4); 575-631.
 - [19] America, P. 1989. *A behavioural approach to subtyping in object-oriented programming languages*. Philips Research Journal 44(2-3); 365-383.
 - [20] America, P. and van der Linden, F. 1990. *A parallel object-oriented language with inheritance and subtyping*. In Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications, and of the European Conference on Object-Oriented Programming, 161-168.
 - [21] Andersen, B. 1992. *Ellie: A general, fine-grained, first-class, object-based language*. Journal of Object-Oriented Programming 5(2); 35-42.
 - [22] Apple 1993. *The NewtonScript programming language*. Apple Computer, Inc.
 - [23] Arnold, K. and Gosling, J. 1996. *The Java™ programming language*. Addison-Wesley.
 - [24] Barendregt, H. P. 1985. *The lambda calculus: Its syntax and semantics*. North-Holland.
 - [25] Birtwistle, G. M., Dahl, O.-J., Myhrhaug, B., and Nygaard, K. 1979. *Simula Begin*. Studentlitteratur.
 - [26] Blaschek, G. 1991. *Type-safe OOP with prototypes: The concepts of Omega*. Structured Programming 12(12); 1-9.
 - [27] Blaschek, G. 1994. *Object-oriented programming with prototypes*. Springer-Verlag.
 - [28] Böhm, C. and Berarducci, A. 1985. *Automatic synthesis of typed λ -programs on term algebras*. Theoretical Computer Science 39(2-3); 135-154.
 - [29] Booch, G. 1986. *Object-oriented development*. IEEE Transactions on Software Engineering 12(2); 211-221.
 - [30] Booch, G. 1994. *Object-oriented analysis and design with applications*. Benjamin / Cummings.

- [31] Borning, A. H. 1981. *The programming language aspects of ThingLab, a constraint-oriented simulation laboratory*. ACM Transactions on Programming Languages and Systems 3(4); 353-387.
- [32] Borning, A. H. 1986. *Classes versus prototypes in object-oriented languages*. In Proceedings of the ACM/IEEE Fall Joint Computer Conference, 36-40.
- [33] Bracha, G. and Cook, W. 1990. *Mixin-based inheritance*. In Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications, and of the European Conference on Object-Oriented Programming, 303-311.
- [34] Browne, R. 1994. *Eiffel: Frequently asked questions*. <comp.lang.eiffel> newsgroup.
- [35] Bruce, K. B. 1994. *A paradigmatic object-oriented programming language: Design, static typing and semantics*. Journal of Functional Programming 4(2); 127-206.
- [36] Bruce, K. B., Cardelli, L., Castagna, G., The Hopkins Objects Group, Leavens, G. T., and Pierce, B. 1995. *On binary methods*. Theory and Practice of Object Systems 1(3); 217-238.
- [37] Bruce, K. B., Schuett, A., and van Gent, R. 1995. *PolyTOIL: A type-safe polymorphic object-oriented language*. Williams College Technical Report.
- [38] Canning, P., Cook, W., Hill, W., Olthoff, W., and Mitchell, J. C. 1989. *F-bounded polymorphism for object-oriented programming*. In Proceedings of ACM Conference on Functional Programming and Computer Architecture, 273-280.
- [39] Cardelli, L. 1988. *A semantics of multiple inheritance*. Information and Computation 76(2-3); 138-164.
- [40] Cardelli, L. 1994. *Extensible records in a pure calculus of subtyping*. In *Theoretical Aspects of Object-Oriented Programming*, C. A. Gunter and J. C. Mitchell, eds., 373-425. MIT Press.
- [41] Cardelli, L. 1995. *A language with distributed scope*. Computing Systems 8(1); 27-59.
- [42] Cardelli, L. and Longo, G. 1991. *A semantic basis for Quest*. Journal of Functional Programming 1(4); 417-458.
- [43] Cardelli, L. and Mitchell, J. C. 1991. *Operations on records*. Mathematical Structures in Computer Science 1(1); 3-48.
- [44] Cardelli, L., Mitchell, J. C., Martini, S., and Scedrov, A. 1994. *An extension of system F with subtyping*. Information and Computation 109(1-2); 4-56.
- [45] Cardelli, L. and Wegner, P. 1985. *On understanding types, data abstraction and polymorphism*. Computing Surveys 17(4); 471-522.
- [46] Cardone, F. 1989. *Relational semantics for recursive types and bounded quantification*. In Proceedings of Automata, Languages and Programming. Lecture Notes in Computer Science 372, 164-178. Springer-Verlag.

