To my mother and the memory of my late father
# Contents

*List of Figures*     xi

*Preface*     xv

I Type Problems in Object-Oriented Languages     1

1 *Introduction*     3
  1.1 Type systems in programming languages     4
  1.2 Type checking and strongly typed languages     6
  1.3 Focus on statically typed class-based languages     12
  1.4 Foundations: A look ahead     13

2 *Fundamental Concepts of Object-Oriented Languages*     17
  2.1 Objects, classes, and object types     17
  2.2 Subclasses and inheritance     22
  2.3 Subtypes     24
  2.4 Covariant and contravariant changes in types     26
  2.5 Overloading versus overriding methods     27
  2.6 Summary     32

3 *Type Problems in Object-Oriented Languages*     33
  3.1 Type checking object-oriented languages is difficult     33
  3.2 Simple type systems are lacking in flexibility     35
  3.3 Summary of typing problems     48

4 *Adding Expressiveness to Object-Oriented Languages*     49
4.1 GJ 49
4.2 Even more flexible typing with Eiffel 60
4.3 Summary 69

5 Understanding Subtypes 71
5.1 Subtyping for non-object types 72
5.2 Object types 83
5.3 Subtyping for class types 84
5.4 Summary 86

6 Type Restrictions on Subclasses 89
6.1 Allowable changes to method types 89
6.2 Instance variable types invariant in subclasses 91
6.3 Changing visibility 92
6.4 Summary 93

7 Varieties of Object-Oriented Programming Languages 95
7.1 Multi-methods vs. object-based vs. class-based languages 95
7.2 Well-known object-oriented languages 103
7.3 Summary 111

Historical Notes and References for Section I 113

II Foundations:
The Lambda Calculus 117

8 Formal Language Descriptions and the Lambda Calculus 119
8.1 The simply-typed lambda calculus 120
8.2 Adding pairs, sums, records, and references 132
8.3 Summary 140

9 The Polymorphic Lambda Calculus 141
9.1 Parameterized types and polymorphism 141
9.2 Recursive expressions and types 147
9.3 Information hiding and existential types 151
9.4 Adding subtypes to the polymorphic lambda calculus 156
9.5 Summary 165

Historical Notes and References for Section II 167
### Contents

**III  Formal Descriptions of Object-Oriented Languages**  
171

10 *SOOL, a Simple Object-Oriented Language*  
173
10.1 Informal description and example  
173
10.2 Syntax and type-checking rules  
176
10.3 Summary  
200

11 *A Simple Translational Semantics of Objects and Classes*  
201
11.1 Representing objects at runtime  
201
11.2 Modeling *SOOL* types in \( \Lambda^p_\mathcal{C} \)  
203
11.3 Modeling *SOOL* expressions in \( \Lambda^p_\mathcal{C} \)  
207
11.4 Modeling classes — first try  
212
11.5 Problems with modeling subclasses  
218
11.6 Summary  
223

12 *Improved Semantics for Classes*  
225
12.1 (Re-)Defining classes  
225
12.2 A correct subclass encoding  
232
12.3 Summary and a look ahead  
233

13 *SOOL's Type System Is Safe (and Sound)*  
239
13.1 The translation of *SOOL* to \( \Lambda^p_\mathcal{C} \) is sound  
239
13.2 The translation is well defined  
255
13.3 *SOOL* is type safe  
258
13.4 Errors  
260
13.5 Summary  
262

14 *Completing SOOL: super, nil, Information Hiding, and Multiple Inheritance*  
263
14.1 Using methods from superclasses  
263
14.2 Translating *nil*  
266
14.3 A complication with *self*  
271
14.4 Finer control over information hiding  
272
14.5 Multiple inheritance  
275
14.6 Summary  
279

*Historical Notes and References for Section III*  
283
# Contents

## IV Extending Simple Object-Oriented Languages 289

15 Adding Bounded Polymorphism to $SOOL$ 291
   15.1 Introducing $PSOOL$ 291
   15.2 Translational semantics of $PSOOL$ 296
   15.3 Summary 297

16 Adding $MyType$ to Object-Oriented Programming Languages 299
   16.1 Typing self with $MyType$ 300
   16.2 $MOOL$: Adding $MyType$ to $SOOL$ 309
   16.3 Translational semantics of $MOOL$ 319
   16.4 Soundness of translation for $MOOL$ 322
   16.5 Summary 330

17 Match-Bounded Polymorphism 331
   17.1 Benefits of match-bounded polymorphism 331
   17.2 Introducing $PMOOL$ 333
   17.3 Examples and comparison with F-bounded polymorphism 334
   17.4 Translational semantics of $PMOOL$ 335
   17.5 Soundness of the translation of $PMOOL$ 342
   17.6 Summary 347

18 Simplifying: Dropping Subtyping for Matching 349
   18.1 Can we drop subtyping? 349
   18.2 Introducing hash types 352
   18.3 Type-checking rules 356
   18.4 An informal semantics of hash types 360
   18.5 Summary 361

Historical Notes and References for Section IV 363

Bibliography 367

Index 379
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>C1rCellClass defined as a subclass of CellClass.</td>
<td>23</td>
</tr>
<tr>
<td>2.2</td>
<td>Covariant and contravariant changes in types.</td>
<td>26</td>
</tr>
<tr>
<td>2.3</td>
<td>Classes with overridden and overloaded method equals.</td>
<td>29</td>
</tr>
<tr>
<td>3.1</td>
<td>Typing deepClone methods in subclasses.</td>
<td>38</td>
</tr>
<tr>
<td>3.2</td>
<td>Node class.</td>
<td>42</td>
</tr>
<tr>
<td>3.3</td>
<td>Doubly linked node class — with errors.</td>
<td>43</td>
</tr>
<tr>
<td>3.4</td>
<td>Legal doubly linked node class — with cast.</td>
<td>45</td>
</tr>
<tr>
<td>3.5</td>
<td>Example showing why IndDoubleNodeType cannot be a subtype of NodeType.</td>
<td>46</td>
</tr>
<tr>
<td>3.6</td>
<td>Circle and color circle classes.</td>
<td>47</td>
</tr>
<tr>
<td>4.1</td>
<td>Point interfaces in Java.</td>
<td>52</td>
</tr>
<tr>
<td>4.2</td>
<td>Point class in Java.</td>
<td>53</td>
</tr>
<tr>
<td>4.3</td>
<td>RATIONAL class in Eiffel, part 1.</td>
<td>62</td>
</tr>
<tr>
<td>4.4</td>
<td>RATIONAL class in Eiffel, part 2.</td>
<td>63</td>
</tr>
<tr>
<td>4.5</td>
<td>Eiffel classes LINKABLE and BILINKABLE, part 1.</td>
<td>65</td>
</tr>
<tr>
<td>4.6</td>
<td>Eiffel classes LINKABLE and BILINKABLE, part 2.</td>
<td>66</td>
</tr>
<tr>
<td>4.7</td>
<td>Eiffel version of COMPARING.</td>
<td>69</td>
</tr>
<tr>
<td>5.1</td>
<td>A record ( r: [m: S; n: T; p: U] ), and another record ( r': [m: S'; n: T'; p: U'; q: V'] ) masquerading as an element of type ( [m: S; n: T; p: U] ).</td>
<td>74</td>
</tr>
<tr>
<td>5.2</td>
<td>A function ( f: S \rightarrow T ), and another function ( f': S' \rightarrow T' ) masquerading as having type ( S \rightarrow T ).</td>
<td>75</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>--------</td>
<td>-----------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>5.3</td>
<td>A variable (x: \text{Ref } S), and another variable (x': \text{Ref } S') masquerading as having type (\text{Ref } S).</td>
<td>78</td>
</tr>
<tr>
<td>5.4</td>
<td>Summary of subtyping rules.</td>
<td>88</td>
</tr>
<tr>
<td>6.1</td>
<td>Changing types of methods in subclasses.</td>
<td>90</td>
</tr>
<tr>
<td>7.1</td>
<td>\textbf{Cell} and \textbf{StringCell} classes in Beta.</td>
<td>105</td>
</tr>
<tr>
<td>7.2</td>
<td>The \textbf{Subject-Observer} pattern expressed with virtual types.</td>
<td>106</td>
</tr>
<tr>
<td>7.3</td>
<td>Specializing \textbf{Subject-Observer} to windows.</td>
<td>107</td>
</tr>
<tr>
<td>7.4</td>
<td>Lack of least upper bounds in Java interfaces.</td>
<td>109</td>
</tr>
<tr>
<td>8.1</td>
<td>Typing rules for expressions of the typed lambda calculus.</td>
<td>126</td>
</tr>
<tr>
<td>8.2</td>
<td>Type-checking rules for (\Lambda^\rightarrow), part 1.</td>
<td>136</td>
</tr>
<tr>
<td>8.3</td>
<td>Type-checking rules for (\Lambda^\rightarrow), part 2.</td>
<td>137</td>
</tr>
<tr>
<td>8.4</td>
<td>Computation rules for (\Lambda^\rightarrow).</td>
<td>139</td>
</tr>
<tr>
<td>9.1</td>
<td>Kind checking and equivalence rules.</td>
<td>144</td>
</tr>
<tr>
<td>9.2</td>
<td>New type-checking and congruence rules in (\Lambda^P).</td>
<td>146</td>
</tr>
<tr>
<td>9.3</td>
<td>Type-checking rules for existentials in (\Lambda^P).</td>
<td>155</td>
</tr>
<tr>
<td>9.4</td>
<td>Kind checking rules for bounded polymorphic and existential types.</td>
<td>157</td>
</tr>
<tr>
<td>9.5</td>
<td>Subtyping rules for (\Lambda^P): Part 1.</td>
<td>159</td>
</tr>
<tr>
<td>9.6</td>
<td>Subtyping rules for (\Lambda^P): Part 2.</td>
<td>160</td>
</tr>
<tr>
<td>9.7</td>
<td>Type-checking rules for (\Lambda^P), part 1: revised rules.</td>
<td>164</td>
</tr>
<tr>
<td>9.8</td>
<td>Type-checking rules for (\Lambda^P), part 2: revised rules.</td>
<td>165</td>
</tr>
<tr>
<td>9.9</td>
<td>Type-checking rules for (\Lambda^P), part 3: new rules.</td>
<td>166</td>
</tr>
<tr>
<td>10.1</td>
<td>\textbf{Cell} class.</td>
<td>174</td>
</tr>
<tr>
<td>10.2</td>
<td>\textbf{CellClass} in the fully expanded language.</td>
<td>175</td>
</tr>
<tr>
<td>10.3</td>
<td>\textbf{PointExample} program in language with abbreviations.</td>
<td>182</td>
</tr>
<tr>
<td>10.4</td>
<td>\textbf{PointExample} program in language without abbreviations.</td>
<td>183</td>
</tr>
<tr>
<td>10.5</td>
<td>Parameterized \textbf{Point} class in language with abbreviations.</td>
<td>184</td>
</tr>
<tr>
<td>10.6</td>
<td>Parameterized \textbf{Point} class in language without abbreviations.</td>
<td>185</td>
</tr>
<tr>
<td>10.7</td>
<td>Type checking rules for declarations.</td>
<td>188</td>
</tr>
<tr>
<td>10.8</td>
<td>Type-checking rules for expressions of (\text{SOOL}), part 1.</td>
<td>189</td>
</tr>
<tr>
<td>10.9</td>
<td>Type-checking rules for expressions of (\text{SOOL}), part 2.</td>
<td>192</td>
</tr>
<tr>
<td>10.10</td>
<td>Type-checking rules for statements of (\text{SOOL}).</td>
<td>194</td>
</tr>
<tr>
<td>10.11</td>
<td>Subtyping rules for (\text{SOOL}).</td>
<td>195</td>
</tr>
<tr>
<td>10.12</td>
<td>Class definition from \textbf{PointExample}.</td>
<td>197</td>
</tr>
</tbody>
</table>
### Table of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>11.1</td>
<td>Translation of types of SOOL to types in $\Lambda_{\prec}$:</td>
</tr>
<tr>
<td>11.2</td>
<td>Translation of selected expressions of SOOL to expressions in $\Lambda_{\prec}$.</td>
</tr>
<tr>
<td>11.3</td>
<td>Translation of more expressions and statements of SOOL to expressions in $\Lambda_{\prec}$.</td>
</tr>
<tr>
<td>11.4</td>
<td>Translation of class and new expressions of SOOL to expressions in $\Lambda_{\prec}$.</td>
</tr>
<tr>
<td>12.1</td>
<td>Revised type-checking rules for classes and subclasses.</td>
</tr>
<tr>
<td>12.2</td>
<td>Revised type-checking rules for messages and instance variables.</td>
</tr>
<tr>
<td>12.3</td>
<td>New translation of classes and class types.</td>
</tr>
<tr>
<td>12.4</td>
<td>New translation of new expressions.</td>
</tr>
<tr>
<td>12.5</td>
<td>New translation of subclasses.</td>
</tr>
<tr>
<td>12.6</td>
<td>Final translation of types of SOOL to types in $\Lambda_{\prec}$.</td>
</tr>
<tr>
<td>12.7</td>
<td>Translation of selected expressions of SOOL to expressions in $\Lambda_{\prec}$.</td>
</tr>
<tr>
<td>12.8</td>
<td>Final translation of expressions of SOOL to $\Lambda_{\prec}$. Part 2.</td>
</tr>
<tr>
<td>12.9</td>
<td>Translation of statements of SOOL to expressions in $\Lambda_{\prec}$.</td>
</tr>
<tr>
<td>13.1</td>
<td>CellExample program.</td>
</tr>
<tr>
<td>14.1</td>
<td>Type-checking rules for subclasses with super.</td>
</tr>
<tr>
<td>14.2</td>
<td>Translation of subclass with super from SOOL to $\Lambda_{\prec}$.</td>
</tr>
<tr>
<td>14.3</td>
<td>Illustration of the use of super in a subclass.</td>
</tr>
<tr>
<td>14.4</td>
<td>Alternative 1 for supporting nil: Object types as references.</td>
</tr>
<tr>
<td>14.5</td>
<td>Alternative 3 for supporting nil: Object types as sums.</td>
</tr>
<tr>
<td>14.6</td>
<td>Type-checking for classes and objects with hidden and visible methods.</td>
</tr>
<tr>
<td>14.7</td>
<td>Translation semantics for classes and objects and their types with hidden and visible methods.</td>
</tr>
<tr>
<td>14.8</td>
<td>A difficult case for multiple inheritance.</td>
</tr>
<tr>
<td>14.9</td>
<td>Type-checking rules for subclasses with multiple inheritance.</td>
</tr>
<tr>
<td>14.10</td>
<td>Semantics of multiple inheritance in SOOL.</td>
</tr>
<tr>
<td>15.1</td>
<td>Congruence rules for $\mathcal{P}SOOL$.</td>
</tr>
<tr>
<td>15.2</td>
<td>New subtyping rules for $\mathcal{P}SOOL$.</td>
</tr>
<tr>
<td>15.3</td>
<td>Typing rules for new expressions of $\mathcal{P}SOOL$.</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
</tr>
<tr>
<td>15.4</td>
<td>Translation of type constructors and the new types of $\mathcal{PSOOL}$ to corresponding type constructors and types in $\Lambda^{P,}$.</td>
</tr>
<tr>
<td>15.5</td>
<td>Translation of selected expressions of $\mathcal{PSOOL}$ to expressions in $\Lambda^{P,}$.</td>
</tr>
<tr>
<td>16.1</td>
<td>Typing deepClone methods with MyType.</td>
</tr>
<tr>
<td>16.2</td>
<td>Typing deepClone methods with MyType, continued.</td>
</tr>
<tr>
<td>16.3</td>
<td>Node class with MyType.</td>
</tr>
<tr>
<td>16.4</td>
<td>Doubly linked node class with MyType.</td>
</tr>
<tr>
<td>16.5</td>
<td>Types of objects generated from node classes with MyType.</td>
</tr>
<tr>
<td>16.6</td>
<td>Subtyping rules for higher kinds and replacement subtyping rule for object types.</td>
</tr>
<tr>
<td>16.7</td>
<td>Matching rules for object types.</td>
</tr>
<tr>
<td>16.8</td>
<td>Rules for $\langle # _{vis} \rangle$.</td>
</tr>
<tr>
<td>16.9</td>
<td>Typing rules for classes and subclasses in $\mathcal{MOOL}$.</td>
</tr>
<tr>
<td>16.10</td>
<td>Typing rules for message sending and instance variables in $\mathcal{MOOL}$.</td>
</tr>
<tr>
<td>16.11</td>
<td>Translation of new types and constructors.</td>
</tr>
<tr>
<td>16.12</td>
<td>Translation of classes and subclasses of $\mathcal{MOOL}$ to $\Lambda^{P,}$.</td>
</tr>
<tr>
<td>17.1</td>
<td>Typing and subtyping rules for new expressions of $\mathcal{P,MOOL}$.</td>
</tr>
<tr>
<td>17.2</td>
<td>Linked list example in $\mathcal{P,MOOL}$, Part 1.</td>
</tr>
<tr>
<td>17.3</td>
<td>Linked list example in $\mathcal{P,MOOL}$, Part 2.</td>
</tr>
<tr>
<td>17.4</td>
<td>Linked list example in $\mathcal{P,MOOL}$, Part 3.</td>
</tr>
<tr>
<td>17.5</td>
<td>Translations of selected types and constructors of $\mathcal{P,MOOL}$.</td>
</tr>
<tr>
<td>17.6</td>
<td>Translation of $\mathcal{P,MOOL}$ expressions involving polymorphism.</td>
</tr>
<tr>
<td>18.1</td>
<td>Matching rules for object types in $\mathcal{N,OOL}$.</td>
</tr>
<tr>
<td>18.2</td>
<td>Typing rules for classes and subclasses in $\mathcal{N,OOL}$.</td>
</tr>
<tr>
<td>18.3</td>
<td>Typing rules for message sending and subsumption in $\mathcal{N,OOL}$.</td>
</tr>
</tbody>
</table>
Preface

I wrote this book to provide a description of the foundations of statically typed class-based object-oriented programming languages for those interested in learning about this area. An important goal is to explain how the different components of these languages interact, and how this results in the kind of type systems that are used in popular object-oriented languages. We will see that an understanding of the theoretical foundations of object-oriented languages can lead to the design of more expressive and flexible type systems that assist programmers in writing correct programs.

Programmers used to untyped or dynamically typed languages often complain about being straitjacketed by the restrictive type systems of object-oriented languages. In fact many existing statically typed object-oriented languages have very restrictive type systems that almost literally force programmers to use casts or other mechanisms to escape from the static type system. In this work we aim to meet the needs of a programmer who wants a more expressive type system. Thus another goal of this text is to promote richer type systems that reduce the need for bypassing the type checker.

Because of the semantic complexity of the features of object-oriented languages, particularly subtyping and inheritance, it is difficult to design a static type system that is simultaneously safe and flexible. To be sure that there are no holes in the type system we need to prove that the type system is safe (essentially that no type errors can occur at run time), but we cannot do that without a description of the meaning of programs. Thus this book contains careful formal descriptions of the syntax, type system, and semantics of several progressively more complex object-oriented programming languages. With these definitions, it is possible to prove type safety.

Object-oriented programming languages have been of great practical and theoretical interest, but most of the interesting developments in foundations have been accessible only to researchers in the area. Moreover, papers in the area have taken quite different approaches, as well as using different
notation and even different terminology from each other. As a result, it has been difficult for outsiders to learn the basic material in this area.

This book differs from other recent books in the foundations of object-oriented languages in several ways. First, the focus of attention is class-based object-oriented languages, rather than object-based or multi-method languages. Thus our study is very relevant to the most popular kind of object-oriented languages in use today.

Second, this book approaches the foundations from the point of view of a programmer or language designer wishing to understand the type systems of object-oriented languages and to see how to extend the type systems to increase the expressiveness of these languages. The semantics presented suggest extensions to the language and provide the foundations for verifying the safety of the type system.

Third, we base the foundation of object-oriented programming languages on the classical typed lambda calculus and its extensions rather than introducing new calculi to explain the fundamental constructs. Thus we can rely on classical results, only including a brief review of the lambda calculus to introduce readers to the notation.

This book is intended for several different audiences. My intention has been to make it accessible to students, especially advanced undergraduates and graduate students, to practitioners wishing to have a deeper understanding of the foundations of object-oriented programming languages, and to researchers who wish to understand developments in the foundations of object-oriented languages. It can be used as the main text for a course in the foundations of object-oriented programming languages or as a supplementary text for a course with a broader focus that includes object-oriented programming languages.

We have designed the first part of the book, comprising the first seven chapters, to be especially accessible to a wide variety of readers. These chapters provide a relatively non-technical introduction to key issues in the type systems of object-oriented programming languages. As such, this part may be especially appropriate for use in a general undergraduate or graduate course covering concepts of object-oriented programming languages or as the basis for self-study.

The next part, comprising Chapters 8 and 9, provides a relatively quick introduction to the simply typed lambda calculus and many of its extensions. The goal of this part is to have the reader understand how the lambda calculus can provide a formal description of programming language constructs. This part also introduces the formalism for writing the syntax and
type-checking rules for programming languages. For readers with a solid understanding of programming languages as provided for by Pierce’s text, *Type Systems for Programming Languages* [Pie02], or Mitchell’s *Foundations for Programming Languages* [Mit96], for example, these chapters will simply provide a quick review. Others will need to spend more time to understand how such a primitive language can be used as a model of important programming language concepts and to learn how to read and understand the type-checking rules. It is not necessary to understand the deep results about the lambda calculus found in more specialized texts in order to understand the use of lambda calculus in this book.

The third part of the book, comprising Chapters 10 through 14, presents the core foundational material on class-based object-oriented languages. We begin by providing a formal definition of a simple object-oriented language, S00L, and its type system. Chapters 11 and 12 explore understanding the semantics of S00L by translating terms into a very rich extension of the typed lambda calculus. With this understanding of the language, Chapter 13 presents a proof of soundness and safety of S00L. This chapter is the technically most difficult of the book. The details of the proof in the first section of that chapter may be skipped on the first reading, but the statement of the soundness and safety theorems and the other material in the chapter are important as they illustrate how a careful formal definition of a language can lead to provable safety.