- [47] Cardone, F. 1990. *Tipi ricorsivi e inheritance in linguaggi funzionali*. Ph.D. Thesis, Dipartimento di Informatica, Università di Torino.
- [48] Castagna, G. 1995. *Covariance and contravariance: Conflict without a cause*. ACM Transactions on Programming Languages and Systems 17(3); 431-447.
- [49] Chambers, C. 1993. *The Cecil language specification and rationale*. Technical Report 93-03-05. University of Washington, Dept. of Computer Science and Engineering.
- [50] Chambers, C., Ungar, D., and Lee, E. 1989. *An efficient implementation of Self, a dynamically-typed object-oriented language based on prototypes*. In Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications. ACM SIGPLAN Notices 24(10); 49-70.
- [51] Compagnoni, A. B. 1995. *Higher-order subtyping with intersection types*. Ph.D. Thesis, Cip-Data Koninklijke Bibliotheek, Den Haag, Nijmegen.
- [52] Cook, W., Hill, W., and Canning, P. 1990. *Inheritance is not subtyping*. In Proceedings of the Seventeenth Annual ACM Symposium on Principles of Programming Languages, 125-135.
- [53] Cook, W. and Palsberg, J. 1994. *A denotational semantics of inheritance and its correctness*. Information and Computation 114(2); 329-350.
- [54] Cook, W. R. 1989. *A denotational semantics of inheritance*. Ph.D. Thesis, Computer Science Dept., Brown University.
- [55] Cook, W. R. 1989. *A proposal for making Eiffel type-safe*. In Proceedings of the European Conference of Object-Oriented Programming, 57-72.
- [56] Curien, P.-L. and Ghelli, G. 1992. *Coherence of subsumption, minimum typing and type-checking in F_{\leq}* . Mathematical Structures in Computer Science 2(1); 55-91.
- [57] Dahl, O. and Nygaard, K. 1966. *Simula, an Algol-based simulation language*. Communications of the ACM 9(9); 671-678.
- [58] Day, M., Gruber, R., Liskov, B., and Myers, A. C. 1995. *Subtypes vs. where clauses: Constraining parametric polymorphism*. In Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications, 156-168.
- [59] Dony, C., Malenfant, J., and Cointe, P. 1992. *Prototype-based languages: From a new taxonomy to constructive proposals and their validation*. In Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications, 201-217.
- [60] Eifrig, J., Smith, S., Trifonov, V., and Zwarico, A. 1995. *An interpretation of typed OOP in a language with state*. Lisp and Symbolic Computation 8(4); 1-41.
- [61] Fisher, K., Honsell, F., and Mitchell, J. C. 1994. *A lambda calculus of objects and method specialization*. Nordic Journal of Computing 1, 3-37.