The language S00L was kept very simple so that the proof of soundness could avoid as many complications as possible. The last chapter of this part discusses many of the more specialized concepts commonly found in object-oriented languages that were left out of S00L. These include references to methods from the superclass, more refined access control in classes, nil objects, and even a discussion of multiple inheritance.

The final part of this book explores extensions of the type systems of object-oriented languages suggested by our understanding of the semantics of S00L. The extensions include F-bounded polymorphism, a new type keyword, MyType, standing for the type of self, and a relation, matching, that is more general than subtyping. We will find that the addition of these features adds considerably to the expressiveness of object-oriented languages, yet we will prove that they do not compromise the type safety of the language. We end with the presentation of a language that incorporates MyType, matching, and a new form of bounded polymorphism using matching, but that no longer contains the notion of subtyping. We will see that this simpler language is still very expressive, even without subtyping.
The topics covered in this book represent an active area of research, with new papers appearing every year. There are many topics that I would have liked to have included, but could not because of a desire to keep the size of this book manageable. The best way to keep up with current research in the area is to attend or examine the proceedings of major conferences and workshops in this area. The major conferences presenting new research in the broad area of programming languages are the Principles of Programming Languages (POPL) and Programming Language Design and Implementation (PLDI) conferences. The most important conferences presenting research on object-oriented languages are the annual Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA) conference and the European Conference on Object-Oriented Programming (ECOOP). The annual Foundations of Object-Oriented Languages (FOOL) workshop provides an important, though less formal, forum for new results in the area covered by this book. Information on the FOOL workshops is available at

http://www.cs.williams.edu/~kim/FOOL/.

One of my favorite quotes, first encountered as a signature tag on e-mail, is the following:

“The difference between theory and practice is greater in practice than in theory”  Author unknown

In pursuing my own research on topics central to the issues covered in this book, I have tried to keep this quote in mind. As a result, rather than just theorizing about issues in programming language design, my students and I have implemented interpreters and compilers for languages similar to those discussed here. (For pedagogical reasons the languages described in the text are different in inessential ways from the languages we have implemented.)

The experience of implementing and using these languages has provided better insight to the strengths and limitations of the type systems discussed here. It is my hope, and indeed one of the reasons for writing this book, that the knowledge obtained by the research community in the foundations of object-oriented programming languages will eventually work its way into practical and widely used programming languages. The growing interest in the extension, GJ, of Java described in Section 4.1 provides evidence that this kind of technology transfer has already begun.

The material presented in this book is the result of the dedicated and creative work of many researchers. The Historical Notes and References sections at the end of each of the four parts of the book credit the contributions of
many of those doing research in this area. I have also benefitted greatly from personal and professional interactions from many researchers in this area.

Primary credit for helping me get started doing research in the semantics of programming languages goes to Albert Meyer, from whom I learned an enormous amount, both about semantics and about the process of doing research, while on my first leave from Williams College. A ten-year-long professional collaboration with Guiseppe Longo was extremely productive and enjoyable, while incidentally introducing me to the beauty of Italy and France. Peter Wegner deserves credit for introducing me to object-oriented programming languages and asking annoying questions that led to many interesting results. John Mitchell and Luca Cardelli provided key influences (and funding) during a visit to Palo Alto in the spring of 1991 that led to my work on the design and proofs of type safety of object-oriented programming languages.

A three-month visit to the Newton Institute of Mathematical Sciences in the fall of 1995 during the special program on Semantics of Computation provided a great opportunity to work with other researchers in the semantics of programming languages. The interaction with Benjamin Pierce and Luca Cardelli there led to our joint paper comparing different styles of semantics for object-oriented languages.

Similarly, early meetings of the workshops on the Foundations of Object-Oriented Languages (the FOOL workshops) resulted in many interesting discussions (and arguments), some of which led to the paper “On binary methods” [BCC95], a paper with 8 co-authors who at times seemed to have at least 10 different opinions on how best to approach the issues involved. I have learned more through writing these papers (in spite of the difficulty of writing conclusions!) than through almost any other activity as a researcher. Teaching a graduate programming languages course while on a visiting professorship at Princeton University allowed me to begin writing this book while trying out the material on students.

Opportunities for collaboration with my computer science honors students at Williams College and my co-authors have taught me a great deal over the years. My honors students in computer science include Robert Allen, Jon Burstein, David Chelmow, John N. (Nate) Foster, Benjamin Goldberg, Gerald Kanapathy, Leaf Petersen, Dean Pomerleau, Jon Riecke, Wendy Roy, Angela Schuett, Adam Seligman, Charles Stewart, Robert van Gent, and Joseph Vanderwaart. Aside from the researchers and students mentioned above, my co-authors in programming language research papers include Roberto Amadio, Giuseppe Castagna, Jon Crabtree, Roberto DiCosmo, Allyn Dimock, Adrian
Fiech, Gary Leavens, Robert Muller, Martin Odersky, Scott Smith, and Philip Wadler.

I owe a great debt of gratitude to the National Science Foundation, most recently through the offices of Frank Anger, for their long-standing support for my research. NSF research grants supporting the research reported here include NSF CCR-9121778, CCR-9424123, CCR-9870253, and CCR-9988210. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.

Special thanks go to those who provided comments and corrections on drafts of this manuscript. Narciso Martí-Oliet, John N. Foster, and an anonymous reviewer provided very detailed and helpful comments on a complete draft of this book. Andrew Black provided very useful and detailed comments on an early survey paper that evolved into this book. Others who provided useful comments on different portions of the book, suggested approaches, or were helpful in clearing up historical details included Martin Abadi, Luca Cardelli, Craig Chambers, Kathleen Fisher, Cheng Hu, Assaf Kfoury, John Mitchell, Benjamin Pierce, and Jack Wiledon. Thanks to my editor Bob Prior for his friendship, for his faith in this project, and for making this task less painful than it might have been. I am grateful to Christopher Manning for sharing the LaTeX macros that resulted in this book design.

I take full credit for all omissions and errors remaining in this book. Please send corrections to kim@cs.williams.edu. I will provide a web site with errata or clarifications at


and through MIT Press at

http://mitpress.mit.edu/

I give great thanks to my family for their love and support during the long years spent writing this book. Thanks to my colleagues in the Computer Science Department at Williams for their professional support and intellectual stimulation. Finally, thanks to my teachers whose guidance led me to begin this interesting journey. Special thanks are due to H. Jerome Keisler and the late Jon Barwise at the University of Wisconsin, the late Harry Mullikan and Paul Yale at Pomona College, and Shirley Frye and Mike Svaco at Scottsdale Arcadia High School.
Part I

Type Problems in Object-Oriented Languages
1 Introduction

It is often stated that object-oriented programming languages are a major improvement over older procedural style languages. If so, why are their static type systems so poor? Some of the static type systems of object-oriented languages are too restrictive, resulting in the need for a plethora of type casts, either checked (as in Java [AGH99]) or unchecked (as in C++ [ES90]). Others allow programs with type errors to be executed. In some of these languages the type errors may be caught at run time (as in the language Beta [KMMPPN87]), while in others (like current implementations of Eiffel [Mey92]) the errors may result in run-time crashes.

In this text we will explore the foundations of object-oriented programming languages. Our purpose in examining the formal underpinnings of object-oriented languages is to answer questions like the one in the previous paragraph. This study will help the reader gain deeper insight into the fundamental concepts of these languages. It will help explain why certain features are designed the way they are, as well as provide a tool to help design more expressive, yet statically type-safe, object-oriented languages.

While the first object-oriented language, Simula 67 [BDMN73], was designed and implemented in the mid-60’s, and the Smalltalk [GR83] language was first introduced in the early ’70’s, it wasn’t until the advent of C++ in the mid-’80’s that a large number of programmers and organizations began adopting object-oriented languages. Even then, many users of C++ simply used it as a “better C” with support for abstraction. However, programmers increasingly adopted pure object-oriented languages like Smalltalk, Eiffel, and, most recently, Java, while an increasing number of C++ programmers write programs in an object-oriented style.

Why has the object-oriented style become so popular? Certainly no small part has been played by the tendency of programmers to jump on the latest
“fad” language. However there is real substance behind the reasons for the increasing use of object-oriented languages. There seem to be clear advantages for the object-oriented style in organizing and reusing software components. For example, subtyping and inheritance (notions we will define more carefully later) seem to make it much easier to adapt and reuse existing software components.

However, in many ways the quality of object-oriented programming languages falls short of existing procedural and functional languages. In this text we will focus on two ways in which they fall short – the shortcomings of type systems and the deficiencies in expressiveness of existing object-oriented programming languages.

Based on our years of experience in programming (and teaching programming) in traditional procedural languages such as FORTRAN [Bac81], Pascal [Wir71], C [KR78], Modula-2 [Wir85], and Ada [US80], as well as functional languages like LISP [MAE+65], Scheme [SS75], ML [MTH90], Miranda [Tur86], and Haskell [HJW92], we are convinced that a strong type system, especially a statically type-safe system, is a very important tool in implementing reliable programs. Thus it would be highly advantageous to provide static type systems for object-oriented languages that are of the same quality as those available for traditional procedural and functional languages, yet make it easy for the programmer to express his or her algorithmic ideas in an object-oriented style.

Unfortunately, commercially available object-oriented languages fall far short of that goal. The static type systems of object-oriented languages tend to be either insecure or more inflexible than one might desire. In some cases the rigidity of the type system leads programmers to rely on type casts (sometimes checked at run time, sometimes not) in order to obtain the expressiveness desired. In other cases, the type systems are too flexible, requiring the run-time system to generate link-time or run-time checks to ensure the integrity of the computation.

1.1 Type systems in programming languages

Type systems in programming languages assign types to all values in a computation. Static type systems also assign type expressions to all expressions of the language. Operations are provided with type information that determines to which types of values they may be applied. For example, a concatenation operator may be restricted to be applied to pairs of strings. An
"integer" addition operator may be restricted to be applied only to pairs of integers. A "real" addition operator (which may be represented by the same symbol as the "integer" addition operator) may be restricted to be applied only to pairs of reals. (We treat an overloaded operator symbol or name as referring to multiple operations rather than a single operation with multiple typings.)

Programming languages include primitive data types like integers, reals, booleans, etc., and operations that apply to values of those types. These languages also provide type constructors that allow programmers to build up composite or structured data types (e.g., records or structs, arrays, sets, etc.), as well as providing operations that may construct or be applied to values of these types. In most languages, these more complex types can be named, though their structure is visible and accessible to programmers. While more operations on these types may be designed by the programmer by writing new functions or procedures, these new operations are built from the primitive operations provided by the language. However, any programmer using these structured types may take advantage of the built-in operations to access components of the data structure, by-passing the new operations provided by the type designer. Thus these new type definitions do not appear like predefined types – their structure is visible to all.

The introduction of the notion of abstract data type (ADT) [GTW78, Gut77] in the early 1970's, and its introduction in a number of programming languages (e.g., Clu [L⁺81], Modula-2, and Ada) provided programmers with a mechanism that made it possible to introduce a collection of data type and value definitions, and operations on those definitions, that behaved more like a primitive data type.

ADT's included both a specification and an implementation, which were usually provided separately. The ADT specification provided a name for the type and provided specifications, both type and behavioral, for a collection of operations on the type. The type specification for an operation includes the types of the parameters, if any, and the return type. We will refer to such a type specification as the signature of the operation. These specifications were usually packaged together, and provided sufficient information for a programmer to write programs that used the type. The ADT implementation provided a representation for the values of the type, typically as a structured data type, and the implementations of the operations, written as procedures and functions that were allowed to access the representation of the data type.

Programmers using ADT's were not allowed access to the implementation of a data type, thus making it easier to replace one implementation of
an ADT by another. This information hiding was an important feature of the use of ADT’s. Early language mechanisms that provided support for ADT’s included Clu’s clusters, Modula-2’s modules, and Ada’s packages. ML’s signatures and structures later provided similar mechanisms.

Object-oriented languages introduced the notions of classes and objects. Objects contain both state (values) and methods (operations). The main operation provided for objects is sending a message to an object. Classes provide both specification and implementation information on objects. Not only are the names and specifications of methods included in classes, but also representation information for the state and methods. Most object-oriented languages provide mechanisms for allowing the programmer to restrict access to the representation of the state or methods of objects from clients or subclasses in order to support information hiding.

Some object-oriented languages also allow programmers to provide only specification information on objects. For example, several languages allow the programmer to provide pure abstract (C++, Java) or deferred (Eiffel) classes. The programmer simply provides method names and signatures, omitting all mention of the representation of state and implementations of methods. Java’s interfaces, while they may have initially been included to provide support for some aspects of multiple inheritance, provide a clean representation for this separation of interface and implementation. Several classes with entirely different representations may implement the same interface. A procedure or function whose parameter type is given by an interface can take as actual parameters objects generated from any class that implements the interface. This promotes a notion of reusability that is essentially independent of the notions of inheritance and subtyping.

Languages like Ada, Clu, and ML allow the user to define parameterized types (e.g., Stack(T), Tree(T), etc.). These can be seen as functions that take types as parameters and return new types. These languages also typically allow the programmer to define polymorphic functions (functions that take types as parameters, but return values rather than types). There appears to be a strong correlation between the increased expressiveness of programming languages and the increasing richness of their type systems.

1.2 Type checking and strongly typed languages

Type systems for programming languages are typically designed to provide several important functions. These include:
1.2 Type checking and strongly typed languages

- **Safety:** Type checking of programs should prevent (either at compile or run time) the execution of certain illegal operations. In Chapter 13 we go into more detail on which illegal operations type systems are responsible for preventing. For now, we simply provide the examples of attempting to add a string to an integer as a type error, and dividing an integer by zero as a non-type error.

The first is a type error because that operation should never be applied to two operands, one of which is a string and the other of which is an integer. The second is not a type error because division is an operation that is normally applied to pairs of integers. However, when the operation is applied to certain combinations of values from those types, an error results. Thus, information on the types of the operands is not sufficient to determine whether the operation will be erroneous.

- **Optimization:** Type checking can provide useful information to a compiler or interpreter. This information can be used to allocate storage for values, select appropriate code to execute (e.g., for overloaded operations), and support various optimizations.

- **Documentation:** Type annotation (or, to a lesser extent, inference) provides documentation on constructs that make it easier for programmers to determine how the constructs can or should be used. Of course, the programmer should provide more than just type information as documentation, but our experience is that omission of type information significantly impacts the comprehensibility of code.

- **Abstraction:** The ability to name types, and, even more importantly, the ability to hide the implementation of types, allows (even forces) the programmer to think at a higher level of abstraction in programming. This hiding of details allows more straightforward modeling of the problem domain, while making it possible to change the implementation of a type and its operations without impacting the correctness of programs using the implementation. Of course, an important reason for changing an implementation is to improve some aspect of the behavior of the program, but correctness of the program should be dependent only on the specification of the provided operations.

Every value generated in a program is associated with a type, either explicitly or implicitly. In a *strongly typed* language, the language implementation is required to provide a type checker that ensures that no type errors will occur at run time. For example, it must check the types of operands in order to
ensure that nonsensical operations, like dividing the integer 5 by the string “hello”, are not performed. Strongly typed languages may either be dynamically or statically type checked. Dynamic type checking normally occurs during program execution, while static type checking occurs prior to program execution, typically at compile time. In other type-related checks may take place at program link time.

In a dynamically typed language like LISP or Scheme, many operations are type checked just before they are performed. Thus the code for a plus operation may have to check the type of its operands just before the addition is performed. If both operands are integers, then an integer addition is performed. If both operands are floating point numbers or one is floating point and the other is an integer, then a floating point addition is performed. However, if one operand is a string and the other is a floating point number, then execution is terminated with an error message. In some languages an exception may be raised, which may either be handled by the program before resuming normal execution or, if there is no handler or no handler can successfully execute, the program terminates.

In a statically typed language, every expression of the language is assigned a type at compile time. If the type system can ensure that the value of each expression has a type compatible with the statically assigned type of the expression, then type checking of most operations can be performed at compile time, rather than delayed to run time.

Dynamically typed programming languages can be more expressive and flexible than statically typed languages, because the type checking is postponed until run time. In general, the problem of determining statically for an arbitrary program whether a type error will occur at run time is undecidable, yet it is generally accepted that a static type system should be decidable. As a result, sound static type checkers will rule out some programs as potentially unsafe that would actually execute without a type error.

While the exclusion of safe programs would seem to be a major problem with static type checking, there are many advantages to having a statically type-checked language. These include:

- providing earlier, and usually more accurate, information on programmer errors,

1. For convenience, we will refer to static checks as occurring at compile time, even though similar checks take place before execution in interpreted as well as compiled languages.
2. We leave it as an exercise for the more sophisticated reader to show this problem can be reduced to the halting problem. Hint: Have a type error result only if a program that is input as data halts.
1.2 Type checking and strongly typed languages

- eliminating the need for run-time type checks that can slow program execution and increase program size,

- providing documentation on the interfaces of components (e.g., procedures, functions, and packages or modules), and

- providing extra information that can be used in compiler optimizations.

As a result most modern languages have static type systems.

Procedural languages like Pascal [Wir71], Clu [L+81], Modula-2 [Wir85], and Ada 83 [US 80], and functional languages like ML [HMM86] and Haskell [HJW92] have reasonably safe static typing systems. While some of these languages have a few minor holes in the type system (e.g., variant records in Pascal), ML, Haskell, Clu, and Ada provide fairly secure type systems.

Programmers used to dynamically type-checked languages may worry that the use of a static type system will disallow or restrict the use of programs that can be dynamically determined to be type safe. For example, the static type system of standard Pascal is so inflexible that it will not allow the programmer to write a single sort procedure that will work for integer arrays of different sizes, let alone for arrays of other types like reals or characters. The language C has a similarly restrictive type system, but provides specific mechanisms (type casts) to allow the programmer to bypass the static type system when it gets in the way of the programmer.

However, modern programming languages allow more flexible use of arrays as parameters and often include support for more advanced features, such as parametric polymorphism, that have increased the expressiveness of statically typed languages. Examples of statically type-safe, yet flexible, procedural and functional programming languages include Clu, Modula-2, Ada, ML, and Haskell.

Unfortunately the situation for static type checking in object-oriented languages is not as good. The following is a list of some properties of type-checking systems of some of the more popular object-oriented languages (or the object-oriented portions of hybrid languages).

- Some provide only dynamic type checks.
  
  Smalltalk

- Some are mainly statically type-safe (if no casts), but inflexible. These languages often require explicit mechanisms to escape from the type system (e.g., unsafe type casts) to overcome deficiencies of the type system.
  
  C++, Object Pascal
• Some have very flexible static type systems, but the type systems as implemented are not sound.
  
  *Eiffel*

• Some are flexible, but need run-time type checks to overcome weaknesses in static typing.
  
  *Beta, Java, Ada95*

At the boundary between static and dynamic type systems are several constructs. Here there may be differences of opinion on what features are considered to be part of static type systems and which are part of dynamic systems.

For example, we consider constructs like `typecase` statements, which make explicit tests on the run-time type of a value, to be statically type-safe as long as the execution of such statements cannot give rise to run-time type errors or system-generated exceptions. An example of the use of such a construct in the language Theta [DGLM94] is given below. Assume the identifier `x` is declared with static type `S`, and assume that `T` and `U` are subtypes of `S`.

```plaintext
typecase x
  when T(t): ... t ...
  when U(u): ... u ...
  others: ... x ...
end;
```

In this statement, if `x`’s run-time type is a subtype of `T`, the value of `x` will be denoted by `t` (which is an identifier with static type `T`), and the code following the first `when` clause will be executed. Similarly, the code in the second `when` clause will be executed (with `u` denoting the value of `x` if the run-time type is a subtype of `U`, but not of `T`). Finally, if the run-time type of `x` fails to be a subtype of any of the types listed in the `when` clauses, then the code in the `others` clause will be executed. This is type safe because each of the branches is required to type check correctly.

No run-time type errors can occur, because if `x` has a type that is not a subtype of the types specified in the `when` clauses, the code in the `others` clause will be executed, and it must be type safe for `x` having static type `S`.

Eiffel’s “reverse assignment” involves an assignment from an expression with static type `T` to a variable whose static type `S` is a subtype of `T`. We consider this to be in the same category as `typecase`.

Suppose `x` is declared to have type `S`, where `S` is a subtype of `T`, the static type of `exp`. Then the statement

```plaintext
Team-Fly®
```
1.2 Type checking and strongly typed languages

\[ x \ ?= \ exp; \]

will type check. If the run-time type of \( \text{exp} \) is a subtype of \( S \), the value of \( \text{exp} \) will be stored in the location corresponding to \( x \). However, if the run-time type of \( \text{exp} \) fails to be a subtype of \( S \), the value \text{void} is assigned to \( x \). Thus in neither case does a run-time type error or system-generated exception occur.

This reverse assignment can be understood as a very restricted form of typecase. We can code the reverse assignment above using typecase as follows:

```plaintext
typecase \text{exp}
  when S(s): x := s;
  others: x := \text{void};
end;
```

On the other hand, we treat Java’s type cast as not being statically type safe because the failure of a cast raises a run-time exception.\(^3\)

As we shall see later, type restrictions on the redefinition of methods in many object-oriented languages give rise to situations where programmers often feel the need to by-pass the static type system. Some of these type restrictions follow from the need to preserve type safety when redefined methods are used in combination with inherited methods. Other restrictions are due to the desire to have subclasses always generate subtypes. While the introduction of bounded parametric polymorphism has helped loosen some of the rigidities of these languages, programmers of statically typed object-oriented languages are more likely to feel that static type safety gets in their way than programmers in statically typed procedural or functional languages.

As a result, in choosing from existing statically typed object-oriented languages, programmers are faced with unfortunate choices for overcoming the deficiencies of the type systems. They may attempt to program around these deficiencies, use constructs that require dynamic type checking, or use languages that allow run-time type errors to occur.

We make the case in this book that it is possible to define safe statically typed object-oriented languages that are sufficiently expressive to obviate the need for either run-time type checks or ways of escaping the type system. While borderline features like typecase statements or run-time checked

---

3. If Java could somehow guarantee that an instanceof check occurred before every type cast, like typecase statements in some languages, we would consider this to be a statically type-safe operation.
reverse assignments may occasionally be necessary to handle difficult problems with heterogeneous data structures, we prefer to have type systems that allow us to program as naturally as possible, while catching all type errors.

As we shall see in the course of this text, many type problems and rigidities arise in statically typed object-oriented languages because of the conflation of type with class, and with the mismatch of the inheritance hierarchy with subtyping. Whatever the cause, there appears to be much room for improvement in moving toward a combination of better security and greater expressiveness in the type systems.

### 1.3 Focus on statically typed class-based languages

In this text we explore the foundations of object-oriented languages by paying careful attention to the design of type systems and semantics for object-oriented languages. We will focus particularly on static type systems for class-based object-oriented languages.

There are great advantages to using statically typed languages; for example in helping programmers find and fix errors more efficiently. On the other hand, the restrictions on expressiveness can lead programmers to use languages that are not statically type safe or to find ways of by-passing the type system when it gets in the way. One of the goals of research in this area has been to ameliorate these inherent conflicts by designing language constructs that are both statically type safe and provide increased expressiveness.

Our focus on class-based rather than object-based languages comes from both practical and conceptual considerations. Class-based languages rely on classes that form templates for the generation of new objects. Object-based languages allow programmers to define objects directly, and usually provide mechanisms, for example prototypes, delegation, and cloning operations, for the creation of new objects from old. Like all distinctions in computer science, there is blurring at the edges between this categorization of languages, but the distinctions provided by this categorization are useful. (See Section 7.1.1 for a more detailed description of object-based languages.)

Virtually all popular object-oriented languages (e.g., Simula 67, Smalltalk, Object Pascal, Eiffel, Objective C, C++, Ada95, and Java) are class-based. On the other hand, object-based languages (e.g., Self, Cecil, and Emerald) tend to be research languages or are used by relatively small communities. Of course this popularity is not an indication that class-based languages are necessarily better, but it does suggest that there may be more interest in achieving a
better understanding of class-based languages.

There are also conceptual reasons for preferring to analyze class-based languages. In class-based languages, classes and objects separate important concerns. Classes form extensible templates that can be used to create new objects. Objects are the fundamental components of computation, with computation taking place by sending messages to objects. The execution of methods of an object may update its state (instance variables), but no mechanism is provided to update or add methods to existing objects. In class-based languages methods in classes may be updated by using the mechanism of inheritance to create a new subclass with the updated (or added) method. In object-based languages, the methods of objects may be updated in place or (depending on the language) be updated in the creation of a new object based on the original.

In object-based languages, objects essentially play the role of both classes and objects in class-based languages. This causes complications in providing theoretical modeling of these languages, especially in providing support for method update or addition of methods in objects. At this point, it is hard to explain the technical reasons for these difficulties without going into a much more detailed discussion of the modeling of instance variables, methods, and, particularly, the modeling of \texttt{self} (written \texttt{this} in Java and C++), a keyword representing the object currently executing a method. We will discuss some of these difficulties later in Chapter 7; for now we hope the reader is satisfied with these explanations.

Not all other researchers agree with our views on this topic. For example, Abadi and Cardelli, in their very influential text, \textit{A Theory of Objects} [AC96], argue that objects are more primitive than classes, and that mechanisms other than classes are useful in generating objects with common properties. Moreover they argue that classes are superfluous because they can be defined in terms of objects. This allows them to start with a very simple object calculus and define a variety of mechanisms (including classes) for generating objects. The associated cost is that it is more complex to model their object calculus in terms of the lambda calculus or denotational semantics in such a way as to preserve subtyping. (See Chapter 7 for a comparison.)

### 1.4 Foundations: A look ahead

We will begin this text by analyzing existing object-oriented programming languages, paying special attention to their type systems and impediments.
to expressiveness. We explore why type systems for these languages include what may at first seem to be rather arbitrary restrictions, and the consequences of ignoring these restrictions. It will become clear that there are a number of constructions that programmers would like to be able to express in these languages, but that are not currently supported in many existing statically typed object-oriented languages. In some cases, relatively simple extensions to these languages can greatly enhance expressiveness while preserving type-safety (see the discussion in Chapter 4 of the extension, GJ, of Java for one example). In other cases, attempts to add expressiveness have resulted in either type insecurities or the need to add dynamic type checking (see the discussion of Eiffel in the same chapter).

In Chapters 5 and 6 we examine the definitions of two key features of object-oriented languages: subtypes and subclasses. In particular we investigate conditions that guarantee that two types are subtypes. We also look at restrictions necessary to ensure that inherited methods in subclasses remain type correct.

We end the first part of the book with a discussion of different kinds of object-oriented languages (e.g., class-based, object-based, and multi-method languages) and an examination of statically typed object-oriented languages Simula 67, Beta, Java, C++, Smalltalk, Eiffel, and Sather with reference to our model languages and type systems.

In order to support a careful analysis of the type systems and semantics of object-oriented languages, we will introduce a prototypical object-oriented language, $\text{Sool}$, with a simple type system that is similar to those of class-based object-oriented languages in common use today. After a discussion of subtypes and subclasses (especially with regard to type restrictions on overriding methods), we begin an analysis of the foundations of object-oriented languages by providing a semantics. The semantics will allow us to precisely specify the meaning of these languages, enabling a more careful examination of the rules sufficient to guarantee the type safety of various programming constructs.

There are many alternatives available for providing the semantics of object-oriented languages. A denotational semantics would provide a mathematical specification of meaning. An operational semantics would specify the meaning of programs by providing instructions for an interpreter that would execute programs using a very simple virtual machine. One might also provide an axiomatic semantics that would provide rules for reasoning about programs. While there are advantages to each of these, and in other situations we have been quite happy with the provision of an operational seman-
Our semantics provides the meaning of programming constructs by translating them to an extended typed lambda calculus. The main advantage of a typed lambda calculus is its simplicity. The core of the calculus is the representation of functions and function application; concepts that are learned quite early in mathematics courses. While the notation may initially be unfamiliar, the ideas behind the calculus should be familiar to all readers. Also rather than restricting ourselves to a stripped-down, “pure” lambda calculus, we add familiar programming constructs such as records, pairs, and references. We also extend the lambda calculus with less familiar notions, such as parametric polymorphism and existential types, that will help to model parameterized classes and information hiding.

Another advantage of providing a translational semantics based on the lambda calculus is that these calculi have been studied in great detail over the years. As a result, rather than providing very detailed and technically intricate proofs of type soundness and safety, we simply show that our translation preserves types. This will enable us to lift type soundness and safety results from the lambda calculus to our object-oriented language. While soundness and safety proofs are of interest in their own right, our goal here is to provide explanations of typing issues in object-oriented languages to a larger audience. Thus we include only the proofs we feel are most necessary in order to provide convincing evidence that our semantics are correct and that the type system is safe. As a result, we do not hesitate to base our results on systems that are intuitively (as well as provably) safe. We provide pointers to the literature for readers who are interested in complete proofs from first principles.

After the introduction to our extended lambda calculus in the second part of the book (Chapters 8 and 9), we begin the third part of the book with a careful formal definition of our prototypical language, SOOC. In Chapter 11, we begin the task of modeling the semantics of SOOC. While modeling of objects and classes will turn out to be rather straightforward, the modeling of subclasses is surprisingly tricky if we hope to preserve type safety. However, the correct modeling provides an explanation for the difficulties in type checking methods that arise if we wish to guarantee that inherited methods remain type safe in subclasses. As one might hope, our modeling of object-oriented languages will suggest the addition of new constructs to the language (e.g., MyType) as well as to help us understand the type-checking rules of object-oriented languages. This modeling leads into one of the most technical chapters of the text, Chapter 13, in which we prove that the type
system is sound by showing that our semantics preserves typing information. We finish this part of the book by adding some common features that were omitted to simplify the original presentation and proof. These include references to methods in the superclass, the handling of null references, more refined information hiding, and multiple inheritance.

In the last part of the book (Chapters 15 through 18) we add desirable features that are not yet included in many statically typed object-oriented languages. These new features include parametric polymorphism (including what is sometimes known as F-bounded polymorphism), and a MyType construct. The combination of these features allows us to overcome many of the expressiveness limitations of existing statically typed object-oriented languages. We end the book with the sketch of a language that includes the MyType construct and drops subtyping for a slightly weaker relation, called matching.

There is much more material that could be included in a text on this subject. For example, we were tempted to include operational semantics for object-oriented languages, and we would have liked to include more material on virtual types and modules. However, our primary goal is to provide in a fairly compact form a good introduction to the concerns in designing safe, yet expressive, object-oriented programming languages. We hope that the following chapters will successfully achieve this goal. After completing this text, the reader should be prepared to go to the research literature to find information on these other topics.
In this chapter we review the fundamental concepts of object-oriented languages. We assume the reader has some experience with object-oriented languages, so our main purpose here is to establish consistent terminology for the rest of the text.

The concepts of object-oriented languages discussed here include objects, classes, methods, instance variables, dynamic method invocation, subclasses and inheritance, and subtypes. Other features include mechanisms to allow the programmer to refer to the current object and to access methods of its superclass. These concepts are described briefly below. In later chapters we will go into much more detail as to their meanings. For now, we also avoid discussion of most issues involving types. We will devote a substantial amount of attention to typing issues later.

2.1 Objects, classes, and object types

*Objects* encapsulate both state and behavior. In particular, they consist of a collection of *instance variables*, representing the state of the object, and a collection of *methods*, representing the behavior that the object is capable of performing. We sometimes refer to instance variables as the *fields* of an object. The methods are routines that are capable of accessing and manipulating the values of the instance variables of the object. When a *message* is sent to an object, the corresponding method of the object is executed. (In C++, instance variables are referred to as *member fields* or variables and methods as *member functions*.)

As is the case in Java and Smalltalk, we will assume that all objects are implicitly references. This results in a *sharing semantics* for assignment. That is, if \( o \) and \( o' \) are objects of the same type, execution of the assignment state-
ment, \( o := o' \), will result in \( o \) referring to the same object as \( o' \).\(^1\) Similarly, the equality test, \( o = o' \), will be true if and only if both have the same reference (\( i.e. \), both point to the same object). Also as in Java and Smalltalk, we will assume that the language implementation is provided with a garbage collector. Thus programmers do not have to worry about disposing of objects when they are no longer needed or accessible. The value \( \text{nil} \) is used as a null reference and is considered to be an element of all object types.

Classes are extensible templates for creating objects, providing initial values for instance variables and the bodies for methods. All objects generated from the same class share the same methods, but contain separate copies of the instance variables. New objects can be created from a class by applying the \texttt{new} operator to the name of the class.

The following is an example of a class written in the notation to be used throughout this text.

```plaintext
class CellClass {
  x: Integer := 0;

  function get(): Integer is
    { return self.x }

  function set(nuVal: Integer): Void is
    { self.x := nuVal }

  function bump(): Void is
    { self := set(self := get()+1) }
}
```

The name of the class is \texttt{CellClass}. It has a single instance variable named \( x \) that holds integer values. When a new object is created by evaluating \texttt{new CellClass}, the initial value of its instance variable \( x \) will be 0.

The class contains three methods: \texttt{get}, \texttt{set}, and \texttt{bump}. The method \texttt{get} takes no parameters and returns an integer. The methods \texttt{set} and \texttt{bump} are procedures (a function that does not return a value), which is indicated by a \texttt{return} type of \texttt{Void}. The method \texttt{set} takes a single integer parameter, \( nuVal \), while \texttt{bump} takes no parameters.

\(^1\) In this text we will use ":=" for assignment and "=" for the equality operator. While this differs from the conventions for the languages C, C++, and Java, we find this notation more sensible in relation to common mathematical usage.
The keyword `self` (written `this` in C++ and Java) is used in method bodies to indicate the object currently executing the method. The “dot” notation is used with `self` to get access to instance variables of the current object. Thus in the bodies of methods `get` and `set`, `self.x` refers to the instance variable `x` of the object executing the method.

Adopting notation from Smalltalk, we use the symbol “⇐” to represent sending a message to an object. While most languages don’t bother to distinguish notationally between accessing an instance variable and sending a message, they are quite different operations, so we use different symbols. In the body of `bump`, the message sends `self ⇐ set` and `self ⇐ get` indicate that the corresponding methods in the current object should be executed.

In most object-oriented languages, it is possible to omit the prefix `self` when used in accessing instance variables or performing message sends. For example `CellClass` could be written:

```plaintext
class CellClass {
    x: Integer := 0;

    function get(): Integer is
        { return x }

    function set(nuVal: Integer): Void is
        { x := nuVal }

    function bump(): Void is
        { set(get()+1) }
}
```

In order to keep meanings as clear as possible, we will generally include the prefix `self` as in the first version of `CellClass`.

For simplicity, and as an aid to abstraction, we will use the default that instance variables of an object are not accessible from outside of that object’s methods. We will also assume that methods are by default publicly accessible from outside of the object.

Later we will introduce notation to allow an object’s methods to be hidden from other objects. Obviously we may provide access to an instance variable accessible from outside of the object by writing appropriate “get” and “set” methods that access or update the variable.
In many object-oriented languages, class names are used for three distinct purposes: as a name for the class, as a name for a constructor of the class, and as a name for the type of objects generated from the class. In order to make it easier to describe the different meanings of these three interpretations, as well as to provide better support for abstraction, we choose not to conflate these uses.

For now, we omit discussion of constructors. Instead, we will depend on classes to provide initial values of instance variables when new objects are created with the new operator. However, we do wish to distinguish classes from the types of objects.

We consider types as abstractions that represent sets of values and the operations and relations applicable to them. In order to better support abstraction, and hence later program modification, we believe that object types should not carry implementation information. Instead they should reveal only the names and types (signatures) of the messages that may be sent to them. From this point of view, classes provide too much information to users. The identity of instance variables should not be revealed to the user, nor should hidden methods. The bodies of existing methods should also not be part of the type of an object. To support this degree of abstraction, we provide a new kind of type expression to represent the types of objects.

The type of objects with public methods \( m_1 \) of type \( T_1 \), up to \( m_n \) of type \( T_n \), will be written as \( \text{ObjectType} \{ m_1 : T_1, \ldots, m_n : T_n \} \). For example, the type of objects generated by class CellClass is:

\[
\text{CellType} = \text{ObjectType} \{ \text{get: Void \rightarrow Integer;} \\
\text{set: Integer \rightarrow Void;} \\
\text{bump: Void \rightarrow Void } \}
\]

The notation \( A \rightarrow B \) denotes the type of all functions from type \( A \) to type \( B \). Thus a function of type \( A \rightarrow B \) may be applied to arguments of type \( A \) and returns values of type \( B \). We follow Java and C++ in using Void in the domain of a function to indicate a parameterless function and, as stated earlier, Void in the range is used to indicate a procedure.

Because object types do not mention instance variables, distinct classes that have the same public methods and types can generate objects with the same object types. For example, we can define

\[
\text{2. We will see later that we can create functions that provide the initial values of instance variables in classes. This will allow us to more closely model constructors.} \\
\text{3. While it is common in some programming languages to refer to method signatures as though they were distinct from types, we shall find it convenient to refer to all such notations as types.}
\]
class DimCellClass {
    z: Integer := -1;

    function get(): Integer is
    { return self.z + 1 }

    function set(nuVal: Integer): Void is
    { self.z := nuVal - 1 }

    function bump(): Void is
    { self := set(self := get() + 1) }
}

Class DimCellClass has a different instance variable than CellClass, and
the methods get and set have bodies that are distinct from those in
CellClass. However, because the public methods are the same as those
in CellType, and they have the same types as specified there, DimCell-
Class also generates objects of type CellType. While, in this case, objects
generated from CellClass and DimCellClass exhibit identical observ-
able behaviors, two classes can generate objects of the same type even if the
methods result in different behaviors.

The following is a a simple program using these two classes. Notice that
the pair // indicates the beginning of a comment, which extends to the end
of the line.

program CellExample;
  ... // Definitions of CellClass, DimCellClass,
  // and CellType omitted
var
  c: CellType := nil; // (1)
  |
  { c := new CellClass; // (2) c := set(17); // (3) set from CellClass c := bump(); // (4) writeln(c := get())); // (5) c := new DimCellClass; // (6) c := set(17); // (7) set from DimCellClass c := bump(); // (8) writeln(c := get()) // (9) }
}
The program begins with a program statement containing the name of the program. The next several lines should contain the definitions of `CellType`, `CellClass`, and `DimCellClass`, but we have omitted them here to avoid repeating code. Line (1) contains a declaration that the variable `c` is of type `CellType`.

The main body of a program, like that of a class or method, is enclosed in curly brackets. The `new` expression on line (2) creates a new object from class `CellClass`, which is assigned to variable `c`. As mentioned earlier, we use `:=` rather than `=` to represent assignment. Lines (3) and (4) send the messages `set(17)` and `bump()` to `c`, while line (5) prints the value of the cell obtained as the result of sending the message `get()` to `c`. Because `c` was generated by class `CellClass`, the actual code executed during these message sends is that of the corresponding methods in `CellClass`.

Line (6) results in `c` being reassigned a value created from class `DimCellClass`, with lines (7), (8), and (9) sending the same messages as lines (3), (4), and (5). Because the object held in `c` is now generated from `DimCellClass`, the code executed during these message sends is now that of the corresponding methods in `DimCellClass`. Executing the above code will result in the number 18 being printed out twice, despite the fact that different method bodies are executed as a result of the method invocations of `set` and `bump` in the first and second half of the listing.

Because `c` is declared to have type `CellType`, it can be sent the messages `get`, `set`, and `bump`, as they are specified in the type. Objects generated by classes `CellClass` and `DimCellClass` can both be assigned to `c` because both have public methods with the signatures required by `CellType`.

The mechanism by which the object receiving a message is responsible for knowing which method body to execute is often called dynamic method invocation. This facility provides an enormous amount of flexibility for object-oriented programming. It allows a program to send messages to an object of unknown origin as long as the object has a type that guarantees it has a method with the appropriate signature. Thus objects generated by different classes may be used interchangeably and simultaneously as long as they have the same object type.

### 2.2 Subclasses and inheritance

One of the important features of object-oriented languages is the ability to make incremental changes to a class by creating a subclass (called a derived


```plaintext
class ClrCellClass inherits CellClass modifies set {
    color: ColorType := blue;

    function getColor(): ColorType is
    { return self.color } 

    function set(nuVal: int): Void is
    { self.x := nuVal;
      self.color := red }
}

ClrCellType =
    ObjectType { get: Void → Integer;
    set: Integer → Void;
    bump: Void → Void;
    getColor: Void → ColorType
    }
```

**Figure 2.1** ClrCellClass defined as a subclass of CellClass.

---

A subclass may be defined from a class by either adding to or modifying the methods and instance variables of the original class. (We will see later that restrictions on the modification of the types of methods and instance variables in subclasses are necessary in order to preserve type safety). If class B is a subclass of C, we say that C is a **superclass** of B.

Figure 2.1 includes an example of a subclass of CellClass that adds color to the objects. The *inherits* clause specifies the class from which the new class inherits, while the *modifies* clause indicates which methods from the superclass will be overridden in the subclass. Because ClrCellClass inherits from CellClass, it automatically has all of the instance variables and methods from CellClass, including both those defined in CellClass and any that were inherited. ClrCellClass adds a new instance variable color of type ColorType with initial value blue. It also adds a new method `get-Color`, and *overrides* the method `set` with a new method body that not only updates x, but also sets color to be red. That is, the body of the method `set` contained in the class definition replaces the inherited body from Cell-
Class. We insist that names of methods that are intended to be overridden in the subclass be listed in the “modifies” clause as a double-check in order to ensure that methods are not accidentally overridden. Our type-checking rules will generate an error if a method is overridden, yet its name is not listed in the modifies clause.

Because of inheritance, objects generated from ClrCellClass contain the instance variable x declared in CellClass as well as the new instance variable color. Similarly, they contain methods get and bump as well as the new getColor and the redefined set. The type of objects generated from ClrCellClass is ClrCellType.

When new definitions are given to methods in a subclass, it is useful to be able to refer to the methods of the superclass. For instance, in overriding a method, one often wishes to apply the method body from the superclass, and then perform a few more operations before returning from the redefined method. We provide a keyword, super, to provide access to the methods of the superclass. Using super, we can rewrite the body of set above to be

\[
\begin{align*}
\{ \ & \text{super set}(\text{nuVal}); \\
\ & \text{self.color := red} \}
\end{align*}
\]

While this does not provide enormous savings in this case, in other cases it can. Moreover it can guarantee that any changes made later to the method body in the superclass will automatically be carried over to the subclass.

Dynamic method invocation plays an important role during inheritance. Recall that the body of the bump method in CellClass involves the message send self set(...). Inside CellClass, the method set simply updates the instance variable x. However, inside ClrCellClass, set also updates color. When bump is inherited in ClrCellClass, the message send of self set(...) now results in invoking the method set of ClrCellClass. Thus sending a bump message to an object generated from ClrCellClass will end up both incrementing that object's x instance variable, and updating the value of color to be red. (In the terminology of C++, all methods are virtual.)

2.3 Subtypes

We say type T is a subtype of U, written T <: U, if a value of type T can be used in any context in which a value of type U is expected. That is, a value of type T can masquerade as an element of type U in all contexts if T <: U. We say U is a supertype of T if T is a subtype of U.
2.3 Subtypes

Because values may have multiple types in languages supporting subtyping, we say these languages support *subtype polymorphism* (we shall examine other sorts of polymorphism later). If v has type T, subtype polymorphism allows it to be used in any context that expects a value of some type U as long as T is a subtype of U.

Subtyping depends only on the types or interfaces of values, while inheritance depends upon their implementations. In most simple object-oriented languages, the type of objects generated by a subclass is a subtype of objects generated by the superclass. For example, ClrCellType <: CellType. However, we provide examples later that show that if one enriches the language slightly, a class may inherit from another, yet the type of the objects generated by the subclass may not necessarily be a subtype of the type of the objects generated by the superclass.