- [62] Girard, J.-Y. 1971. *Une extension de l'interprétation de Gödel à l'analyse, et son application à l'élimination des coupures dans l'analyse et la théorie des types*. In Proceedings of the Second Scandinavian Logic Symposium, 63-92. North-Holland.
- [63] Girard, J.-Y., Lafont, Y., and Taylor, P. 1989. *Proofs and types*. Cambridge University Press.
- [64] Gonthier, G., Lévy, J.-J., and Melliès, P.-A. 1992. *An abstract standardisation theorem*. In Proceedings of the Seventh Annual IEEE Symposium on Logic in Computer Science, 72-81.
- [65] Gordon, A. and Rees, G. 1996. *Bisimilarity for a first-order calculus of objects with subtyping*. In Proceedings of the 23rd Annual ACM Symposium on Principles of Programming Languages, 386-395.
- [66] Goldberg, A. and Robson, D. 1983. *Smalltalk-80. The language and its implementation*. Addison-Wesley.
- [67] Gunter, C. A. 1992. *Semantics of programming languages: Structures and techniques*. MIT Press.
- [68] Gunter, C. A. and Mitchell, J. C., eds. 1994. *Theoretical aspects of object-oriented programming*. MIT Press.
- [69] Harper, R. 1994. *A simplified account of polymorphic references*. Information Processing Letters 51(4); 201-206.
- [70] Harper, R. and Pierce, B. 1991. *A record calculus based on symmetric concatenation*. In Proceedings of the 18th Annual ACM Symposium on Principles of Programming Languages, 131-142.
- [71] Hofmann, M. and Pierce, B. C. 1995. *Positive subtyping*. In Proceedings of the 22nd Annual ACM Symposium on Principles of Programming Languages, 186-197.
- [72] Hofmann, M. and Pierce, B. C. 1995. *A unifying type-theoretic framework for objects*. Journal of Functional Programming 5(4); 593-635.
- [73] Kamin, S. N. 1988. *Inheritance in Smalltalk-80: A denotational definition*. In Proceedings of the 15th Annual ACM Symposium on Principles of Programming Languages, 80-87.
- [74] Kamin, S. N. and Reddy, U. S. 1994. *Two semantic models of object-oriented languages*. In *Theoretical Aspects of Object-Oriented Programming*, C. A. Gunter and J. C. Mitchell, eds., 463-495. MIT Press.
- [75] Katiyar, D., Luckham, D., and Mitchell, J. C. 1994. *Polymorphism and subtyping in interfaces*. ACM SIGPLAN Notices 29(8); 22-34.
- [76] Lang, K. J. and Pearlmuter, B. A. 1986. *Oaklisp: An object-oriented scheme with first-class types*. In Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications, 30-37.

- [77] Leroy, X. 1992. *Polymorphic typing of an algorithmic language*. Ph.D Thesis, Rapport de Recherche 1778. INRIA.
- [78] Lieberman, H. 1981. *A preview of Act1*. AI Memo 625. MIT.
- [79] Lieberman, H. 1986. *Using prototypical objects to implement shared behavior in object-oriented systems*. In Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications, 214-223. ACM Press.
- [80] Lieberman, H. 1987. *Concurrent object-oriented programming in Act 1*. In *Object-oriented concurrent programming*, A. Yonezawa and M. Tokoro, eds., 9-36. MIT Press.
- [81] Liskov, B. H. and Guttag, J. 1986. *Abstraction and specification in program development*. MIT Press.
- [82] MacQueen, D. B., Plotkin, G. D., and Sethi, R. 1986. *An ideal model for recursive polymorphic types*. Information and Control 71(1-2); 95-130.
- [83] Madsen, O. L., Magnusson, B., and Møller-Pedersen, B. 1990. *Strong typing of object-oriented languages revisited*. In Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications, and of the European Conference on Object-Oriented Programming, 141-160.
- [84] Madsen, O. L. and Møller-Pedersen, B. 1988. *What object-oriented programming may be, and what it does not have to be*. In Proceedings of the European Conference on Object-Oriented Programming. Lecture Notes in Computer Science 276, 1-20. Springer-Verlag.
- [85] Madsen, O. L. and Møller-Pedersen, B. 1989. *Virtual classes, a powerful mechanism in object-oriented programming*. In Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications, 397-406.
- [86] Madsen, O. L., Møller-Pedersen, B., and Nygaard, K. 1993. *Object-oriented programming in the Beta programming language*. Addison-Wesley.
- [87] Mellies, P.-A. February 1996. Personal communication.
- [88] Meyer, B. 1988. *Object-oriented software construction*. Prentice Hall.
- [89] Meyer, B. 1992. *Eiffel: The language*. Prentice Hall.
- [90] Milner, R. 1978. *A theory of type polymorphism in programming*. Journal of Computer and System Sciences 17(3); 348-375.
- [91] Milner, R., Tofte, M., and Harper, R. 1989. *The definition of Standard ML*. MIT Press.
- [92] Mitchell, J. C. 1990. *Toward a typed foundation for method specialization and inheritance*. In Proceedings of the 17th Annual ACM Symposium on Principles of Programming Languages, 109-124.
- [93] Mitchell, J. C. and Plotkin, G. D. 1988. *Abstract types have existential type*. ACM Transactions on Programming Languages and Systems 10(3); 470-502.