It is also not necessary to restrict subtypes to those relationships that arise from subclasses. As long as object type T has at least all of the methods of object type U, and the corresponding methods have the same type, then an object of type T can successfully masquerade as an object of type U. For example, the type of objects generated by ClrCellClass is a subtype of the type of objects generated by DimCellClass, even though they are not subclasses.

By separating object types from classes, we hope to make it clearer that subtyping is a relation between types, while subclassing is a relation between classes. The first has to do with public interfaces, while the second has to do with the inheritance of implementations.

We remark that we are using *structural subtyping* here. Thus subtyping is determined by the structure of the type rather than its declaration. By contrast, Java requires the programmer to explicitly specify when one type is to be a subtype (extension in Java terminology) of another.

In Chapter 5 we examine the subtyping relation very carefully and see that it can allow more variations in object types than we have allowed here. In Chapter 16 we will also define later another relation, *matching*, between object types that is similar to subtyping, but may be even more important in languages with a keyword for the type of self.

---

4. We will loosen this restriction later.
2.4 Covariant and contravariant changes in types

One of the most confusing things in understanding the type systems of statically typed object-oriented languages has been understanding restrictions on changing the types of methods and instance variables inherited from superclasses when defining subclasses. Most statically typed object-oriented languages allow no changes to types in subclasses. However, as we shall see in the next chapter, this puts real restrictions on the expressiveness of the language. As a result there has been pressure on language designers to allow changes to types.

In Chapter 5 we will see why some restrictions on changing types are necessary in order to preserve type safety. For now, we simply introduce the terminology that will allow us to discuss the issue. Figure 2.2 includes the definitions of a class, C, and its subclass, SC. In the subclass we have changed the type of the instance variable v, and the parameter and return types of method m.

If the types in C are replaced by subtypes in SC (i.e., $T_1' <: T_1$, $T_2' <: T_2$, and $T_3' <: T_3$), then the changes are referred to as covariant. This is the most obvious change to make to types in subclasses. For example, C++ now allows the programmer to make covariant changes in the return types of methods, though it allows no changes to the type of instance variables or the parameters of methods.

On the other hand, we will see later that we can preserve type safety if we allow the types of parameters of methods to be replaced by supertypes in...
2.5 Overloading versus overriding methods

Some languages, including both Java and C++, allow programmers to overload method names in classes. A method name is overloaded in a context if it is used to represent two or more distinct methods, and where the method represented by the overloaded name is determined by the type or signature of the method.

Look at the following excerpt from the definition of a class Rectangle.

```java
class Rectangle {
    ...
    function contains(pt: Point): Boolean is
        { ... }
    function contains(x, y: Integer): Boolean is
        { ... }
}
```

The method name `contains` is listed twice. The two versions of `contains` have slightly different signatures as one takes a parameter of type `Point` while the other takes two parameters of type `Integer` that represent the coordinates of a point. It is convenient to use the same name for these methods because they represent essentially the same operation, even though the parameter types are different.

As long as their signatures are different, both Java and C++ treat methods with overloaded names as though they had completely different names. In particular, if code is written using these methods, the language processor statically determines what method body is to be executed. Examine the following code fragment:
var
  r: Rectangle;
  pt: Point;
  x, y: Integer;
function m(...): ... is {
  ... r \leftarrow contains(pt) ... r \leftarrow contains(x, y) ...
}

The language processor can easily determine that the first invocation corresponds to the first definition of contains in Rectangle, while the second invocation corresponds to the second definition.

Let us compare and contrast overloaded and overridden methods. Message sends involving overloaded methods are resolved statically. By contrast, overridden methods always occur in different classes, typically when one of the classes is a subclass of the other. They typically have the same signatures, though some languages allow the overriding method in the subclass to have a signature that is a subtype of the method in the superclass. Message sends involving overridden methods are resolved at run time.

Languages have different rules for when method names may be overloaded. In C++, overloaded methods must be defined in the same class, while in Java, the overloading can happen when a method in a superclass is inherited in a subclass that has a method with the same name, but different signature.

Many programmers believe that overloaded method names make it much easier to understand programs, because they allows the user to reuse names that suggest the operation being performed. For example, in the Rectangle class, both versions of method contains determined whether a particular location, represented alternatively as a Point or as a pair of integer coordinates, was contained in the rectangle.

Moreover, allowing overloading seems to have few consequences for a language, as the language processor automatically renames all overloaded methods with distinct names before runtime.

However, the interaction between overloaded method names with static resolution, and overridden methods with dynamic resolution can result in great confusion as to what methods are called when in a program. We illustrate this with an example.

Class C and subclass SC are defined in Figure 2.3. Class C has an equals method taking a parameter of type CType. Class SC has two equals methods. Thus equals in class SC is overloaded.
class C {
    ...

    function equals(other:CType): Boolean is
        { ... }            // equals 1
}

class SC inherits C modifies equals {
    ...

    function equals(other:CType): Boolean is
        { ... }            // equals 1

    function equals(other:SCType): Boolean is
        { ... }            // equals 2
}

Figure 2.3  Classes with overridden and overloaded method equals.

The first definition of equals in SC takes a parameter of type CType, overriding the equals method of class C. As a result, the comments label them both with equals 1.

The second equals method in class SC takes a parameter of type SC- Type. Because the parameter type is different from the other two definitions of equals, this method is treated as being statically different from the others. As a result, we label it with equals 2.

As usual, let CType and SCType be the types of objects created from those classes. Clearly SCType <: CType.

Let c and c’ be variables with declared type CType, and let sc be a variable with declared type SCType. Consider the following code:

    c := new C;
    sc := new SC;
    c’ := new SC;

    c << equals(c);
    c' << equals(c');
The variable $c$ is assigned an object created from class $C$ and $sc$ is assigned an object created from class $SC$. Variable $c'$ is also assigned an object created from class $SC$. This is legal because $SC$Type is a subtype of $C$Type.

The 9 message sends shown above correspond to all possible combinations of receiver and parameter. Which equals method is actually executed as a result of each of the sends? Think carefully about each case before looking at the answers in the next paragraph.

The answer to this question is rather surprising.

• All 3 message sends to $c$ result in the execution of method $equals\ 1$ from class $C$.
• All 3 message sends to $c'$ result in the execution of method $equals\ 1$ in class $SC$.
• The first two message sends to $sc$ also result in the execution of method $equals\ 1$ from class $SC$.
• Only the last message send, $sc \Leftarrow equals\ (sc)$, results in the execution of method $equals\ 2$ from class $SC$.

Most people get this wrong, even when they understand the rules for overloading given above. Usually the error is thinking that method $equals\ 2$ is selected for some or all of the message sends to $c'$, and for two or more of the message sends to $sc$. The key to understanding which method body is selected in these examples is to remember what is resolved statically and what is resolved dynamically.

The overloading of equals is resolved statically. That is, the selection of $equals\ 1$ versus $equals\ 2$ is resolved solely on the static types of the receiver and parameters.

Because the type of both variables $c$ and $c'$ is $C$Type, when the equals message is sent to $c$ or $c'$, the type system examines the methods in class $C$ at
compile time to determine if there is an appropriate method \texttt{equals}. There is only one method \texttt{equals} in \texttt{C}, and it has a parameter of type \texttt{CType}. That method is appropriate for each of the three actual parameters to \texttt{equals}. The actual parameters \texttt{c} and \texttt{c'} are clearly of the appropriate type. The parameter \texttt{sc} is also fine because its type, \texttt{SCType}, is a subtype of \texttt{CType}. Thus the first 6 method calls are all to \texttt{equals 1}.

The first two message sends to \texttt{sc} have parameters with static type \texttt{CType}. This is an exact match with the signature of \texttt{equals 1} in \texttt{SC}, so they resolve to that method. The last message send has a parameter of type \texttt{SCType}, so its best match is method \texttt{equals 2}.

In summary, the first 8 message sends resolve statically to method \texttt{equals 1}, while the last resolves to method \texttt{equals 2}.

Now all we have to do is figure out which of the first 8 message sends execute the body of \texttt{equals 1} from class \texttt{C}, and which execute the body from class \texttt{SC}. Because we know that all of these resolve statically to method \texttt{equals 1}, we determine which version of \texttt{equals 1} is executed by determining the class that generated the receiver of the message.

This is now easy because the receiver of the first 3 message sends is a \textit{value} generated from class \texttt{C}. Therefore those 3 message sends result in the execution of the body of \texttt{equals 1} from class \texttt{C}. The receivers of the rest of the message sends are \textit{values} generated from class \texttt{SC}. Hence all of those message sends result in the execution of method bodies from class \texttt{SC}.

\textbf{Exercise 2.5.1} Suppose that class \texttt{SC} did not include the first \texttt{equals} method – the one with parameter of type \texttt{CType} that overrode the \texttt{equals} from class \texttt{C}. Determine which of the two remaining method bodies is executed for each of the 9 message sends given above.

Interestingly, the answer is not the same for \texttt{C++} and \texttt{Java} because \texttt{C++} does not allow overloading methods across class boundaries. Look up the rules for each of these languages and determine the correct answer for each. (Hint: The only difference between the two languages with this example is that one or more message sends in \texttt{C++} result in static type errors.)

We hope that this example provides convincing evidence that the combination of static overloading and dynamic method invocation in object-oriented languages is likely to result in confusion on the part of programmers. While many programmers believe that overloading makes it easier to understand programs, we have seen very experienced programmers working on real code incorrectly believe that one method is being executed, when static res-
olution of overloading resulted in a different version of the method being selected.

We strongly recommend that object-oriented languages not support static overloading of method names. As a result we will not further consider static overloading in the languages discussed in this book.

2.6 Summary

In this chapter we defined the fundamental concepts of object-oriented languages, including classes, objects, methods, messages, and instance variables. We also introduced object types, which include the types of public methods, but do not include the names or types of instance variables.

We also discussed the use of inheritance in the definition of subclasses, and noted its differences from the notions of subtype and subtype polymorphism. We also discussed covariant versus contravariant changes of types that may arise when modifying object types. We shall see later that careful identification of locations in type expressions allowing covariant versus contravariant changes in types is very important in obtaining type safety in object-oriented languages.

Finally we discussed static overloading. As a result of the possible confusion in resolving static overloading and dynamic method invocation, we strongly recommend that programmers avoid using static overloading.
We begin our study of the type systems of object-oriented programming languages by first providing a critique of the type systems of existing statically typed object-oriented programming languages. The reason for providing such a critique is that our goal is not only to describe existing programming languages, but also to use a deep understanding of the object-oriented concepts in order to design better object-oriented languages. While the next few chapters will mainly focus on describing the types and semantics of existing languages, we hope this chapter will give the reader a better insight into the overall goals of this work and provide motivation for the later work on extending the expressiveness of object-oriented languages.

3.1 Type checking object-oriented languages is difficult

The features of object-oriented languages that provide added flexibility, like subtyping and inheritance, also create difficulties in type checking these languages. We provide here a brief overview of some of the typing difficulties that arise from them.

While subtyping is trivial for simple types, defining a correct notion of subtype for more complex types like record, function, and object types can be tricky. In particular, there has been great confusion over what is the proper subtyping rule for functions. Later we explain why the so-called “contravariant” subtyping rule for the types of parameters in function types is correct and why the “covariant” rule may lead to typing errors.

Adding new instance variables and methods to a subclass does not cause typing difficulties, but modifying existing methods may create problems. If the types of parameters or the return type of the modified method differ
from those in the corresponding method of the superclass, it might cause type problems.

If the method $m$ being modified was used in a second method $n$ of the superclass, then changes in types in $m$ may destroy the type correctness of $n$ when it is inherited in the subclass. We will see later that considerations involving subtyping can be used to determine which changes in types are guaranteed to preserve type safety.

Another important type-related question that arises with subclasses is determining whether the type of an object generated from a subclass is always a subtype of the type of an object generated from the superclass. While most current object-oriented languages have type systems that ensure this is the case, we will see that it need not hold if the language contains certain features that provide for more flexible constructions of subclasses.

These and other features of object-oriented languages have made it difficult to create statically typed object-oriented programming languages that are both very expressive and type safe. The following enumerates some of the strengths and weaknesses of the type-checking systems of some of the more popular object-oriented languages (or the object-oriented portions of hybrid languages$^1$).

- Some show little or no regard for static typing (e.g., Smalltalk).
- Some have relatively inflexible static type systems, requiring type casts to overcome deficiencies of the type system. These type casts may be unchecked, as in C++ and Object Pascal [Tes85], or checked at run time, as in Java.
- Some provide mechanisms like “typecase” statements to allow the programmer to instruct the system to check for more refined types than can be determined by the type system (e.g., Modula-3 [CDG+88], Simula 67 [BDMN73], and Beta [KMMPN87]).
- Some allow “reverse” assignments from superclasses to subclasses, which require run-time checks (e.g., Beta, Eiffel [Mey92]).
- Some require that parameters of methods overridden in subclasses have exactly the same types as in the superclasses (e.g., C++, Java, Object Pascal, and Modula-3), resulting in less flexibility than would be desirable, while

$^1$ We consider a hybrid language to be one that attempts to support multiple paradigms. C++ and Object Pascal are examples of languages that attempt to support both procedural and object-oriented styles of programming.
3.2 Simple type systems are lacking in flexibility

Languages like Object Pascal, Modula-3, and C++ arose as object-oriented extensions of imperative programming languages. These languages, as well as Java, have relatively simple and straightforward type systems whose features are similar to those of the procedural languages from which they were derived. In these simple type systems, the programmer has little flexibility in redefining methods in subclasses. They require that a redefined method have exactly the same type as the original method in the superclass. Similarly the types of instance variables may not be changed in subclasses. We refer to type systems that restrict the types of methods and instance variables in subclasses to be identical to those in superclasses as invariant type systems.

Interestingly, in these invariant systems, when a method is inherited or redefined in the subclass, the programmer is often able to deduce more refined types for methods than the language allows to be written. For example the programmer may know that a certain method always returns an object of type $DType$ even though the type system of the program restricts the programmer from writing that type as the return type because it does not allow changes to method types in subclasses. We present examples illustrating this below.

As mentioned earlier, we will find it helpful to keep the notions of class and type separate. The type represents only the public interface of the object (in our case the names and types of all of the methods, but not the instance variables), while the class includes names, types, and initial values for instance variables and names, types, and code for methods.

We will often use the convention in this book of writing $CType$ for the type of objects generated by a class named $C$ or $CClass$. In Chapter 2, we fol-
lowed that convention in using CellType as the name of the type of objects generated by class CellClass.

### 3.2.1 The need to change return types in subclasses

In our first examples we show that it is useful to be able to modify the return types of methods when the methods are redefined in subclasses (and sometimes even when they are not).

In most pure object-oriented languages (e.g., Eiffel, Java, and Smalltalk, as well as the languages we introduce later in this text), all objects are represented as references (i.e., implicit pointers). Thus assignment results in sharing, rather than copying. In these languages, it is useful to have an operation that makes a new copy or clone of an object. A common way of supporting this is to provide a built-in clone method in the top-most class of the object hierarchy (called Object in Java), so that all other classes automatically inherit it. For the rest of this chapter we assume that our language has a top-most class named Object, and its type is ObjectType.

A shallow copy is made by copying the values of instance variables and taking the same suite of methods as the original. If the instance variables hold references to other objects, only the references are copied, not the objects being referred to. Thus if this shallow clone method is applied to the head of a linked list, only the head node is copied while the rest of the list is shared between the new and old lists.

What should be the type of clone? When defined in the class Object, it seems apparent that it should return a value of type ObjectType. However, when this is inherited by a class CellClass, we would like it to return a value of type CellType. In the invariant type systems, the return type of clone remains ObjectType, even though the method actually returns a value that is a cell! That is, the semantics of the language does the correct thing, but the type system is not expressive enough to capture that. Instead the programmer must perform a type cast or other operation after the clone method has returned in order to allow the system to treat the value as having the proper type!

Often it is desirable to write a deepClone method that is based on clone. This is typically done by first writing code to send the clone message to self to make the shallow copy, and then writing code to clone all objects held in the instance variables of the original object.

Suppose we have a class C that includes a method deepClone, which returns an object of type CType. See Figure 3.1. Suppose we now define a sub-
class \textit{SC} of \textit{C} that includes a new method, \textit{newMeth}, as well as a new instance variable, \textit{newVar}, holding an object with type \textit{newObjType}. We assume for simplicity that \textit{newObjType} also supports a \textit{deepClone} method.

We would like to redefine \textit{deepClone} to clone the contents of this new instance variable after all of the code in the original \textit{deepClone} method from \textit{C} has been executed. (This is a quite common desire in real object-oriented languages as subclass methods cannot obtain access to private instance variables from the superclass.)

Unfortunately, the rules of the simple type systems require that \textit{deepClone} for \textit{SC} also return a \textit{CType}, just as in \textit{C}, even though it is obvious that it actually returns an object of type \textit{SCType}. While this is not type-unsafe, it represents an unnecessary loss of information in the type system.

Suppose \textit{anSC} is a variable of type \textit{SCType}. If we write \texttt{(anSC \leftarrow \textit{deepClone}()) \leftarrow \textit{newMeth}()}, the type checker will complain, even though the object resulting from \texttt{anSC \leftarrow \textit{deepClone}()} has the method \textit{newMeth}. The problem is that the type system is not aware of this!

In these circumstances, Object Pascal, C++, and Java programmers would normally be forced to perform a type cast to tell the compiler that the cloned object has the type \textit{SCType}. In the case of Object Pascal and C++, the type cast is unchecked. In Java, it would be checked at run time. Modula-3 programmers would typically use a \texttt{typecase} statement that also performs a run-time check to get the same effect.

One could attempt working around these deficiencies in the static type system by making up a new name for the revised \textit{deepClone} method (e.g., \textit{SCdeepClone}). Unfortunately this would mean that inherited methods that included a message send of \textit{deepClone} would call the old \textit{deepClone} for \textit{C} rather than the updated method from \textit{SC} actually desired.

For example, suppose \textit{C} includes a method \textit{m} invoking \textit{deepClone} as follows

```pascal
function m(): Void is
    { ... 
      self \leftarrow \textit{deepClone}() 
      ... 
    }
```

Also suppose class \textit{SC} is defined as a subclass of \textit{C} as in Figure 3.1, except that it adds a new method \textit{SCdeepClone} that returns a value of type \textit{SCType}. If \textit{sc} has type \textit{SCType}, then \texttt{sc \leftarrow \textit{SCdeepClone}()} will certainly result in the newly defined clone method being called. However the execu-
class C {
    ...

    function deepClone(): CType is
    { self ← clone(); ... }
}

class SC inherits C modifies deepClone {
    newVar: newObjType := nil;

    function newMeth(): Void is
    { ... }

    function setNewVar(newVarVal:newObjType): Void is
    { self.newVar := newVarVal }

    function deepClone(): SCType is
    // illegal return type change!
    // Must return CType instead
    var // local variable declaration
    newClone: SCType := nil;
    { 
        newClone := super ← deepClone();
        // (*) another problem
        newClone ← setNewVar(newVar ← deepClone());
        return newClone
    }
}

Figure 3.1 Typing deepClone methods in subclasses.

...
the way of the programmer, even though the run-time system does the right thing. Not surprisingly, we have similar problems even writing down the type of the built-in (shallow) clone. In Java, clone is simply given a type indicating that it returns an element of the top class, Object. The result must then be cast to the appropriate type before anything substantial may be done with it.

In newer versions of C++, it is possible to specialize the return type of methods in subclasses. Thus method deepClone in SC could be specified to return a value of type SCType rather than CType. Unfortunately this does not solve all of our problems. First note that the right side of the assignment on line (*) of Figure 3.1 returns a value of type CType. Because the type of the variable on the left side is a subtype of CType, the assignment is illegal. (A value of the subtype can masquerade as a supertype, not the reverse!) Thus a type cast would have to be inserted to make the assignment legal.

Moreover, suppose class SC has a subclass SSC that adds new methods, but no new instance variables. As a result, there is no need to override method deepClone. However, if it is not overridden then it will continue to return type SCType rather than the desired SSCType. To get the types right, the programmer would have to override deepClone solely to cast the return type of the call of the superclass to the new type.

Thus, allowing covariant changes to the types of methods in subclasses would be helpful, but it would be even more helpful if some way could be found to have them change automatically. In the next section we show that it would also be convenient to be able to change the types of method parameters in subclasses.

### 3.2.2 Problems with binary methods

Our next class of examples is one that arises surprisingly frequently in practice. The particular typing problem arises in connection with what are often called *binary methods*. Binary methods are methods that have a parameter whose type is intended to be the same as the receiver of the message. Methods involving comparisons, such as `eq`, `lt`, and `gt`, or other familiar binary operations or relations are common examples of such methods (e.g., `someElt <= lt(otherElt)`). In procedural languages, these methods would be written as functions that take two parameters (hence the “binary”). However, they are written with single parameters in object-oriented languages because the receiver of the message plays the role of the other pa-
rameter. Other compelling examples arise in constructing linked structures (e.g., \(\text{node} \leftarrow \text{setNext} \left(\text{newNext}\right)\)).

While it is not difficult to define binary methods when specifying classes from scratch, it is much more problematic for subclasses. Suppose we define class \(C\) below with method \(\text{equals}\):

```
class C {
  ...
  function equals(other: CType): Boolean is
    { ... }
  ...
}
```

In the example we use our convention that \(C\text{Type}\) is the type of objects generated from class \(C\). If \(o\) is generated from class \(C\), the signature of \(\text{equals}\) requires that the parameter \(o'\) in \(o.\text{equals}(o')\) have type \(C\text{Type}\) for the message send to be well-typed.

However, we have problems when we define a subclass \(SC\):

```
class SC inherits C modifies equals {
  ...
  function equals(other: CType): Boolean is
    // Want parameter type to be \(S\text{CType}\) instead!
    { super \leftarrow equals(other);
      ... // Can’t access \(SC\)-only features in other
    }
  ...
}
```

Because we are not allowed to change the type of parameters in subclasses, we cannot change the type of \(C\text{Type}\) to \(S\text{CType}\), even though that may be what is desired here.

Similarly to our previous example with \(SC\text{deepClone}\), changing the name of the method to \(\text{newEquals}\) does not help. If there are occurrences of \(\text{equals}\) in the inherited methods of the superclass, we would like the new method body to be called, but instead the old would be invoked. Overloading the name \(\text{equals}\) does not help either. Overloading is resolved statically rather than at run time, so the problem of calling the wrong method is no different than using the new name above.
3.2 Simple type systems are lacking in flexibility

Programmers using languages with invariant type systems sometimes use the following trick of overriding equals in the subclass with a body that casts the argument to the desired type before it is used.

```plaintext
class SC' inherits C modifies equals {
    ...
    function equals(other: CType): Boolean is
        var
            otherSC: SCType := nil;
        { otherSC := (SCType)other; // type cast!
        ...
            return super \leftarrow equals(other) & ... }
    ...
}
```

The expression `(SCType) other` represents casting the expression `other` to type `SCType`. However, these casts can fail at run time. This technique requires the programmer to be quite disciplined in adding casts to all overridden versions of binary methods. It also suffers from the twin disadvantages of adding run-time checks to each execution of a method, as well as requiring the programmer to handle the situation where the cast would fail. Clearly it would be a significant advantage both in programming and execution time to be able to check such calls statically.

A second example of the problems with binary methods arises from linked structures. Figure 3.2 contains a definition of the class, `Node`, which generates objects of type `NodeType` that form nodes for a singly linked list of integers. In the class there is one instance variable, `value`, for the value stored in the node, and another, `next`, to indicate the successor node. There are methods `getValue` and `setValue` to get and set the values stored in the node, and methods `getNext` and `setNext` to get and set the successor of the node.

Notice that method `getNext` returns a value of type `NodeType`, the type of object generated by class `Node`, while the `setNext` method takes a parameter of type `NodeType`. Thus the type `NodeType` is recursively defined. It is not uncommon to have such recursively-defined types in object-oriented languages.

Suppose we now wish to define a subclass of `Node`, `_DoubleNode`, which implements doubly linked nodes, while reusing as much as possible the code for methods in `Node`. Figure 3.3 contains an attempt at defining such a
NodeType = ObjectType {
    getValue: Void → Integer;
    setValue: Integer → Void;
    getNext: Void → NodeType;
    setNext: NodeType → Void
}

class Node {
    value: Integer := 0;
    next: NodeType := nil;

    function getValue(): Integer is
    { return self.value }

    function setValue(newValue:Integer): Void is
    { self.value := newValue }

    function getNext(): NodeType is
    { return self.next }

    function setNext(newNext:NodeType): Void is
    { self.next := newNext }
}

Figure 3.2  Node class.

subclass, DoubleNode. DoubleNode adds to Node an additional instance variable, previous, as well as new methods to retrieve and set the previous node. If DoubleNodeType is the type of objects generated from the class, then we will want both the next and previous instance variables to have type DoubleNodeType. Similarly, the methods that get and set next or previous nodes should take parameters or return values of type DoubleNodeType rather than NodeType. This is particularly important because we do not want to allow the attachment of a singly linked node to a doubly linked node.  

2 The method setPrev is not really intended for public use because it only sets one of the two links. Normally it would be given a designation that would indicate this. Because we have not
3.2 Simple type systems are lacking in flexibility

DoubleNodeType = ObjectType {
    getValue: Void -> Integer;
    setValue: Integer -> Void;
    getNext: Void -> NodeType;
    setNext: DoubleNodeType -> Void;
    getPrev: Void -> DoubleNodeType;
    setPrev: DoubleNodeType -> Void
}

class DoubleNode inherits Node modifies setNext {
    previous: DoubleNodeType := nil;

    function getPrev(): DoubleNodeType is
        { return self.previous }

    function setPrev(newPrev: DoubleNodeType): Void is
        { self.previous := newPrev }

    function setNext(newNext: DoubleNodeType): Void is
        // error - illegal change to parameter type
        { super setNext(newNext);
          newNext := setPrev(self) }
}

Figure 3.3 Doubly linked node class — with errors.

Unfortunately, in the simple type system described here, we have no way of changing these types, either automatically or manually, in the subclass. The class DoubleNode defined in the figure illegally changes the parameter type of method setNext in a covariant way. It is also troublesome that method getNext returns type NodeType, while setNext takes a parameter of type DoubleNodeType.

Suppose we create LglDbleNode as a legal subclass of Node. We might write it as shown in Figure 3.4. Problems become apparent when overriding the setNext method. We cannot send a setPrev message to the bare parameter newNext, since its declared type is NodeType rather than LglD-
bleNodeType. As a result the programmer must insert a cast to tell the type checker to treat it as though it has type LglDbleNodeType.

However, if a programmer sends setNext to an object generated from LglDbleNode with a parameter that is generated from Node, it will not be picked up statically as an error. Instead the cast will fail at run time.

Even if a variable dn has type LglDbleNodeType, the evaluation of

\[(dn \leftarrow \text{getNext()}) \leftarrow \text{getPrev()}\]

will generate a static type error, because the type checker can only predict that the results of \(dn \leftarrow \text{getNext()}\) will be of type NodeType, not the more accurate LglDbleNodeType. Thus, even if the programmer has created a list, all of whose nodes are of type LglDbleNodeType, the programmer will still be required to write type casts to get the type checker to accept the program.

To get the desired types for the instance variable next and the methods getNext and setNext, we would instead have to define DoubleNode independently of Node, even though much of the code is identical. This is clearly undesirable.

We do not bother to show the code for an independently defined class, IndDoubleNode, with the same behavior as DoubleNode. We leave it as an exercise for the reader to write it and notice how similar the code is to that of Node. However, we show the corresponding object type, IndDoubleNodeType, in Figure 3.5. Even such an independent definition has problems. While we have not yet discussed the rules for determining when object types are in the subtype relation, the code in Figure 3.5 can be used to show that the resulting type, IndDoubleNodeType, cannot be a subtype of NodeType.

The function breakit in the figure is well-typed, as setNext takes a parameter of type NodeType, and the expression new Node as the actual parameter creates a value of type NodeType.

Suppose IndDoubleNodeType were a subtype of NodeType. If so, and if dn was a value generated from IndDoubleNode, then breakit(dn) would be well-typed, as values of type IndDoubleNodeType could masquerade as elements of type NodeType. However, that is not the case!

The execution of node \(\leftarrow \text{setNext(newNode)}\) in the body of breakit would result in the message send of setPrev to the parameter newNext. But newNext holds a value that is generated from Node. This would result in a run-time type error as elements generated from Node have no setPrev method. This shows that object type IndDoubleNodeType could not be
3.2 Simple type systems are lacking in flexibility

LglDbleNodeType = ObjectType {
    getValue: Void → Integer;
    setValue: Integer → Void;
    getNext: Void → NodeType;
    setNext: NodeType → Void;
    getPrev: Void → LglDbleNodeType;
    setPrev: LglDbleNodeType → Void
}

class LglDbleNode inherits Node modifies setNext {
    previous: LglDbleNodeType := nil;

    function getPrev(): LglDbleNodeType is
        { return self.previous }

    function setPrev(newPrev: LglDbleNodeType): Void is
        { self.previous := newPrev }

    function setNext(newNext: NodeType): Void is
        { super
            setNext(newNext);
            ((LglDbleNodeType)newNext) ← setPrev(self) }
        // cast necessary to recognize setPrev
}

Figure 3.4 Legal doubly linked node class — with cast.

a subtype of NodeType, as elements of type IndDoubleNodeType cannot safely masquerade as elements of type NodeType.

This example is particularly troubling in that it seems to be tailor-made for the use of inheritance, but it (i) cannot be written correctly with an invariant type system, and (ii) would result in a type error if the type of objects generated from the desired subclass were a subtype of the type of objects generated from the original class. These problems are not special to the Node example, but arise with all binary methods because of the desire for a covariant change in the parameter type of binary methods.

We will later find a way to add expressiveness to languages in order to allow us to write DoubleNode as a subclass of Node. We will then need to
IndDoubleNodeType = ObjectType {
    getValue: Void → Integer;
    setValue: Integer → Void;
    getNext: Void → IndDoubleNodeType;
    setNext: IndDoubleNodeType → Void;
    getPrev: Void → IndDoubleNodeType;
    setPrev: IndDoubleNodeType → Void
}

function breakit(node: NodeType): Void is
{ node ← setNext(new Node) }

var
    n: NodeType
    dn: IndDoubleNodeType
{
    n := new Node;
    dn := new IndDoubleNode;
    breakit(n); // No problem
    breakit(dn) // Run-time error here!
}

Figure 3.5 Example showing why IndDoubleNodeType cannot be a subtype of NodeType.

ensure that the resulting object types are not subtypes in order to avoid the second problem.

invariant type systems do have the desirable property that subclasses generate subtypes, but it may be worth losing subtyping in some circumstances if it makes it significantly easier to define desirable subclasses. We will evaluate these trade-offs when we examine the more flexible type system of Eiffel in the next chapter.

3.2.3 Other typing problems

In both of the examples above, the difficulties arose from an attempt to keep return or parameter types the same as those of the object being defined. While this is an extremely important special case, there are other examples
3.2 Simple type systems are lacking in flexibility

class CircleClass {
    center: PointType := nil;
    ...
    function getCenter(): PointType is
        { return self.center }
    ...
}

class ColorCircleClass inherits Circle modifies getCenter {
    color: ColorType := black;
    ...
    function getCenter(): ColorPtType is { ... }
        // illegal type change in subclass!
    ...
}

Figure 3.6 Circle and color circle classes.

where it is desirable to change a type in a subclass in a covariant way. In these examples, the type to be changed may have no relation to the type of objects generated by the classes being defined. Many examples of this phenomenon arise when we have objects with other objects as components.

Figure 3.6 contains the definition of a class, CircleClass, with a getCenter method that returns a point. If we define a subclass that represents a color circle, it would be reasonable to wish to redefine getCenter to return a color point, as in the code in the figure. This would be illegal by the rules on method types in these invariant object-oriented languages. We would like to have a typing system that allows such changes, as long as they are type safe.

Again, however, even if we allowed a change in the return type of getCenter in the subclass, we still have the problem that we cannot change the type of the instance variable center. If center is still of type PointType, we will either need to add a type cast (which may fail) to the body of getCenter or we may have to change the body to build a new color point from the color and center instance variables. The latter involves a lot of work each time getCenter is called, and is probably not worth the ef-
fort compared to separately returning the values of center (as a point) and color.

Finally, there likely will be a method `setCenter` in `CircleClass` that takes a parameter of type `PointType`. Even C++ will not allow changing the types of parameters of methods in subclasses, so `setCenter` in `ColorCircleClass` must accept parameters of type `PointType`. Thus if we wish `center` to have type `ColorPtType` in the subclass, we will have to add a dynamic check before assigning the value or live with the possibility that the value could fail to be a color point.

### 3.3 Summary of typing problems

In this chapter we illustrated several problems with invariant type systems. In each case the difficulty arose from a desire to change the types of methods that are modified in subclasses. However no changes to the types of methods are allowed in invariant type systems.

In order to help alleviate the rigidity in these simple systems, C++ allows changes to the return types of methods in subclasses. This provided some help with overriding `deepClone` and `getCenter` methods, but as we saw above, we are still left with significant problems to be overcome. Moreover, it provides no help in taking care of problems arising in examples involving binary methods, in particular those involved with redefining the type of the instance variable `next` and the parameter type of the method `setNext` in the `DoubleNode` subclass. In Chapter 16 we present a detailed design for an extension to invariant type systems that, when combined with parametric polymorphism, provides one possible solution to all of these problems.

Given these examples, readers may be wondering why statically typed object-oriented languages are so restrictive in not allowing covariant changes to the types of instance variables or the parameters of methods in subclasses. We put off this discussion until after we have discussed the rules for subtyping in chapter 5. For now we content ourselves with the explanation that we are simply examining the restrictions in the most popular existing statically typed object-oriented languages. In the next chapter we examine some more expressive type systems for object-oriented languages that will help overcome these difficulties.
4 Adding Expressiveness to Object-Oriented Languages

In the last chapter, we looked at the limits of expressiveness of statically typed object-oriented languages with invariant typing disciplines. These are languages in which the types of instance variables and methods of a class are not allowed to vary in subclasses. Moreover, we saw that allowing covariant changes to the return types of methods (like C++ does) provided only a small amount of help, as the other restrictions in the language kept us from defining the instance variables and methods in the subclass as directly as we desired.

In this chapter we explore ways of increasing expressiveness provided by some existing object-oriented languages. The languages examined are Gj, an extension to Java supporting parametric polymorphism, and Eiffel, a statically typed object-oriented language that allows a great deal of flexibility in changing the types of methods and instance variables in subclasses.

4.1 GJ

In this section we discuss an extension, GJ [BOSW98], of the object-oriented language Java. Before going into the details of Java, we first present a brief introduction to the notion of polymorphism, and then a short introduction to Java syntax.

4.1.1 Parameterized types and polymorphism

Many families of types are conveniently expressed using parameterized types. A parameterized type is a type expression that takes another type as a parameter. Examples of parameterized types include array of T, stack of T,
tree of $T$, list of $T$, etc.\textsuperscript{1} Most programming languages include array as a built-in parameterized type, but only relatively modern programming languages like Clu, Ada, ML, Haskell, C++, and Eiffel allow the programmer to define parameterized types or classes.

Many algorithms, for example, sorting, are generic, meaning that they depend minimally on the type of elements being manipulated. The difference between the code for a quicksort of an array of reals and the quicksort of an array of integers is typically only in the type declarations of variables (and the type of the comparison operator). The actual algorithm is the same in the two cases. The types of these generic operations can be given by parameterized types. Thus a generic sort can be applied to a value of type array of $T$, a generic tail function can be applied to a value of type list of $T$, etc. Operations that can be applied to values of more than one kind are called polymorphic operations.

In section 2.3 we defined subtype polymorphism. A language supporting subtype polymorphism allows values to be given multiple types; in particular, if a value has type $T$, it also has any supertype of $T$.

Older languages like Pascal and C force the programmer to create new copies of code in order to handle different instances of generic operations, even though the code is nearly identical. Thus separate copies must be made of the code to perform a quicksort of an array of integers and of an array of reals. In languages with polymorphic type systems, one can define parameterized types and polymorphic operations that operate uniformly on these parameterized types.

A language supporting parametric polymorphism allows an operation to be applied to arguments of a parameterized family of types. Thus in a language with a parameterized type of the form list of $T$, a single polymorphic tail function can be applied to any list with type list of $T$, no matter what type $T$ is.

One source of frustration for Java programmers has been its lack of support for parameterized types or classes. By contrast, C++’s templates allow programmers to define, for example, a stack template class that can be instantiated with the type of element to be held in the stack. While C++’s templates have not been without problems\textsuperscript{2}, programmers moving from C++ to Java

\textsuperscript{1} These may be thought of more accurately as functions from types to types.
\textsuperscript{2} Templates are generally not type checked until they are instantiated. Moreover, because C++’s templates are typically compiled into disjoint code segments for each instantiation rather than a single shared code segment, problems arise with code bloat. Support for parameterized types has generally been better in the other cited languages.
have found the lack of support for parameterized classes a major limitation of the language.

The Java extension, GJ (think Generic Java\textsuperscript{3}), attempts to remedy this omission by adding parameterized classes (types) to Java.\textsuperscript{4} The syntax of parameterized classes in GJ is similar to C++’s templates, though the type checking and implementation are quite different.

Later in this section we will describe the extensions to Java and how they increase the expressiveness of the language. However we first take a brief excursion to review Java syntax for those who may not be familiar with it.

### 4.1.2 Quick review of Java syntax

Java syntax is very similar to C++ syntax, which is itself derived from C syntax. The reader is referred to Gosling and Arnold [AGH99] or any other Java text for more details.

Java’s interfaces are similar to the object types we discussed in earlier chapters. Interfaces \texttt{PointIfc} and \texttt{ColorPointIfc} are given in Figure 4.1. Interfaces include method headers, but not their bodies. Interfaces never include instance variables. All methods listed in interfaces are public by default. An interface may be declared to extend another by adding new methods. Thus \texttt{ColorPointIfc} inherits the \texttt{move} and \texttt{atOrigin} methods from \texttt{PointIfc}.

Because \texttt{ColorPointIfc} extends \texttt{PointIfc}, any value satisfying \texttt{ColorPointIfc} may be used in any context that expects a value satisfying \texttt{PointIfc}. Thus interface extensions can serve as subtypes in Java.\textsuperscript{5}

Figure 4.2 contains two Java class definitions. Note that Java comments begin with // and continue until the end of the line. The class \texttt{Point} contains two private instance variables, \texttt{x} and \texttt{y} of type \texttt{int}. The next item listed is a \texttt{constructor} that is used to create (or generate) a new object from the class. Evaluating \texttt{new Point(7,3)} results in the creation of a \texttt{Point} object whose instance variables have values 7 and 3. Finally the class contains two public methods, \texttt{move} and \texttt{atOrigin}. The method \texttt{move} is a procedure

\textsuperscript{3}..., but not too loud, as the name "Java" is trademarked by Sun Microsystems.

\textsuperscript{4} GJ or a similar extension of Java supporting parametric polymorphism is likely to be incorporated into Java in the near future. It may already be part of Java by the time you read this.

\textsuperscript{5} Technically, this is not quite right as subtyping is not fully supported in Java conditional expressions. However, for simplicity in the exposition we'll blithely ignore this detail and pretend Java supports subtypes.
public interface PointIfc {
    void move(int dx, int dy);
    boolean atOrigin();
}

public interface ColorPointIfc extends PointIfc {
    void setColor(ColorType newColor);
    ...
}

Figure 4.1   Point interfaces in Java.

that adds dx and dy to the corresponding instance variables. The method atOrigin returns true if and only if both instance variables are 0.

A class may be declared to implement an interface if it contains every method declared in the interface (and perhaps more). Class Point may legally implement PointIfc because Point includes methods move and atOrigin. Similarly ColorPoint may be declared to implement ColorPointIfc.

Subclasses in Java are defined as extensions of their superclasses. In Figure 4.2, ColorPoint is defined as a subclass of Point. An object generated from a subclass can be used in a context expecting a value of the superclass. Thus subclasses can also serve as subtypes.

An object generated from a class SomePoint can be used in a context expecting a value satisfying interface PointIfc only if SomePoint or one of its superclasses is declared to implement either PointIfc or an interface that extends PointIfc. Thus an object generated from either of the classes Point or ColorPoint defined in Figure 4.2 can be used in a context expecting a value satisfying interface PointIfc.

4.1.3   Expressiveness problems in Java

A parameterized class is a class that takes a type parameter. An example might be a class for a stack that takes a type parameter representing the type of object held in the stack. Parameterized classes have been included in languages like Clu, Ada, and Eiffel, and as templates in C++ (though C++'s implementation is not as robust as those in the other languages).

Java does not currently support parameterized classes. Here is an example
public class Point implements PointIfc {
    private int x; // instance variables
    private int y;

    public Point(int newX, int newY) // constructor
    {
        x = newX;
        y = newY;
    }

    public void move(int dx, int dy) // methods
    {
        x = x + dx;
        y = y + dy;
    }

    public boolean atOrigin()
    {
        return (x==0) && (y==0);
    }
}

public class ColorPoint extends Point implements ColorPointIfc {
    private ColorType color;
    ...
    public void setColor(ColorType newColor) {...}
}

Figure 4.2  Point class in Java.
where parameterized classes would be useful. Suppose a Java programmer wishes to write programs that use stacks holding different sorts of elements. The programmer could write one class for a stack of strings, another class for a stack of points, and so on, but when the code is examined, it becomes clear that aside from the types of instance variables and methods, the code is exactly the same for each of these classes. It seems unreasonable (and error-prone) for programmers to have to repeatedly write nearly identical code for each type of stack.

One way of working around this omission in Java is to use the class `Object`, which is a superclass of all object types in the language. Programmers can create data structures holding elements of type `Object` (or some other class guaranteed to be a superclass of all classes of elements likely to be used), and take advantage of subtyping to allow the use of objects generated by these other classes in the data structure.6

For example, suppose a programmer wants to use the built-in library class `java.util.Stack`. Its declaration is similar to:

```java
public class Stack extends Vector {
    public void push(Object item){...}
    public Object pop(){...}
    public Object peek(){...}
    public boolean empty(){...}
    public int search(Object o){...}
}
```

Thus the programmer can push and pop elements of type `Object` with the stack. Suppose she wishes to manipulate elements of class `Point` with the stack `myStack`. Creating the stack and inserting items is no problem:

```java
Stack myStack = new Stack();
Point aPoint = new Point(2,3);
myStack.push(aPoint);
```

The use of a `Point` as the parameter to `push` is well-typed because `Point` is a subclass (and hence a subtype) of `Object`.

However, suppose we wish to remove the top element from the stack and then send it a `move` message. The static type system believes the result of

---

6. Java allows the use of classes as types, so we shall follow Java in this section and not make a distinction between the two. This causes no difficulties here because subclasses always give rise to subtypes.
evaluating \texttt{myStack.pop()} is an element of type \texttt{Object} because of the typing of \texttt{pop}, even though we know we pushed a \texttt{Point} on the stack. Because \texttt{Object} does not contain the method \texttt{move}, we must first \textit{cast} the element to type \texttt{Point} before sending the \texttt{move} message:

\[
\text{Point stackPoint} = \texttt{((Point)(myStack.pop())))\texttt{.move(1,2)};
\]

Type casts work as follows. Let \texttt{exp} be an expression and \texttt{SomeType} a class or interface name. A type cast expression, written as \texttt{(SomeType)exp}, tells Java to treat the cast expression as if it had class (interface) \texttt{SomeType}. Java will check this at run time and raise an exception if the class (interface) of the value of \texttt{exp} is not an extension of \texttt{SomeType}.

Thus the displayed expression above will raise an exception if the result of \texttt{myStack.pop()} is not of class \texttt{Point} or one of its subclasses. To avoid the danger of raising an exception, the Java programmer can use the built-in \texttt{instanceof} function to determine if the cast will succeed.