- [94] Myers, B. A., Giuse, D. A., and Vander Zanden, B. 1992. *Declarative programming in a prototype-instance system: Object-oriented programming without writing methods*. In Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications, 184-200.
- [95] Nelson, G. ed. 1991. *Systems programming with Modula-3*. Prentice Hall.
- [96] Paepcke, A. ed. 1993. *Object-oriented programming: The CLOS perspective*. MIT Press.
- [97] Palme, J. 1973. *Protected program modules in Simula 67*. Modern DataTeknik 12, 8.
- [98] Palsberg, J. 1995. *Efficient inference of object types*. Information and Computation 123(2); 198-209.
- [99] Palsberg, J. and Schwartzbach, M. I. 1994. *Static typing for object-oriented programming*. Science of Computer Programming 23(1); 19-53.
- [100] Palsberg, J. and Schwartzbach, M. I. 1994. *Object-oriented type systems*. John Wiley & Sons.
- [101] Parnas, D. L. 1972. *On the criteria to be used in decomposing systems into modules*. Communications of the ACM 15(12); 1053-1058.
- [102] Pierce, B. C. and Steffen, M. *Higher-order subtyping*. Theoretical Computer Science (to appear).
- [103] Pierce, B. C. and Turner, D. N. 1994. *Simple type-theoretic foundations for object-oriented programming*. Journal of Functional Programming 4(2); 207-247.
- [104] Plotkin, G. D., Abadi, M., and Cardelli, L. 1994. *Subtyping and parametricity*. In Proceedings of the Ninth Annual IEEE Symposium on Logic in Computer Science, 310-319.
- [105] Rajendra, K. R., Tempero, E., Levy, H. M., Black, A. P., Hutchinson, N. C., and Jul, E. 1991. *Emerald: A general-purpose programming language*. Software Practice and Experience 21(1); 91-118.
- [106] Reddy, U. S. 1988. *Objects as closures: Abstract semantics of object-oriented languages*. In Proceedings of the 1988 ACM Conference on Lisp and Functional Programming, 289-297.
- [107] Reiser, M. and Wirth, N. 1992. *Programming in Oberon: Steps beyond Pascal and Modula*. Addison-Wesley.
- [108] Reynolds, J. C. 1983. *Types, abstraction, and parametric polymorphism*. In *Information Processing*, R. E. A. Mason, ed., 513-523. North Holland.
- [109] Rodriguez, N., Jerusalimschy, R., and Rangel, J. L. 1993. *Types in School*. ACM SIGPLAN Notices 28(8); 81-89.
- [110] Schaffert, C., Cooper, T., Bullis, B., Kilian, M., and Wilpolt, C. 1986. *An introduction to Trellis/Owl*. In Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications, 9-16.