It is worse in Java if we wish to put elements of type \texttt{int} on the stack, as base types (including \texttt{int}) are not object types and hence are not considered subtypes of \texttt{Object}. However, Java does provide object types corresponding to each of the base types in the language. For example, the built-in class \texttt{Integer} corresponds to \texttt{int}. Elements of type \texttt{int} can be \textit{wrapped} up as Integer’s and used in contexts expecting objects. Thus, to push an integer \texttt{n} on the stack, we must push on the corresponding Integer via \texttt{myStack.push(new Integer(n))}. Popping off such an element is also painful as you must pop it off, cast it to \texttt{Integer}, and then send it the \texttt{intValue()} message before being able to do anything with the value:

\[
\ldots \texttt{((Integer)(myStack.pop()))\texttt{.intValue()} \ldots}
\]

There are two major problems with the stack implementation using \texttt{Object}:

1. We can’t enforce a rule that the stack only hold items of a fixed type.
2. Whenever we remove something from the stack, we must cast it to the type we expect it to be so that we can send messages to it.

Both problems may be overcome by writing specialized stack implementations for each desired type, but reusing library code would clearly be more desirable if we could avoid these problems.

The result of these two problems is ugly code with many casts; leaving the code open to errors if somehow an element of some other type is pushed onto the stack.
4.1.4 GJ’s parameterized types

In the GJ extension of Java we can rewrite Stack as:

```java
public class Stack<Elt> extends Vector<Elt>{
    public void push(Elt item){...}
    public Elt pop(){...}
    public Elt peek(){...}
    public boolean empty(){...}
    public int search(Elt o){...}
}
```

Here `Stack` is defined as a parameterized class where `Elt` is a type variable that can be instantiated to the type of element to be held in the stack. Thus `Stack<Point>` represents the type of a stack that holds elements of type `Point`.

We can create and add an element to a stack as follows:

```java
Stack<Point> myStack = new Stack<Point>();
Point aPoint = new Point(2,3);
myStack.push(aPoint);
```

Pushing anything other than a `Point` (or an element of a subclass of `Point`) onto the stack results in a static type error.

The code for popping a point off the stack and moving it is now much clearer, as we no longer need casts.

```java
Point stackPoint = myStack.pop().move(1,2);
```

GJ can also handle more sophisticated examples of parameterized classes in which we wish to restrict the type parameters that can be used.

Suppose we wish to implement a class representing ordered lists. In order to insert an element, we must compare it with other elements in the list. As a result, we should only insert elements of classes that provide the appropriate methods comparing elements:

```java
public interface Orderable {
    boolean equal(Orderable other);
    boolean greaterThan(Orderable other);
    boolean lessThan(Orderable other);
}
```

We can then write a class `OrderedList` as follows:
public class OrderedList<E extends Orderable>
    extends ... {
    public void insert(E item){...}
    public E item removeFirst(){...}
    public boolean empty(){...}
    public int searchFor(E elt o){...}
}

In the declaration of OrderedList, the type variable Elt is constrained to implement Orderable. As a result, we are guaranteed that we can send messages equal, greaterThan, and lessThan to expressions of type Elt. Type parameters can be constrained to implement an interface or to extend a class or interface. This variant on polymorphism involving restriction on type variables is called bounded polymorphism.

We can instantiate OrderedList with any class that implements Orderable. This sounds fine, but we soon discover that it is hard to write a class implementing Orderable. The difficulty is that the comparison operators have to take arguments of any other class implementing Orderable. Here is an example of the problem.

public class IntOrd implements Orderable {
    protected int value = 0;
    ...
    public boolean greaterThan(Orderable other){
        if (other != null && other instanceof IntOrd){
            return (value > ((IntOrd)other).value)
        } else { ... raise an exception??...}
    }
    ...
}

The code for greaterThan is now much more complex than expected or desired. We must cast other to type IntOrd (using instanceof to ensure that the cast is legal) before being able to do anything with it. And what should be done if the instanceof test fails? Returning the value false is likely not the appropriate answer; more likely an exception will need to be thrown.

7. A similar difficulty arises in Java when attempting to write an implementation of the equals method, which, because it is inherited from Object, must handle any element of type Object as a parameter.
Moreover, one could virtually never use a pre-existing class to instantiate \texttt{OrderedList}, because the comparison methods of existing classes are highly unlikely to take parameters of type \texttt{Orderable}. Instead they most likely would take parameters of the same type as the class being defined.

For example, the class \texttt{IntOrd} defined above would normally be defined instead as \texttt{IntOrdBC}, below:

```java
public class IntOrdBC {
    protected int value = 0;
    ...
    public boolean greaterThan(IntOrdBC other) {
        return (value > other.value);
    }
}
```

The simple bounded polymorphism illustrated above was first proposed as a programming language construct in the mid-80's by Cardelli and Wegner [CW85]. However, within a few years examples like the above showed that simple bounded quantification was not expressive enough to capture the restrictions on type parameters needed for many important examples that contain binary methods (as described in Section 3.2.2).

A generalized form of parametric polymorphism, \textit{F-bounded polymorphism} [CCH+89], was introduced to handle these more complex cases. This construct has been included in GJ. We illustrate it by example. Rather than defining the interface \texttt{Orderable}, we now define a parameterized interface, \texttt{OrderableF}:

```java
interface OrderableF<T> {
    boolean equal(T other);
    boolean greaterThan(T other);
    boolean lessThan(T other);
}
```

\texttt{OrderableF} is a parameterized interface, which may be instantiated with any type (class or interface) \texttt{T}. It provides methods for making comparisons with elements of type \texttt{T}.

The class \texttt{IntOrdBC} above can implement \texttt{OrderableF<IntOrdBC>} if we rewrite the header as
public class IntOrdBC
    implements OrderableF<IntOrdBC> {
    ...
}

It implements OrderableF<IntOrdBC> because it provides the methods equal, greaterThan, and lessThan, which all take parameters of type IntOrdBC and have return type boolean.

In general, if class Elt implements OrderableF<Elt> then it will support comparison operations that take parameters of type Elt. This is a sort of recursive constraint in that the constraint on the class Elt involves Elt. However, there is no technical difficulty in determining if an existing class satisfies such a constraint. One need only check that it has methods of the appropriate types.

Class OrderedList can now be rewritten as:

public class BPOrderedList<Elt implements OrderableF<Elt>> extends ... {
    public void insert(Elt item){...}
    public Elt removeFirst(){...}
    public boolean empty(){...}
    public int search(Elt o){...}
}

The constraint on Elt is now somewhat harder to read, but it does state exactly the restrictions necessary to instantiate BPOrderedList with a type Elt: Elt must have methods equal, greaterThan, and lessThan, each of which takes a parameter of type Elt and has return type boolean.

Because IntOrdBC satisfies these constraints, we can instantiate parameterized class BPOrderedList with IntOrdBC:

BPOrderedList<IntOrdBC> ordListSC = new BPOrderedList<IntOrdBC>();
IntOrdBC a = new IntOrdBC();
ordListSC.insert(a);
...

GJ’s addition of F-bounded polymorphism to Java has greatly increased the expressiveness of Java’s type system. Whereas earlier one had to use Object or another fixed type to represent a whole collection of possible types that could be used in a data structure, we can now instantiate the class to a specific type as long as it satisfies the given constraints.
Unfortunately, F-bounded quantification does not interact well with the subclass or subtype hierarchies in object-oriented languages. The problem is that if a class satisfies an F-bounded constraint, its subclasses will generally not satisfy the constraint. For example, define:

```java
public class ExtIntOrdBC extends IntOrdBC {
    ...
    public boolean greaterThan(IntOrdBC other) {
        return (value > other.value) && ...
    }
}
```

where as usual we are not allowed to change the type of the method parameter from the superclass.

No matter how few additions or changes are made in the subclass (and even if no methods from the subclass are overridden), ExtIntOrdBC will implement OrderableF<IntOrdBC>, not OrderableF<ExtIntOrdBC>. (We will see later that OrderableF<ExtIntOrdBC> also cannot be a subtype of OrderableF<IntOrdBC> because of the covariant change to the parameter of greaterThan.) As a result the subclass cannot be used with BPOrderedList.

The difficulty is that in Java we are not allowed to change the types of method parameters in subclasses. As a result, subclasses do not fit in consistently with the use of F-bounded polymorphism. We will see later that the use of a "MyType"-construct and the notion of match-bounded polymorphism gives similar expressiveness as F-bounded polymorphism, but is consistent with the subclass relation.

In summary, GJ’s addition of F-bounded polymorphism to Java helps to increase the expressiveness of Java and aids the programmer to express type constraints in a readable and type-safe way. While regular bounded polymorphism interacts well with the class hierarchy, F-bounded polymorphism does not.

### 4.2 Even more flexible typing with Eiffel

The programming language Eiffel [Mey92] was designed in the mid-80’s by Bertrand Meyer. It is a statically typed, class-based, object-oriented language...
4.2 Even more flexible typing with Eiffel

that supports multiple inheritance, bounded polymorphism, and run-time testable assertions. Like Java, all objects are implicit references and garbage collection is built in.

The definition of a sample Eiffel class, RATIONAL, is given in Figures 4.3 and 4.4. It represents rational numbers with methods to set, add, and compare rationals. Comments in Eiffel begin with "--" and continue to the end of the line.

This example illustrates several important features of Eiffel. First, it is possible for a class to inherit from multiple classes. Methods in inherited classes can be renamed either for convenience or to resolve name clashes between classes inherited from. Methods that are overridden must be declared to be "redefined". This is required to avoid the problem of having a programmer accidentally override a method in a superclass. It is also possible to change the visibility of inherited methods with a declaration in the inherit clause, though we don’t show it here.

Methods listed in the creation section of the class header can be used as constructors. CreateRat is the only creation routine in RATIONAL. If r has type RATIONAL, then !r.CreateRat will create a new object of type RATIONAL. Instance variables not explicitly initialized are given a default value (e.g., 0 or void, which corresponds to null in Java).

Classes may be parameterized by class (type) variables. The inherited parameterized class ORDEREDPAIR has been specialized to type INTEGER.

Features include both instance variables and methods. A feature clause can be annotated to indicate the visibility of items listed. The default is ALL, the keyword NONE corresponds to Java’s private, while you can also list any collection of classes (e.g. “friends”) that you wish to be able to have access to particular features. Instance variables can only be made available as read-only outside of the methods of the class. From outside an object it is impossible to distinguish between instance variables and parameterless procedures.

An important part of the Eiffel philosophy is “design by contract”. This encourages the program to include assertions indicating what should be true at different points in the program.

In method set, a precondition is given using keyword require and a postcondition is provided using ensure. The class invariant at the bottom of the class is presented with keyword invariant. It is also possible to specify loop invariants and “variants”. A variant is an integer expression that is always positive, but that gets closer to zero after each iteration. It is used to help prove termination of the loop. The assertions can be labeled in order to give better error messages when an assertion fails. Determination
class RATIONAL
  inherit
    ORDEREDPAIR [INTEGER]
    rename x as n, y as d -- change feature names
    redefine same -- method same is redefined
  end
OTHERCLASS
  redefine ... end
creation
  CreateRat
feature{ALL} -- public features
  error: boolean;

CreateRat is -- create a rational
do
d := 1
end; -- Create

set(numer, denom : INTEGER) is -- set n and d
require -- precondition
denom /= 0
local
gcd : INTEGER
do
  n := numer;
d := denom;
  if d < 0 then
    n := -n;
d := -d
  end;
gcd := reduce; -- see method below
  n := n // gcd; -- ‘//' is integer division
d := d // gcd
ensure -- postcondition
d > 0
end; -- set

Figure 4.3  RATIONAL class in Eiffel, part 1.
4.2 Even more flexible typing with Eiffel

plus(other : like Current) : like Current is
   local
      sumnum, sumden : INTEGER;
   do
      sumnum := n*other.d + other.n*d;
      sumden := d*other.d;
      !!Result.Create;
      Result.set(sumnum,sumden)
   -- returns value of Result automatically
   end;  -- plus

same(other : like Current) : BOOLEAN is
   do -- override inherited method
      ...
   end;
   ...

feature {NONE} -- private method
   reduce : INTEGER is ...
   end;

invariant d /= 0

end -- RATIONAL

Figure 4.4 RATIONAL class in Eiffel, part 2.

of which kinds of assertions are checked at run time is made by setting a
compile-time flag. The default is that preconditions are always checked, but
others are not.

Current is used in Eiffel methods as the name of the object executing the
method. Thus it plays the same role as Java and C++’s this or Smalltalk’s
self. If id is a declared identifier, then like id can be used to stand for the
static type of id. Thus the method plus in RATIONAL uses like Current
as the type of the argument and result of the method. If plus is sent to an
object of (static) type RATIONAL, then its actual parameter must also be of
type RATIONAL, and the result will be of type RATIONAL.

If plus is inherited in a subclass, SUBRATIONAL of RATIONAL, then the meaning of like Current will change appropriately. That is, if plus is sent to an object of (static) type SUBRATIONAL, then its actual parameter must also be of type SUBRATIONAL, and the result will be of type SUBRATIONAL.

This is very useful and provides great flexibility in creating subclasses. For example, in Figures 4.5 and 4.6 definitions are given of two parameterized classes, LINKABLE[G] and a subclass BILINKABLE[G], that are used to create singly and doubly linked nodes. The type parameter G represents the type (class) of the element held in the item field. The classes are slightly simplified versions of classes from the Eiffel structure library.

The type of instance variable right and the parameter of putRight of class LINKABLE[G] are declared to be of type like Current, which means they must be of the same type as Current. Using like Current in method signatures is very useful because the meaning of like Current will be different in a subclass from its meaning in the superclass.

Because of the use of like Current in the superclass, elements generated from class BILINKABLE will have both right and left fields holding values of type BILINKABLE. Similarly, the methods putLeft and putRight of an object generated from BILINKABLE will take parameters of type BILINKABLE.

This provides a great deal more flexibility than found in languages with the invariant type discipline. We have seen in Section 3.2.2 how frustrating it is not to be able to write a doubly linked node as a subclass of a singly linked node in languages like Java and C++, so this seems to be a great aid to expressiveness.

But now suppose we write the following routine:

```pascal
trouble(p, q : LINKABLE [RATIONAL] ) is
do
  p.putRight(q);
  ....
end
```

Suppose we have declared variables sNode : LINKABLE [RATIONAL] and biNode : BILINKABLE [RATIONAL], and we write:

```pascal
trouble(biNode, sNode)
```

As BILINKABLE[RATIONAL] is a subtype of LINKABLE[RATIONAL] in
4.2 Even more flexible typing with Eiffel

class LINKABLE [G]

feature{ALL}

    item: G;           -- value held
    right: like Current; -- Right neighbor

    putRight (other: like Current) is
        -- Put ‘other’ to the right of current cell.
        do
            right := other
            ensure
                chained: right = other
        end;
    end -- class LINKABLE

class BILINKABLE [G] inherit

    LINKABLE [G]
        redefine
        putRight
    end

feature{ALL} -- Access

    left: like Current; -- Left neighbor

    putRight (other: like Current) is
        -- Put ‘other’ to the right of current cell.
        do
            right := other;
            if (other /= Void) then
                other.simplePutLeft (Current)
            end
        end;
    end;
putLeft (other: like Current) is
  -- Put ‘other’ to the left of current cell.
do
  left := other;
  if (other /= Void) then
    other.simplePutRight (Current)
  end
ensure
  chained: left = other
end;

feature {BILINKABLE}
  simplePutRight (other: like Current) is
    -- set ‘right’ to ‘other’
do
    right := other
  end;

  simplePutLeft (other: like Current) is
    -- set ‘left’ to ‘other’
do
    left := other
  end;

invariant
  rightSymmetry:
    (right /= Void) implies (right.left = Current);
  leftSymmetry:
    (left /= Void) implies (left.right = Current)
end -- class BILINKABLE

Figure 4.6  Eiffel classes LINKABLE and BILINKABLE, part 2.
4.2 Even more flexible typing with Eiffel

Eiffel, this should work. Instead, the program crashes when the code for `putRight` in `BILINKABLE` attempts to send the message `simplePutLeft` to `sNode`, because `sNode` has no such method. *(The reader should trace the execution of this code to see exactly how things go wrong.)*

This example shows that implementations of Eiffel are not statically type-safe. In fact, Eiffel allows a number of statically unsafe changes in creating subclasses. One is allowed to replace the type of an instance variable, method parameter, or method result type by a subtype of the one occurring in the superclass. We will see in Chapter 6 that only method result types can be allowed to change in a covariant way if one wishes to guarantee type safety. Eiffel also allows programmers to hide inherited methods in a subclass that were visible in the superclass, creating yet another type insecurity.

This type insecurity has been known since the late 1980’s. Bertrand Meyer, the designer of Eiffel, has made various responses to these problems over time. He has consistently claimed that the insecurity is inconsequential because Eiffel users do not write code that would result in this error. That is, while they use features like these, they automatically use them correctly. He also claimed that proposed fixes like eliminating `like` declarations and instead using bounded polymorphism forced programmers to “plan ahead” too much and would result in programmers having to go back and rewrite superclasses.

After several years of criticism of Eiffel’s type problems, Meyer proposed a “system validity check” to ensure that these errors did not occur at run time. Essentially the idea was to perform a conservative “dataflow” analysis at link-time to make sure no unfortunate combinations of types would arise at run time due to interactions of the various classes. None of the Eiffel compiler suppliers ever implemented this check. It is not known why they failed to implement it, but it is likely that either the test was too expensive to run or it ruled out too many useful programs.

Even if this test had worked, programs would have been quite fragile with respect to the test. A program could pass the system validity check, but then the addition of a new class could cause the system validity check to fail due to an unexpected interaction that was only barely related to the new class.

More recently, Meyer proposed an alternative “fix” to Eiffel’s static type safety problems called “No polymorphic catcalls”. A call is polymorphic if its target can represent objects of more than one type. According to Meyer:

“A routine is a CAT (Changing Availability or Type) if some redefinition changes its export status or the type of one of its arguments. A call
is a catcall if some redefinition of the routine would make it invalid because of a change of export status or argument type.”

His proposal is to ban “catcalls” to expressions that can represent objects of more than one type. Essentially this requires that if the type of a method parameter is changed in a subclass, then you must know the exact type of any object that is a target of the corresponding message.

While a few type-safety problems were found in the original proposal, they have since been fixed. However, it is still not known if implementing this proposal would result in a statically type-safe language. Unfortunately, the proposal seems so restrictive that programmers will not be able to write useful programs that both use the dangerous features and pass this test. At the time this was written, no Eiffel compiler implemented the proposal.

In spite of its static typing problems, Eiffel was quite an advance over existing object-oriented languages. Its “design-by-contract” philosophy, which encourages the use of preconditions and postconditions, may be its most lasting legacy. Like Smalltalk, which is dynamically typed, it also showed how useful certain constructs could be that were not available in other languages.

Eiffel was also one of the first commercially available compilers for object-oriented languages that supported bounded polymorphism. Figure 4.7 includes Eiffel code defining classes similar to those given for GJ in the previous section.

INTORD inherits from the deferred class COMPARING. Deferred classes are like abstract classes in C++ and Java. They need not include code for all of the methods they declare. Classes in which all features are deferred are similar to interfaces in Java. Notice how the use of like Current solves the problems we had before that led us to the use of F-bounded polymorphism in GJ.

We can also use COMPARING as an upper bound for polymorphic classes:

class Sorting[T -> COMPARING]
feature
  sort (thearray: ARRAY[T]): ARRAY[T] is
    local ....
    do
      ..... 
      ....
      if thearray.item(i).lessThan(thearray.item(j))
      ..... 
    end;
end -- Sorting
deferred class COMPARING -- like abstract in Java
feature specification
  lessThan (other: like Current): BOOLEAN is
defered;
  greaterThan (other: like Current): BOOLEAN is
defered;
end -- class COMPARING

class INTORD inherit COMPARING
feature
  value:INTEGER;
  lessThan (other:like Current) is
do
    Result := value < other.value
  end;
...
end -- class INTORD

Figure 4.7  Eiffel version of COMPARING.

The notation "T -> COMPARING" indicates that the type variable T must be a subclass of COMPARING, which means that it must provide the comparison methods that take parameters of type like Current. The fact that COMPARING uses like Current allows the programmer to use regular bounded polymorphism rather than F-bounded polymorphism.

In summary, while Eiffel's static type system does not catch all type errors, and thus is not statically type safe, it does suggest constructs that would be quite useful in other statically typed object-oriented languages. In particular, the use of like Current seems to add a great deal of expressiveness to the language. In Chapter 16 we investigate the possibility of adding a similar MyType construct in a type-safe way.

4.3  Summary

In this chapter we investigated two languages, GJ and Eiffel, each of which offered extensions that increased the expressiveness of the language. Both
languages support parametric polymorphism, a feature that appears to be extremely important in designing secure generic data structures. GJ goes further in providing F-bounded polymorphism to support “binary methods” like comparison operators. GJ is statically typed and type-safe.

Eiffel uses the “like” construct to describe types. Among other things, this allows the programmer to declare that a type is like Current, that is, it is of the same type as the receiver of the message. This construct makes it much easier to define subclasses of classes with binary methods so that these methods have the desired types. Because of the availability of like Current, Eiffel also allows the programmer to use ordinary bounded polymorphism to provide the same constraints that require F-bounded polymorphism in GJ.

Unfortunately, this added flexibility in Eiffel results in a statically typed language that is not type-safe (at least in current implementations). However, the constructs introduced in Eiffel seem to be very useful in the cases that require F-bounded polymorphism in GJ. In Chapter 16, we will discuss the addition of type-safe variants of Eiffel’s like construct.
Thus far we have assumed that only object types have subtypes, and that subtypes are formed only by adding new methods to object types. In this chapter we provide some insight into ways that subtyping can be extended to more types, and how the subtyping relation on object types can be made richer.

Recall from Chapter 2 that type $S$ is a subtype of a type $T$, written $S <: T$, if an expression of type $S$ can be used in any context that expects an element of type $T$. Another way of putting this is that any expression of type $S$ can masquerade as an expression of type $T$.

This definition can be made more concrete by introducing a rule stating that if $S <: T$ and expression $e$ has type $S$, then $e$ also has type $T$. This rule, usually termed the subsumption rule, provides a mechanism for informing the type checker that an expression of a subtype can masquerade as an element of a supertype.

Subtyping provides added flexibility in constructing legal expressions of a language. Let $x$ be a variable holding values of type $T$. If $e$ is an expression of type $T$, then of course $x := e$ is a legal assignment statement.

Now suppose that $S$ is a subtype of $T$ and $e'$ has type $S$. Then $e'$ can masquerade as an element of type $T$, and hence $x := e'$ will also be a legal assignment statement. Similarly an actual parameter of type $S$ may be used in a function or procedure call when the corresponding formal parameter’s type is declared to be $T$.

In most pure object-oriented languages, objects are represented as implicit references. Assignment and parameter passing are interpreted as binding new names to existing objects, i.e., as ways of creating sharing. Because elements of a supertype and subtype both take the same amount of space (the space to hold a reference) there is no implementation difficulty in using el-
elements of the subtype in contexts expecting elements of the supertype. The
difficulty instead is determining when using an element of another type is
logically correct, i.e., all operations expected for the supertype make sense
for the subtype.

How can we determine when one type is a subtype of another? A careful
theoretical analysis of this topic would take us far afield from the aims of
this text into complex issues of domain theory in denotational semantics.
Instead we will present intuitive arguments for determining when one type
is a subtype of another. The subtyping rules in this section are based on those
given by Cardelli [Car88].

5.1 Subtyping for non-object types

Because object types have similarities to records of functions, we begin with
examining the simpler cases of subtyping for record and function types,
holding off on object types until later in this chapter. We also include a dis-
cussion of references (i.e., the types of variables in programming languages)
here, in order to prepare for the later discussion of instance variables in ob-
jets. This will also be useful in discussing subtyping for arrays and mutable
records.

5.1.1 Record types

In order to keep the initial discussion as simple as possible, we deal in this
subsection only with immutable (or “read-only”) records of the sort found
in functional programming languages like ML. While one can create records
in a single operation, the only operations that may be applied to existing
immutable record values are to extract the values of particular fields. No
operations are available to update particular fields of these records. Because
the operations do not depend on the order of the fields, we consider record
types that differ only in the order of their fields as identical.

An object can be interpreted as a record whose fields include their meth-
ods. Because methods may not be updated in objects, the study of immutable
records will be important to our understanding of object types. We discuss
in Section 5.1.3 the impact of allowing updatable fields.

Records associate values to labels so that the values may be extracted using
the name of the label. The type of a record specifies the type of the value
corresponding to each label. For example, we can define the record type
SandwichType = \{ \text{bread: BreadType; filling: FoodType} \}.

An example of an element of type SandwichType is

\[
\begin{align*}
s: \text{Sandwich} & := \{ \text{bread: BreadType := rye; filling: FoodType := pastrami} \} \\
\end{align*}
\]

Because these records are immutable, the only operations available on \( s \) are the extraction of values held in the \text{bread} and \text{filling} fields via expressions \( s.\text{bread} \) and \( s.\text{filling} \).

Suppose that we are given that CheeseType <: FoodType. Let

\[
\begin{align*}
\text{CheeseSandwichType} & = \{ \text{bread: BreadType; filling: CheeseType; sauce: SauceType} \} \\
\end{align*}
\]

and

\[
\begin{align*}
\text{cs: CheeseSandwich} & := \{ \text{bread: BreadType := white; filling: CheeseType := cheddar; sauce: SauceType := mustard} \} \\
\end{align*}
\]

We claim that CheeseSandwichType <: SandwichType.

For elements of CheeseSandwichType to successfully masquerade as elements of SandwichType, expressions of type CheeseSandwichType need to support all of the operations applicable to expressions of type SandwichType. Since the only operation available on these records is extracting fields, it is straightforward to show this.

A record \( cs \) of type CheeseSandwichType has the \text{bread} and \text{filling} fields expected of a value of type SandwichType. Moreover, the results of extracting the \text{bread} field from values of each of the two sandwich types each have type BreadType. The result of extracting the \text{filling} field from a record of type CheeseSandwichType is of type CheeseType, which is not the same as FoodType. However, because CheeseType <: FoodType, it can masquerade as a value of type FoodType.

Thus no matter which label from FoodType is extracted from a value of CheeseSandwichType, the result can masquerade as the corresponding type of SandwichType. Hence CheeseSandwichType is a subtype of SandwichType. The extra fields in CheeseSandwichType are irrelevant as we only need to know that enough fields of the appropriate types are available in order to masquerade as a Sandwich type.

Figure 5.1 illustrates a slightly more abstract version of this argument. In that figure a record \( x' : \{ m: S'; n: T'; p: U'; q: V' \} \) is masquerading
as a record of type \( \{ m: S; n: T; p: U \} \). We illustrate this by placing the figure representing \( r' \) inside a box (think costume) which has the same interface as an element of type \( \{ m: S; n: T; p: U \} \).

For the masquerade to be successful, the value of the \( m \) field of \( r' \), for example, must be able to masquerade as a value of type \( S \). Similarly for the \( n \) and \( p \) fields. Again, notice that the subtype may have more labeled fields (e.g., the \( q \) field) than the supertype, since the extra fields don’t get in the way of any of the operations applicable to the supertype.

Thus one record type is a subtype of another if the first has all of the fields of the second (and perhaps more), and the types of the corresponding fields are subtypes. Notice that the ordering of the fields is irrelevant in determining subtyping. We identify record types that are the same up to the ordering of fields.

We write this more formally as follows. Let \( \{ l_i: T_i \}_{1 \leq i \leq n} \) represent the type of a record with labels \( l_i \) of type \( T_i \) for \( 1 \leq i \leq n \). Then,

\[
\{ l_i: T_i \}_{1 \leq i \leq n} \subset \{ l_i: U_i \}_{1 \leq i \leq k}, \text{ if } k \leq n \text{ and for all } 1 \leq i \leq k, T_i \subset U_i.
\]

By this definition, CheeseSandwichType \( \subset \) SandwichType.

It is sometimes convenient to break up the subtyping for records into two pieces: breadth and depth subtyping rules. One record type is a breadth subtype of another if the first has all of the fields of the second (and perhaps more). A record type is a depth subtype of another if they have exactly the same fields, but the types of the corresponding fields are subtypes.

Again, the general subtyping rule above is appropriate for record values in which the only operations available are extracting labeled fields. Later we
5.1 Subtyping for non-object types

5.1.2 Function types

The proper definition of subtyping for function types has provoked great controversy and confusion, so it is worth a careful look. As discussed earlier, we write \( S \rightarrow T \) for the type of functions that take a parameter of type \( S \) and return a result of type \( T \). If \( (S' \rightarrow T') <: (S \rightarrow T) \), then we should be able to use an element of the first functional type in any context in which an element of the second type would type check.

Suppose we have a function \( f \) with type \( S \rightarrow T \). In order to use a function, \( f' \), with type \( S' \rightarrow T' \), in place of \( f \), the function \( f' \) must be able to accept an argument of type \( S \) and return a value of type \( T \). See Figure 5.2.

To masquerade successfully as a function of type \( S \rightarrow T \), function \( f' \) must be able to be applied to an argument, \( s \), of type \( S \). Because the domain of \( f' \) is \( S' \), it can be applied to elements of type \( S \) as long as \( S <: S' \). In that case, using subsumption, \( s \) can be treated as an element of type \( S' \), making \( f'(s) \) type-correct.

On the other hand, if the output of \( f' \) has type \( T' \), then \( T' <: T \) will guarantee that the output of \( f' \) can be treated as an element of type \( T \). Summarizing,

\[
(S' \rightarrow T') <: (S \rightarrow T), \text{ if } S <: S' \text{ and } T' <: T
\]

If we assume, as before, that \( \text{CheeseType} <: \text{FoodType} \), it follows that...
(Integer → CheeseType) <: (Integer → FoodType)

but

(FoodType → Integer) <: (CheeseType → Integer)

In the latter case, if \( f' : \text{FoodType} \rightarrow \text{Integer} \), then \( f' \) can be applied to an expression of type CheeseType, since that expression can masquerade as being of type FoodType.

The reverse is not true, since if \( f : \text{CheeseType} \rightarrow \text{Integer} \), it may not be possible to apply \( f \) to an argument of type FoodType. The body of \( f \) may apply an operation that is only defined for expressions of type CheeseType. For example, suppose melt is a function that can be applied to elements of type CheeseType, but not FoodType. Then if melt is applied to the parameter in the body of \( f \), an execution error would arise if the actual parameter was of type FoodType and not CheeseType.

Procedure types may be subtyped as though they were degenerate function types that always return a default type Void.

The subtype ordering of parameter types in function subtyping is the reverse of what might initially have been expected, while the output types of functions are ordered in the expected way. We say that subtyping for parameter types is \textit{contravariant} (i.e., goes the opposite direction of the relation being proved), while the subtyping for result types of functions is \textit{covariant} (i.e., goes in the same direction).

The contravariance for parameter types can be initially confusing, because it is always permissible to replace an actual parameter by another whose type is a subtype of the original. However the key is that in the subtyping rule for function types, it is the function, \textit{not} the actual parameter, which is being replaced.

Let us look at one last example to illustrate why contravariance is appropriate for type changes in the parameter position of functions and procedures. The contravariant rule for procedures tells us that it is possible to replace a procedure, \( p \), of type CheeseType → Void by a procedure, \( p' \), of type FoodType → Void.

The procedure \( p \) can be applied to any value, \( \text{cheese} \), of type CheeseType. Because CheeseType <: FoodType, the value \( \text{cheese} \) can masquerade as an element of type FoodType. As a result, \( p' \) can also be applied to the value \( \text{cheese} \). Thus \( p' \), and indeed any procedure of type FoodType → Void, can masquerade as an element of type CheeseType → Void.
5.1 Subtyping for non-object types

5.1.3 Types of variables

Variables holding values of type \( T \) have very different properties than simple values of type \( T \). Variables holding values of type \( T \) may be the targets (left sides) of assignments, while values of type \( T \) may only be the sources (right sides) of such assignments. Obviously, an expression representing a value, e.g., 3, of type integer may not be a target of an assignment statement.

Thus we need to distinguish values of type \( T \) from variables holding values of type \( T \). Because variables are sometimes referred to as references, we will denote the type of variables holding values of type \( T \) as \( \text{Ref } T \). Thus a variable \( x \) holding integer values will have type \( \text{Ref Integer} \), while the number 17 has type \( \text{Integer} \).

Variables in programming languages typically represent two kinds of values. This can be seen by examining the meaning of the statement

\[
\text{x := x + 1}
\]

The \( x \) on the left side of the assignment represents a location in memory that can hold values, while the \( x \) on the right represents the value stored in that location. These values are sometimes referred to as the \( l\text{-value} \) and \( r\text{-value} \) of the variable. The \( l\text{-value} \) (so-called because it is used as the value of variables to the left of an assignment) represents the location of the variable, while the \( r\text{-value} \) (used for variables occurring on the right side of an assignment) represents the value stored in the variable.

To make this distinction clearer as we examine variables, we will use the notation \( \text{val } x \) to stand for the \( r\text{-value} \) of a variable \( x \), while an unqualified \( x \) will represent the \( l\text{-value} \) of the variable. Thus, we would re-write the above assignment as:

\[
\text{x := val } x + 1
\]

In the rest of this subsection we show that the variable (reference) types have only trivial subtypes. We begin as usual with an example.

Suppose once more that \( \text{CheeseType} <: \text{FoodType} \), \( \text{apple} \) is a value of type \( \text{FoodType} \), \( \text{fv} \) is a variable with type \( \text{Ref FoodType} \), and \( \text{cheddar} \) is a value of type \( \text{CheeseType} \). Then the assignment

\[
\text{fv := apple}
\]

is type-correct because \( \text{apple} \) has type \( \text{FoodType} \). It follows that

\[
\text{fv := cheddar}
\]
is also type-correct, because we can always replace a value of type `FoodType` by a value of a subtype. That is, using `cheddar` in a slot expecting a value of type `FoodType` is safe because `CheeseType <: FoodType`.

Suppose `cv` is a variable with type `Ref CheeseType`. We noted above that `fv := apple` is fine, but replacing `fv` by `cv` in the assignment statement to obtain `cv := apple` results in a type error.

For example, suppose again that `melt` is a function that can be applied to cheeses but not general foods like apples. Thus an execution error would result if `melt` were applied to `cv` and it held a value, `apple`, that was not of type `CheeseType`.

Thus it is not type-correct to replace a variable holding values of a given type by a variable holding values of a subtype.

As suggested in the example above, the fact that variables may be the targets of assignments will have a great impact on the subtype properties (or rather the lack of them) of reference types. In particular, the example illustrates that `Ref CheeseType` cannot be a subtype of `Ref FoodType`, even though `CheeseType <: FoodType`.

Suppose we wish variable `x'` with type `Ref S'` to masquerade as a variable holding values of type `S`. See Figure 5.3 for a graphic illustration.

As indicated earlier, a variable `x` holding values of type `S` has two values: an l-value and an r-value, where the latter value is obtained by writing `val x`. Thus two operations are applicable to variables, assignment statements with the variable on the left, and `val` expressions. In the first of these, the variable occurs in a value-receiving context, while in the second it occurs in a value-supplying context.

The second of the two operations is represented in the figure by the arrow labeled "`val`" coming out of the variable (because it supplies a value). If `x` is a variable with type `Ref S`, then `val x` returns a value of type `S`.

![Figure 5.3: A variable `x:Ref S`, and another variable `x':Ref S'` masquerading as having type `Ref S`.](image_url)
5.1 Subtyping for non-object types

For a variable \( x' \) holding values of type \( S' \) to be able to masquerade as a value of type \( S \) in all contexts of this kind, we need \( S' <: S \). This should be clear from the right diagram in the figure, where in order for \( x' \) to provide a compatible value using the \( \text{val} \) operator, we need \( S' <: S \).

A value-receiving context is one in which a variable holding values of type \( S \) is the target of an assignment, e.g., a statement of the form \( x := e \), for \( e \) an expression of type \( S \). This is represented in the figure by an arrow labeled \( " := " \) going into the variable.

In this context we will be interpreting the variable as a reference or location (i.e., the \( l \)-value) in which to store a value. We have already seen that an assignment \( x := e \) is type safe if the type \( S \) of \( e \) is a subtype of the type declared to be held in the variable \( x \). Thus if we wish to use a variable holding values of type \( S' \) in all contexts where the right side of the assignment is a value of type \( S \), we must ensure that \( S' <: S \). Again this should be clear from the right diagram in the figure.

Going back to the example at the beginning of this section, suppose we have an assignment statement,

\[
\text{cv} := \text{cheddar}
\]

for \( \text{cv} \) a variable holding values of type \( \text{CheeseType} \) and \( \text{cheddar} \) a value of type \( \text{CheeseType} \). If \( \text{fv} \) is a variable holding values of type \( \text{FoodType} \), then we can insert \( \text{fv} \) in place of \( \text{cv} \) in the assignment statement, obtaining

\[
\text{fv} := \text{cheddar}
\]

Because \( \text{CheeseType} <: \text{FoodType} \), this assignment is legal. However the assignment \( \text{cv} := \text{apple} \) would not be legal.

Thus for a variable holding values of type \( S' \) to masquerade as a variable holding values of type \( S \) in value-supplying (\( r \)-value) contexts we must have

\[
S' <: S
\]

while it can masquerade in value-receiving (\( l \)-value) contexts only if

\[
S <: S'
\]

It follows that there are no non-trivial\(^1\) subtypes of variable (reference) types.

Thus,

\[
\text{Ref } S' <: \text{Ref } S, \text{ if } S' \simeq S,
\]

\(^1\) A subtype is trivial if it is equivalent to the supertype in the sense that they are each subtypes of each other.
where $S' \simeq S$ abbreviates $S' <: S$ and $S <: S'$. We can think of $\simeq$ as defining an equivalence class of types including such things as pairs of record types that differ only in the order of fields. It is common to ignore the differences between such types and to consider them equivalent.

We can get a deeper understanding of the behavior of reference and function types under subtyping by considering the different roles played by suppliers and receivers of values. Any slot in a type expression that corresponds to a supplier of values must have subtyping behave covariantly (the same direction as the full type expression), while any slot corresponding to a receiver of values must have contravariant subtyping (the opposite direction).

Thus $l$-values of variables and parameters of functions, both of which are receivers of argument values, behave contravariantly with respect to subtyping. On the other hand, the $r$-values of variables and the results of functions, both of which are suppliers of values, behave covariantly. Because variables have both behaviors, any changes in type must be simultaneously contravariant and covariant. Hence subtypes of reference types must actually be equivalent.

5.1.4 Types of updatable records and arrays

The same analysis as for references can lead us to subtyping rules for updatable records and arrays. An updatable record should support operations of the form $r.I := e$, which results in a record whose $I$ field is $e$, while the values of the other fields are unchanged. The simplest way to model this with the constructs introduced so far is to represent an updatable record as an immutable record, each of whose fields represents a reference.2 Thus the fields represent locations whose values could be updated.

An updatable record with name and age fields would have type

\[
\text{PersonInfo} = \{\text{name: Ref String; age: Ref Integer}\}
\]

Thus if mother has type PersonType, then mother.name has type Ref String.

Combining the record and reference subtyping rules,

\[
\{l_j: \text{Ref } T_j\}_{1 \leq j \leq n} <: \{l_i: \text{Ref } U_i\}_{1 \leq i \leq k}, \text{ if } k \leq n \text{ and for all } 1 \leq i \leq k, T_i \simeq U_i.
\]

2. In a real implementation, the locations of the fields would be calculated from the location of the beginning of the record and the size of each field. However this difference has no impact on the subtyping rules.
Thus the subtype has at least the fields of the supertype, but, because the fields can be updated, corresponding fields must have equivalent types. Thus adding fields results in a subtype, but no changes to the types of existing fields is allowed.

Arrays behave analogously to functions. Let 

\[
\text{ROArray[IndexType]} \text{ of } T
\]

denote a read-only array of elements of type \(T\) with subscripts in \(\text{IndexType}\). This data type can be modeled by a function from \(\text{IndexType}\) to \(T\), as one can think of accessing an array element, \(A[i]\), as being similar to applying a function to that index and obtaining the value. As a result, the subtyping rules are similar to those of functions:

\[
\text{ROArray[IndexType']} \text{ of } S' \lessdot \text{ROArray[IndexType]} \text{ of } S, \\
\text{if } S' \lessdot S \text{ and IndexType } \lessdot \text{IndexType'}
\]

Intuitively, the index types of read-only arrays change contravariantly because, like function parameters, they are value receivers, while the types of elements of the arrays change covariantly because read-only arrays supply values of those types, just like function return types.

Of course, arrays in most programming languages allow individual components to be updated. We can model \(\text{Array[IndexType]} \text{ of } T\) by a function from \(\text{IndexType}\) to \(\text{Ref } T\). From function and reference subtyping rules it follows that

\[
\text{Array[IndexType']} \text{ of } S' \lessdot \text{Array[IndexType]} \text{ of } S \\
\text{if } S' \simeq S \text{ and IndexType } \lessdot \text{IndexType'}
\]

As before, the index types of arrays change contravariantly, but now the types of elements of the arrays are invariant because arrays both supply and receive values of those types.

Java’s [AGH99] type rules for array types are not statically type-safe. In Java the type of an array holding elements of type \(T\) is written \(T[]\). The subtyping rule for array types in Java is

\[
S'[] \lessdot S[], \text{ if } S' \lessdot S.
\]

The following Java class will illustrate the problems with this typing rule. Suppose \(C\) is a class with a subclass \(CSub\), and suppose the method \(\text{methodOfCSubOnly()}\) is defined in class \(CSub\), but was not available in \(C\). Now define the class \(\text{BreakJava}\) below:

\[\text{3. Java array types do not mention the type of subscripts because they are always integers.}\]
The first two lines of the `main` method construct a new instance of the class `BreakJava` and create an array of `CSub` with 10 elements. The message send of `arrayProb` to `bj` at (1) will result in a type error.

The problem is that `paramArray`, an array of elements of type `CSub`, is passed to method `arrayProb` where an array of type `C` was expected. Because of this, the assignment in line (2) of `arrayProb` will result in the assignment of a value `v` from class `C` into an array which is supposed to hold values of type `CSub`. We know that it is illegal to assign a value of a superclass into a variable holding values of the subclass. In fact, if allowed this would result in a run-time error in line (3), which would be executed immediately after the method `arrayProb` finishes executing. Because the value `v` of class `C` was assigned to `paramArray[0]`, the message send of `methodOfCSubOnly()` would fail as elements of class `C` do not support that method.

The correct rule for arrays specified above implies that the message send in line (1) would result in a static type error because `CSub[]` fails to be a subtype of `C[]`.

While the Java designers used an incorrect rule for static checks of subtyping with arrays, they compensated for this by inserting extra dynamic checks. Thus Java would not indicate any type errors at compile time, but it would insert a dynamic check at line (2) because of the assignment to an array parameter. That dynamic check would fail during the execution of the message send `bj.arrayProb(paramArray)` from line (1). The message send at line (3) would never be reached at run time because an exception
would have been raised due to the failure of the dynamic check at line (2).

Thus the Java designers compensate for not catching the type error statically by performing dynamic checks when an individual component of an array is assigned to. Why did they use this obviously faulty subtyping rule, when it results in having to add extra code to assignments to array parameters? While it is necessary for type safety, this extra code in compiled programs is problematic as it results both in increased size of programs and a slowdown in their execution.

One reason the Java designers might have included this faulty rule would be to allow generic sorts (and similar operations) to be written that could pass the static type checker. Java programmers can write sort methods that take elements of type Comparable[], where Comparable is an interface supporting a method compareTo that returns a negative, zero, or positive int depending on whether the receiver is smaller than, equal to, or larger than the parameter. Java’s unsafe subtyping rule for arrays allows any array of elements that implement Comparable to be passed to such sort methods, even though they are in theory vulnerable to the same errors as illustrated above.