- [111] SIS 1976. *Data processing - Programming languages - Simula Swedish standard SS 636114*. ISBN 91-7162-234-9 Stockholm, Sweden.
- [112] Smyth, M. B. and Plotkin, G. D. 1982. *The category-theoretic solution of recursive domain equations*. SIAM Journal of Computing 11(4); 761-783.
- [113] Snyder, A. 1987. *Inheritance and the development of encapsulated software systems*. In *Research directions in object-oriented programming*, 165-188. MIT Press.
- [114] Stein, L. A., Lieberman, H., and Ungar, D. 1988. *A shared view of sharing: The treaty of Orlando*. In *Object-oriented concepts, applications, and databases*, W. Kim and F. Loehovsky, eds., 31-48. Addison-Wesley.
- [115] Stroustrup, B. 1991. *The C++ programming language*. 2nd ed. Addison-Wesley.
- [116] Szypersky, C., Omohundro, S., and Murer, S. 1993. *Engineering a programming language: The type and class system of Sather*. TR-93-064. ICSI, Berkeley.
- [117] Taivalsaari, A. 1992. *Kevo, a prototype-based object-oriented language based on concatenation and module operations*. Report LACIR 92-02. University of Victoria.
- [118] Taivalsaari, A. 1993. *A critical view of inheritance and reusability in object-oriented programming*. Jyväskylä studies in computer science, economics and statistics 23, University of Jyväskylä.
- [119] Taivalsaari, A. 1993. *Object-oriented programming with modes*. Journal of Object-Oriented Programming 6(3); 25-32.
- [120] Tofte, M. 1990. *Type inference for polymorphic references*. Information and Computation 89(1); 1-34.
- [121] Ungar, D., Chambers, C., Chang, B.-W., and Hözle, U. 1991. *Organizing programs without classes*. Lisp and Symbolic Computation 4(3); 223-242.
- [122] Ungar, D. and Smith, R. B. 1991. *Self: The power of simplicity*. Lisp and Symbolic Computation 4(3); 187-205.
- [123] Viswanathan, R. February 1996. Personal communication.
- [124] Wadsworth, C. 1980. *Some unusual λ -calculus numeral systems*. In *To H. B. Curry: Essays on combinatory logic, lambda calculus and formalism*, J. P. Seldin and J. R. Hindley, eds., 215-230. Academic Press.
- [125] Wand, M. 1987. *Complete type inference for simple objects*. In Proceedings of the Second Annual IEEE Symposium on Logic in Computer Science, 37-44.
- [126] Wand, M. 1989. *Type inference for record concatenation and multiple inheritance*. In Proceedings of the Fourth Annual IEEE Symposium on Logic in Computer Science, 92-97.
- [127] Wirth, N. 1983. *Programming in Modula-2*. Springer-Verlag.
- [128] Wright, A. K. and Felleisen, M. 1994. *A syntactic approach to type soundness*. Information and Computation 115(1); 38-94.

Index

A

abstract

- ~ class 11, 101
- ~ type 173
- bounded ~ type
 - see partially ~ type*
- partially ~ type 29, 173

abstraction

- bounded type ~ 171
- constructor ~ 288
- operator ~ 288
- type ~ 169, 243

application

- constructor ~ 288
- operator ~ 288
- type ~ 169, 243

assumption (of rule)

see premise judgment

attribute (of object)

B

binary method

- ~ in O-1/2/3 157, 274, 308

bounded

- ~ abstract type
 - see partially abstract type*
- ~ existential quantifier 173
- ~ polymorphism 171
- ~ type abstraction 171
- ~ type parameterization 29
- ~ type variable 171
- ~ universal quantifier 171, 228, 243, 288
- ~ universal quantifier in O-2/3 274, 306

C

call-by-

- ~ keyword 67

~name

~value

call-next-method

class

- 11
 - ~ encoding 73, 100, 124, 144, 209, 214, 237, 295
 - ~ in O-1/2/3 153, 274, 307
 - ~ selection 155, 275, 307
 - ~ type in O-1/2/3 153, 274, 306
- abstract ~ 101
- concrete ~ 102
- leaf ~ 102
- root ~ 155, 276, 307

class-based

cloning

- 36, 130
 - interpretation of ~ 260, 263, 271

closed term

closure

coalesced sum

complete metric space (of cupers)

complete uniform per

conclusion (of rule)

concrete class

congruence rule

constructor

- 288
 - ~ abstraction 288
 - ~ application 288

context

contractiveness

contravariance

- 21, 28, 94, 116, 178, 231
 - ~ annotation 110, 223

- ~ of function types 21, 94, 112, 178

- ~ of universal quantifiers 171

covariance

- 20, 94, 116, 178, 231
 - ~ annotation 110, 223

- ~ of existential quantifiers 174

- ~ of object types 109

- ~ of product types 20, 111, 179

- ~ of record types 106, 112, 179
- ~ requirement for Self types 202, 223
- cuper (complete uniform per) 187
- cyclic-record interpretation 262, 267
- D**
 - default parameters 67
 - delegation 13, 39, 42
 - dynamic ~ 46
 - static ~ 46
 - delegation-based 42
 - denotational semantics 185
 - derivation 63, 79
 - disjoint union type 122
 - dispatch
 - dynamic ~ 18
 - multiple ~ 53
 - single ~ 53
 - static ~ 18
 - distance (for cupers) 188
 - dynamic
 - ~ dispatch 18
 - ~ inheritance 46
 - ~ typing 126
 - see also typecase*
- E**
 - elder 108
 - embedding 13, 39
 - embedding-based 40
 - embedding-retraction pair 186
 - environment (for judgments) 79
 - environment (semantic) 197
 - type ~ 197
 - equational theory 84
 - ~ and self 90
 - ~ for constructors 290
 - ~ for existential quantifiers 174
 - ~ for records 106
 - ~ for recursive types 114
 - ~ for Self quantifiers 181
 - ~ for structural objects 217
 - ~ for subtyping 98
 - ~ for universal quantifiers 171
 - ~ for ζ -objects 203
- F**
 - F-bounded parameterization 33
 - field 12, 58
 - ~ selection 52, 58, 130
 - ~ update 52, 58, 130, 245
 - imperative ~s 130, 245
 - finiteness 187, 188
 - fixpoint
 - ~ of a function on cupers 188
 - ~ of an operator 295
 - ~ operator 68, 92
 - formal system 79
 - fragment 79
 - equational ~ 84
 - free variable 61, 118, 176
 - fresh method 239
 - function 66
 - ~ encoding 66, 91, 112, 115, 131, 178
 - function type 21, 83
 - ~ encoding 91, 112, 115, 178
 - contravariance of ~s 21, 94, 112, 178
 - subtyping of ~s 21, 94
 - G**
 - grammar notation 60
 - ground type 82
 - H**
 - higher-order
 - ~ bounded parameterization 33
 - ~ subtyping 295
 - see also subconstructor*
 - host object 9, 38, 57
 - I**
 - implementation
 - ~ of abstract type 173
 - ~ of inheritance 14

inclusion 287
 inheritance 9, 38
 ~ in O-1/2/3 153, 273, 305
 ~ of binary methods 30, 297, 305
 ~-is-not-subtyping 30
 dynamic ~ 46
 encoding of ~ 74, 100, 209, 238, 295
 encoding of multiple ~ 75, 101
 explicit ~ 39
 implementation of ~ 14
 implicit ~ 39
 mixin ~ 39
 multiple ~ 14, 16, 44, 48
 single ~ 16
 single ~ in O-1/2/3 153, 273, 305
 static ~ 46
 inner 16
 inspect 20
 instance interface 102, 239
 interface
 ~ of abstract type 173
 instance ~ 102
 subclass ~ 102, 239
 interpretation
 ~ of objects 257
 cyclic-record ~ 262, 267
 recursive-record ~ 78, 260, 265
 self-application ~ 76, 185, 259, 264,
 271
 split-method ~ 263, 268
 state-application ~ 261, 266
 invariance 21, 94, 116
 ~ annotation 110, 223
 ~ of object types 94
 ~ of recursive types 116
 invariant
 object shape ~ 215
 recoup ~ 214
 structural ~ 216
 structural subtyping ~ 215
 invocation 13, 18, 39, 42, 52, 58, 59, 129

J

judgment 62, 79
 valid ~ 80

K
 kind 287

L
 leaf class 102
 let 130, 245
 lifting 186
 location 136, 146, 242

M
 match-bound
 ~ quantified type 306
 ~ type abstraction 306
 ~ type application 306
 matching 33, 305
 method 8, 57, 153, 274, 307
 ~ extraction 14, 93, 106
 ~ invocation 13, 18, 39, 42, 52, 58, 59,
 129
 ~ lookup 13
 ~ override 9, 74
 ~ override in O-1/2/3 153, 273, 307
 ~ specialization 22, 28, 239, 273, 282
 ~ suite 13
 ~ type 146, 247
 ~ update 14, 37, 52, 58, 59, 129, 210,
 224, 241
 ~ update in O-1/2/3 153, 275, 307
 binary ~ 31, 208, 294
 binary ~ in O-1/2/3 157, 274, 308
 fresh ~ 239
 imperative ~ update 129, 241
 private ~ 102
 protected ~ 102
 public ~ 102
 recoup ~ 212
 virtual ~ 15

metric space
 complete ~ (of cupers) 188

minimum types 95, 118

mode-switching 46, 49

multiple
 ~ dispatch 53
 ~ inheritance 14, 16, 44, 48
 ~ subtyping 27

- encoding of ~ inheritance 75, 101
- N**
- narrow 20
 - negative occurrence 177, 223
 - new 12
 - ~ in O-1/2/3 155, 275, 307
 - see also class*
 - nonexpansiveness 188, 194, 197
- O**
- object 7, 57
 - ~ extension 38, 60
 - ~ in O-1/2/3 153, 273, 307
 - ~ protocol 25
 - ~ shape invariant 215
 - ~ without class 35
 - ~~orientation 7
 - donor ~ 38
 - host ~ 9, 38, 57
 - prototype ~ 47
 - structural ~ 216
 - trait ~ 47
 - $\zeta\sim$ 201
 - object type 26, 35, 80, 189, 288
 - ~ in O-1/2/3 153, 273, 305
 - ~ with Self 201, 222
 - covariance of ~s 109
 - invariance of ~s 94
 - structural ~ 216
 - subtyping of ~s 94
 - $\zeta\sim$ 201
 - object-based 1, 35, 155, 273, 305
 - occurrence
 - negative ~ 177, 223
 - positive ~ 177, 223
 - operational semantics 64, 86, 120, 127, 135, 224, 228, 242, 244, 289
 - operator
 - ~ abstraction 288
 - ~ application 288
 - fixpoint of an ~ 295
 - type ~ 287
 - override 9, 74
 - ~ in O-1/2/3 153, 273, 307
- P**
- parametric in Self 125, 211, 237, 274, 281
 - parent link 42
 - partial equivalence relation 187
 - partially abstract type 29, 173
 - per (partial equivalence relation) 187
 - polymorphism 169, 274
 - bounded ~ 171
 - subtype ~ 17
 - positive occurrence 177, 223
 - pre-method 23, 73, 100, 144, 209, 214, 237, 297
 - premise judgment 62, 79
 - primitive semantics 58
 - private method 102
 - product type 20
 - ~ encoding 111
 - covariance of ~s 20, 111, 179
 - projection 186
 - protected method 102
 - protocol 32
 - prototype 36, 47
 - public method 102
- Q**
- qua 16, 20
 - quantifier
 - bounded existential ~ 173
 - bounded universal ~ 171, 228, 243, 288
 - bounded universal ~ in O-2/3 274, 306
 - existential ~ 173
 - Self ~ 169, 179, 201, 269
 - universal ~ 170
- R**
- rank 187, 188
 - record 76, 106, 259
 - ~ encoding 112
 - record type 106, 179, 264
 - ~ encoding 112, 179
 - covariance of ~s 106, 112, 179
 - subtyping of ~s 106

- recoup 212
 - ~ invariant 214
- recursive type 113
 - invariance of ~s 116
- recursive-record interpretation 78, 260, 265
- reduction 61
 - weak ~ 64
- repARENT 47
- representation type 173, 202
- resend 16
- result 64, 86, 120, 136, 242, 244, 289
 - ~ typing 146, 248
- root class 155, 276, 307
- rule 79
 - ~ naming 79
 - congruence ~ 62
 - structural ~ 172, 190, 216, 221, 243, 293
- S**
- scoping 61, 118, 176
- selection 52
- selection (of field) 52, 58, 130
- Self
 - ~ quantifier 169, 179, 201, 269
 - ~ type 24, 30, 125, 169, 201, 221, 242, 294
 - ~ type in O-2/3 273, 305
 - ~ type specialization 24
 - see also method specialization*
 - parametric in ~ 125, 211, 237, 274, 281
- self 9, 57
- self-application
 - ~ interpretation 76, 185, 259, 264, 271
 - ~ semantics 185
- self-unfolding 202
- semantics
 - denotational ~ 185
 - operational ~ 64, 86, 120, 127, 135, 224, 228, 242, 244, 289
 - self-application ~ 185
- shallow copy
- single
 - see cloning*
- specialization
 - method ~ 22, 28, 239, 273, 282
 - Self type ~ 24
- split-method interpretation 263, 268
- stack 136, 242
 - ~ type 247
 - ~ typing 147, 248
- state-application interpretation 261, 266
- static dispatch 18
- store 136, 242
 - ~ typing 146, 248
- structural
 - ~ invariants 216
 - ~ object 216
 - ~ object type 216
 - ~ rule 172, 190, 216, 221, 243, 293
 - ~ subtyping 172, 190
 - ~ subtyping invariant 215
 - ~ update 172, 190, 222, 243, 293
- stuck 87, 138, 152
- subclass 15, 27, 30
 - ~ in O-1/2/3 155, 273, 307
 - ~ interface 102, 239
 - see also class, inheritance*
- subconstructor 287, 290
 - see also higher-order subtyping*
- subject reduction 86, 97, 121, 127, 146, 229, 247, 301
- suboperator 295
- subprotocol 32
- substitution 59, 61, 113, 120
- subsumption 18, 93
 - ~ in O-1/2/3 154, 273, 305
- subtype polymorphism 17
- subtyping 18, 27, 30, 93
 - ~ by type name 27
 - ~ by type structure 27
 - ~ for existential quantifiers 174

- ~ for function types 21, 94
 - ~ for object types 94
 - ~ for object types with primitive Self and variance 226
 - ~ for object types with Self 204
 - ~ for record types 106
 - ~ for Self quantifiers 180
 - ~ for universal quantifiers 171
 - ~ for variance annotations 110
 - ~ in O-1/2 153, 273
 - higher-order ~ 295
 - see also subconstructor*
 - multiple ~ 27
 - structural ~ 172, 190
 - sum, coalesced 186
 - super 15
 - ~ encoding 74, 101
 - ~ in O-1/2/3 156, 276, 307
 - syntax notation 60
- T**
- trait 47, 144
 - ~ encoding 73, 101
 - see also class encoding*
 - true type 18
 - type
 - ~ abstraction 169, 243
 - ~ application 169, 243
 - ~ operator 287
 - ~ parameter 28
 - ~ variable 113, 171
 - abstract ~ 173
 - bounded ~ parameterization 29
 - bounded ~ variable 171
 - bounded abstract ~
 - see partially abstract ~*
 - disjoint union ~ 122
 - dynamic ~ 126
 - see also typecase*
 - environment (semantic) 197
 - function ~ 21, 83
 - method ~ 146, 247
 - minimum ~s 95, 118
 - object ~ 26, 35, 80, 189, 288
- object ~ in O-1/2/3 153, 273, 305
 - object ~ with Self 201, 222
 - partially abstract ~ 29, 173
 - product ~ 20
 - record ~ 106, 179, 264
 - recursive ~ 113
 - representation ~ 173, 202
 - Self ~ 24, 30, 125, 169, 201, 221, 242, 294
 - Self ~ in O-2/3 273, 305
 - stack ~ 247
 - store ~ 146, 248
 - unique ~s 85
 - ζ -object ~ 201
 - typecase 19, 23, 126
 - ~ in O-1/2/3 156, 274, 311
 - typing environment 79
- U**
- unfolding 202
 - union type, disjoint 122
 - unique types 85
 - unit 122
 - unit type 122
 - universal
 - ~ quantifier 170
 - bounded ~ quantifier 171, 228, 243, 288
 - bounded ~ quantifier in O-2/3 274, 306
 - update
 - field ~ 52, 58, 130, 245
 - method ~ 14, 37, 52, 58, 59, 129, 210, 224, 241
 - method ~ in O-1/2/3 153, 275, 307
- V**
- valid judgment 63, 80
 - variance
 - see co~, contra~, in~*
 - variance annotations 110, 124, 222
 - ~ in O-1/2/3 154, 274, 307
 - virtual method 15