However, the actual code written in these sort routines typically does not create a dynamic type error because it simply reorders elements of the array, rather than assigning brand new values. Thus one result of the decision to give up static type safety by including an “incorrect” subtyping rule for arrays is to make it easier for programmers to write more flexible programs.4

As we saw in Section 4.1, parametric polymorphism of the sort introduced in GJ would allow the creation of type-correct generic sorts without the need for this unsafe rule. Thus we can recapture static type safety and maintain expressiveness of the language by introducing a richer type system. We will see other examples of this trade-off in later chapters.

5.2 Object types

While most popular object-oriented languages determine subtyping of object types based on whether the corresponding classes are subclasses, this identification of subclass with subtype is not necessary. In this section we determine subtyping rules for objects that depend only on their public interfaces or object types.

4. The reason why this subtyping rule for arrays was included is apparently not as principled. An implementation hack for arrays resulted in a desire for this subtyping rule [Joy98].
The subtyping rules for object types follow from those of records and functions. From the outside, the only operation available on objects is message sending. As a result, object types behave like immutable records. The subtyping rule is:

\[
\text{ObjectType} \left\{ \| l_i : \mathbb{S}_i \|_{1 \leq i \leq k}, \right\} \left[ 1 \leq j \leq n \right] \leq \text{ObjectType} \left\{ \| l_i : \mathbb{S}_i \|_{1 \leq i \leq k}, \right\} \text{ if } k \leq n \text{ and for all } 1 \leq i \leq k, \mathbb{S}_i' \leq \mathbb{S}_i.
\]

Because the types \( \mathbb{S}_i' \) and \( \mathbb{S}_i \) are method types, they are functional types. Suppose \( \mathbb{S}_i' = T_i' \rightarrow U_i' \) and \( \mathbb{S}_i = T_i \rightarrow U_i \). Then, by the subtyping rule for function types, \( \mathbb{S}_i' \leq \mathbb{S}_i \) if both \( T_i \leq T_i' \) and \( U_i' \leq U_i \). That is, object types are subtypes if for every method in the supertype there is a method with the same name in the subtype such that the range types of corresponding methods vary covariantly, and the domain types vary contravariantly.

What is the relation between subclasses and subtypes? Most popular statically typed object-oriented languages allow no changes to method types in subclasses. This clearly implies that the object types generated by a subclass-superclass pair are in the subtype relation. We noted earlier that C++ allows covariant changes to result types in subclasses. By the above, this also results in subtypes.

As we saw in Section 4.2, Eiffel [Mey92] allows covariant changes to both parameter and result types of methods in subclasses. We exhibited an example there showing that this was not type-safe. The subtyping rule given here for object types explains this failure by making it clear that covariant changes to parameter types are not statically type safe. The language Sather [Omo91] allows contravariant changes to parameter types and covariant changes to return types in subclasses. Thus it is the most flexible in allowing changes to subclasses so that the resulting object types are in the subtype relation.

While our focus in this section has been on subtyping, a related interesting question is what, if any, restrictions must be placed on changing types of methods in subclasses, even if we don’t care whether subclasses generate subtypes. We examine that question in Chapter 6.

### 5.3 Subtyping for class types

We haven’t yet introduced the notion of class types as ways of categorizing classes (just as object types categorize objects). We will do that carefully later. However, it is evident that a class type should include information on the types of instance variables and methods. The reason is that to determine
5.3 Subtyping for class types

whether we can extend a class with new methods or instance variables, we
need to know what methods and instance variables already exist there. If a
type of a class is to give us sufficient information about a class to determine
whether or not a particular extension is legal, it will need to include that
information about methods and instance variables.

Let us use the notation ClassType(IV, M) for the type of a class whose
instance variables have names and types given by the labels and types of
record type IV, and whose methods have names and types given by the
record type M. If all instance variables are invisible from outside of an ob-
ject generated by a class, then the type of objects generated from a class with
type ClassType(IV, M) will be ObjectType M.

Because most object-oriented languages use class names as types, this no-
tation for class types may look unusual to the reader. We emphasize again
that objects have types of the form ObjectType M, while classes will now
have types of the form ClassType(IV, M).

We can ask whether one class type can be a subtype of another. As before,
to determine whether one class type can be a subtype of another we must
consider what operations are available on classes. There are only two: creat-
ing new objects and extending classes to form subclasses. We will see that in
our system there can be no non-trivial subtypes of class types exist, because
of the difficulty of masquerading in both of these contexts.

Suppose class C' of type ClassType(IV', M') is attempting to masquerade
as having type ClassType(IV, M). Let us see what constraints on IV', IV,
M', and M follow from this assumption.

Evaluating new C' will generate an object of type ObjectType M'. If C'
is to successfully masquerade as an element of type ClassType(IV, M) then
the type of the expression new C', ObjectType M', must be a subtype of
ObjectType M. Thus we need M' <: M.

Suppose a subclass SC is defined by inheritance from C:

class SC inherits C modifies l_i, ..., l_n {...}

so that SC is well-typed with type ClassType(IV_{sub}, M_{sub}) when C has type
ClassType(IV, M). If the type of C' is a subtype of ClassType(IV, M), then
SC should be well-typed if C is replaced by the masquerading C'. However,
any method m of C could have been overridden in SC with a method of the
same type. Because this override must still be legal in the subclass built from
C', all methods in M must have the same type in M' (as otherwise the override
would have been illegal).

Similarly, M' could have no more methods than M. If M' had an extra
method, \( m' \), we could define a subclass of \( C \) with an added method \( m' \) with an incompatible type from that of \( m' \) in \( M' \). If we attempt to define a similar subclass from \( C' \), we would get a type error in defining the subclass (assuming that there are any restrictions at all on changing types in subclasses).

Thus if \( \text{ClassType}(IV',M') <: \text{ClassType}(IV,M) \) then we must have \( M' \simeq M \). Similar arguments on instance variables can be used to show that \( IV' \simeq IV \). Thus there can be no non-trivial subtypes of class types:

\[
\text{ClassType}(IV',M') <: \text{ClassType}(IV,M), \text{if } IV' \simeq IV \text{ and } M' \simeq M
\]

The language discussed so far in this text has no access qualifiers like Java and C++’s private, protected, and public. We will discuss these qualifiers in Section 14.4, where we introduce the names secret, hidden, and visible for access qualifiers whose meanings are similar to Java and C++’s. Secret features are not visible outside of the class. That is, they are not visible to subclasses or other objects. Our default for instance variables is that they are hidden. This means that they are accessible to subclasses, but not to other objects. In Section 14.4 we assume that class types should not mention secret features (i.e., Java’s private features). Thus two classes whose visible and hidden feature names and signatures are the same have the same class type. With this understanding of class types, the claim that there are no non-trivial subtypes for class types remains true.

### 5.4 Summary

In this chapter, we provided a relatively careful, though informal, analysis of subtyping. The subtyping rules for immutable record types included both breadth and depth subtyping. That is, a subtype of a record type could include extra labeled fields (breadth) or could replace the type of one of the existing labeled fields by a subtype (depth subtyping).

We addressed the issue of covariance versus contravariance changes in creating subtypes of function types. We discovered that to avoid problems, only covariant changes were allowed to return types and only contravariant changes were allowed to domain types in subtyping function types. Most languages allow no changes to either domain or range types in subtyping function types, though some allow covariant changes in range types. There do not seem to be compelling examples where contravariant changes in domain types are useful.

We emphasize that the rules provided above can be proved mathematically to be safe. Languages that allow covariant changes to both range and
domain types (like Eiffel) are not statically type-safe. They either sacrifice type safety altogether or require link or run-time checks to regain type safety.

Reference types (types of variables) allowed no subtyping because elements of these types can both be used as sources of values (e.g., using the `val` construct) and as receivers of values in assignment statements. Subtyping for mutable records and arrays followed naturally from the rules for immutable records, functions, and references. Mutable records allow only breadth subtyping, while arrays only allow contravariant changes to the index types.

Subtyping for object types followed naturally from the rules for immutable records and functions. A subtype of an object type can add new methods (breadth subtyping again) or replace the type of an existing method with a subtype (depth subtyping). By the subtyping rules on function types, one may make contravariant changes to the domain type of the method and covariant changes to the return type. Because instance variables (or hidden methods) do not show up in the public interface of objects, they have no impact on subtyping.

There is no non-trivial subtyping for class types, because of the possibility of conflicts in extending class definitions using inheritance.

We summarize the subtyping rules discussed in this chapter in Figure 5.4. For simplicity we presume that there are no subtype relations involving type constants. (That is, we do not allow `Integer <: Real`, for example.)

We have also generalized the subtyping rule for function types to include functions with more than one argument. The domain of a function with multiple arguments is represented as a product or tuple type.

In the next chapter we address the impact of our rules for subtyping on the allowed changes to types of methods and instance variables in defining subclasses.
Figure 5.4 Summary of subtyping rules.
There are many circumstances under which we would like to change the types of methods in subclasses. What is it that keeps us from making arbitrary changes to the types of these methods?

In the last chapter we saw that we can create subtypes of object types by adding new methods or by replacing the types of existing methods by subtypes. *A priori,* there seems to be no reason for following these restrictions in creating subclasses if we don’t care whether the resulting object type is a subtype of that generated by the superclass. Are any restrictions necessary on changing types of methods or instance variables in subclasses if all we care about is that all of the methods in the subclass are type-safe?

Eiffel allows covariant changes to method return types, method parameter types, and the types of instance variables in subclasses. We saw in Section 4.2 that covariant changes to parameter types break subtyping, but perhaps it is all right to allow these changes if we don’t expect the resulting types to be in the subtype relation.

We will see in this chapter that the mutually recursive nature of the methods in classes requires that we do be careful in what changes are allowed to types of instance variables and methods. Interestingly, the restrictions will be similar to those required to preserve subtyping of object types.

### 6.1 Allowable changes to method types

The main reason for needing restrictions on changes to the types of methods in subclasses is that a method body in a class may call any other method from the class by sending a message to `self`. Thus, changes to one method may have an impact on both the typing and meaning of another method in the
class C {
    function m(s:S): T is { .... }
    function n(anS:S): U is {
        ... self ⇥ m(anS) ...
    }
}

class SC inherits C modifies m {
    function m(s:S'): T' is // For which S', T' safe?
    { ... }
}

Figure 6.1 Changing types of methods in subclasses.

same class. In this section we discuss the changes to method types that are
guaranteed not to cause typing problems.

Figure 6.1 shows a well-typed class containing methods m and n, where m
has type S → T. The body of method n includes a message send of m(anS)
to self. We presume that the body of method n will be well-typed if m has
type S → T. In particular, we assume that the context in which the message
send occurs needs the result to be a value of type T to be type correct.

Suppose we override m in a subclass so that it has type S' → T'. In general,
the only way to be sure that the occurrence of self ⇥ m in the body of
n is still compatible is to require that S' → T' ≤ S → T (this is, after all,
exactly what the subtype relation guarantees).

Thus we can guarantee type safety when forming subclasses if we restrict
ourselves to overriding a method with a new one whose type is a subtype of
the type of the original.

Of course it is also technically possible to keep track of which methods
are called by other methods and annotate each class with that information.
While this would allow more freedom in making changes to some methods,
this seems too painful for regular use and may require a data flow analysis
of even indirect uses of an object. Hence we state this simpler requirement.

Because methods must be functions or procedures, we can indicate more
exactly the changes allowed to types of methods in subclasses. Recall from
the previous chapter that subtyping of function types is contravariant in
the parameter type and covariant in the result type. Thus, in overriding a
method in a subclass, we may replace a result type T by a subtype T' and a
6.2 Instance variable types invariant in subclasses

Unfortunately, we do not have the same flexibility in changing the types of instance variables as we do with methods. Recall from Section 5.1.3 that a variable whose declared type is $T$ is actually a value of type $\text{Ref } T$. As discussed there, reference types have no subtypes because they can be used in either value-receiving (on the left side of $\text{:=}$) or value-supplying positions. As a result, it is not possible to change the types of instance variables in subclasses.

A concrete example of this is given in the following example.

```
class Rect {
    ul: Point := nil;  // upper left corner
    lr: Point := nil;  // lower right corner

    function setUL(newUL: Point): Void is
        { self.ul := newUL }
}

class ColorRect {
    ul: ColorPoint := nil;
    lr: ColorPoint := nil;
    ...
}
```
Suppose the intention is that the redefinitions of ul and lr in ColorRect are to override the corresponding definitions in Rect by providing a different type for the instance variables. Unfortunately this change will break the type safety of the inherited method setUL as a value of type Point may not be assigned to a variable of type ColorPoint.

The inability to safely change the types of instance variables minimizes the advantages obtained by being able to make covariant changes to the return types of methods, as we saw in Chapter 3. Later we will expand on the ideas presented in Chapter 4 to increase the expressiveness of the language while retaining type safety.

6.3 Changing visibility

There is one kind of change that can be made to methods in subclasses that we have not yet considered. Rather than changing the types of methods, we can consider changing the visibility of methods.

We have not discussed this earlier because our defaults have been that all methods are visible. We will continue with those defaults until section 14.4, but since most object-oriented languages do allow the programmer to specify degrees of information hiding, we briefly address this question in terms of preserving safety in subclasses.

It is easy to see that taking methods that are hidden inside superclasses and making them more visible in subclasses causes no problems with inherited methods or even in having subclasses generate subtypes. What is more interesting is that, if we do not care whether subclasses generate subtypes, we can also make methods less visible in subclasses.

For the purposes of this discussion, we adopt the terms public, protected, and private from Java [AGH99]. A public method is visible within its class, within all subclasses, and to any object that wishes to send it messages. A protected method is visible within its class and within all subclasses, but objects not of the same class may not send a message that invokes a protected

1. If you were to do this in Java, it would instead define new instance variables with the same names as the old instance variables. In other words, references to ul in methods inherited from Rect would refer to the instance variables defined there, but references to ul in new methods or overridden methods in ColorRect would refer to the new instance variable. Changes to the old one would have no impact on values of the new one and vice versa. While one might imagine a programmer wanting such behavior, it certainly would be unusual. This is not what we are intending by our redeclaration above.
method. A private method is only visible within the class in which it appears. It is not visible in subclasses or to other objects.

Changing the visibility of a method from public to protected causes no difficulty with inherited methods because public and protected methods are equally visible to inherited methods of the class. Changing the visibility from protected to private also causes no problems. Inherited methods that called the method still have access because their definitions appeared in a class which had access to the method. Of course, if a method body is overridden in a later subclass, it will not be able to access the newly private method. However this does not break any existing code.

Of course, if one hides a previously public method, $m$, in a subclass $SC$ of $C$, then $SC$ no longer generates objects whose type is a subtype of the objects generated by $C$. Objects generated from $C$ can respond to message $m$, while those generated from $SC$ cannot.

Thus languages that allow method visibility to be restricted do not have problems with type safety of inherited methods. However, they need not generate subtypes.

6.4 Summary

In this chapter we learned that even if we don’t care whether or not subclasses generate subtypes, there still need to be restrictions on allowable changes in the types of methods and instance variables in subclasses, if we wish to retain type safety. This is the result of the mutually referential nature of methods and their references to instance variables.

We may only make covariant changes to the return types of methods and contravariant changes to the parameter types of methods. We may make no changes at all to the types of instance variables without raising the possibility of type errors. While allowing covariant changes to return types seems to provide some minor gains in expressiveness, allowing contravariant changes to parameter types doesn’t seem to be very useful. For exactly the same reasons as for records, adding new methods and instance variables in subclasses causes no difficulties.

It is interesting that the restrictions on changing method types were exactly those required for subclasses to generate subtypes. Later we will extend the language (using MyType) in such a way that the restrictions for creating type-safe subclasses are not sufficient to guarantee that subclasses generate subtypes.
Thus far we have restricted our attention to class-based object-oriented languages. In the next chapter we will finish our overview of object-oriented languages by taking a quick look at two other related kinds of languages, object-based and multi-method languages.
In this chapter we discuss two other styles of languages that are often discussed as object-oriented languages: object-based and multi-method languages. We also briefly discuss a few interesting features of existing object-oriented languages that distinguish them from the model languages discussed in this book. The goal is to highlight differences in languages that are grouped under the umbrella label of object-oriented languages.

7.1 Multi-methods vs. object-based vs. class-based languages

In this book we focus on class-based languages, but there are other alternatives. In this section we discuss briefly object-based and multi-method styles of languages.

7.1.1 Object-based languages

As discussed in Chapter 2, classes in class-based languages serve as extensible templates for the creation of new objects (though they often serve other purposes as well). Object-based languages do not support classes, and thus must provide other mechanisms to create new objects.

In most object-based languages, objects can be constructed directly. Moreover, functions can be defined that, when instantiated with appropriate parameters, return objects that all have the same shape. For example, one might write a function `makePoint`, taking parameters `nx` and `ny`, that, when invoked, returns a `Point` object whose instance variables are initialized to the values provided as parameters:
function makePoint(nx: Integer, ny: Integer): PointType is
{
    return object {
        x: Integer := nx;
        y: Integer := ny;
        function move(dx: Integer, dy: Integer): Void is
        {
            x := x + dx;
            y := y + dy;
        }
    }
}

A prototype-based language, a special kind of object-based language, allows
the user to generate “prototypical” objects, and then to use these as proto-
types to create new objects by cloning. Of course, these clones would not
be much more useful than objects generated by functions like makePoint
above if they did not provide a way of updating features of the cloned ob-
jects.

As usual, there is no difficulty in dynamically updating instance variables
in the generated objects. However, these languages typically also allow dy-
namic updates of methods. That is, if a prototype point contains a method
move, the programmer can replace the move method with a new method
with the same name. If the language is statically typed, then the new method
signature must typically be compatible with that of the original. Because
both instance variables and methods can be updated at run time, there is
little reason to distinguish between instance variables and methods.\(^1\) As a
result we shall use the term feature to refer to either of these.

While clones may update features dynamically, the resulting objects all
have the same structure. To add features, as is done with inheritance, a new
language mechanism must be provided. Features may be provided either
to allow the programmer to extend another object or to delegate operations to
another object.

Extension works much like inheritance in classes. One simply declares that
the new object o extends a prototype p and lists the new features. As a result
o contains separate copies of the features of p. Thus if p had a field x of type
Integer, o would also have a field x of type Integer. However, changes

\(^1\) Of course, there may be important distinctions in implementations. Instance variables typi-
cally don’t involve references to self, while methods may use self, resulting in a more com-
plex run-time semantics.
to the $x$ field of $p$ would have no impact on the $x$ field of $o$. This is exactly as one would expect in analogy to what happens when multiple objects are generated from a single class.

Delegation, on the other hand, allows an object to delegate certain operations to a surrogate object. For example, suppose again that a prototype $p$ has a field $x$ of type `Integer`. If a new object $o$ is defined by delegation from $p$, then any attempt to access the $x$ field of $o$ will actually obtain the value of the $x$ field of $p$. Also, any change to the $x$ field of $p$ will be visible from $o$ and vice versa.

Thus an object created using delegation retains ties to its prototype parent, while an object created by extension is independent of its parent.

There are advantages and disadvantages to object-based languages. An important argument for object-based languages is that they are conceptually simpler than class-based languages because they avoid having a special syntactic construct – the class – that is only used to generate new objects. Rather than defining a class and then generating objects from it, one can either directly define objects or build them from prototypes by cloning, delegating, or extending. Because of this unification of classes and objects, there is also little need to distinguish methods and fields in the language. Both can be updated at run time, while in a class-based object-oriented language, only instance variables can be updated at run time.

A counter-argument against object-based languages is that, except in the case where only one object of a particular structure is needed, prototypes are typically not used as objects. They are instead created and used like classes. Moreover, an argument can also be made that constructs like Java’s anonymous classes [AGH99, GJSB00] are nearly as lightweight (at least syntactically) as directly defining objects.

Moreover, the flexibility provided by allowing dynamic update of methods in objects comes with a price attached in semantic complexity. Because methods may be updated on the fly, it is difficult to reason about object-based languages, as objects with the same structure may have very different behavior. It is typically easy to take into account differences in values of instance variables when reasoning statically about collections of objects generated from the same class. However, when methods may change on the fly, the fact that methods may make calls to other methods through `self` makes it hard to predict what any of the methods will do.

This semantic complexity also shows up in the difficulty of modeling these languages. The translational semantics provided later in this book is not sufficient to cover the situation where methods may be updated dynamically.
The modeling is not robust enough to allow self to refer to different method bodies for the same method name over time.

Abadi and Cardelli [AC96] originally did not believe a translational semantics into the lambda calculus existed for such languages. As a result they developed an object calculus to model these languages. However, with Viswanathan [ACV96], they later developed a translational semantics that is somewhat more complex than the one given later in this monograph, but that correctly captures dynamic method update in objects.

The text, *A Theory of Objects* [AC96], by Abadi and Cardelli, is an excellent source of information on the foundations of object-based languages. See especially their Chapter 4 for a discussion of such languages, and the later chapters of the book for a discussion of their object calculus and the translational semantics. The book also discusses how to encode classes in an object-based language.

### 7.1.2 Multi-method languages

In class-based and object-based languages, the receiver of a message determines which method body is called. In *multi-method languages*, there is no receiver of a message. Method calls look like procedure calls in procedural languages. However, unlike in procedural languages, the selection of method body to be executed depends on the run-time types of the values of one or more of the parameters of the call.

For example, suppose we have a program in a multi-method language that involves both points and color points. The equal procedure that we use may depend on the types of both objects being compared:

```plaintext
function equal(p1: Point, p2: Point): Boolean is
{ return p1.x = p2.x & p1.y = p2.y }

function equal(p1: ColorPoint, p2: ColorPoint): Boolean is
{ return p1.x = p2.x & p1.y = p2.y
  & p1.color = p2.color }
```

Assume that ColorPoint is a subtype of Point and that pt1 and pt2 are identifiers declared statically to have type Point, but the values of both at run time are of type Point. If we evaluate equal(pt1, pt2) in a multi-method language, then the first of the two equal methods above would be chosen to execute because the choice is based on the run-time types of both parameters. On the other hand, if the values of pt1 and pt2 at run
time are both of type \texttt{ColorPoint} then the multi-method language would choose the second of the two methods to execute, because the parameters are a closer match to the second definition.

Thus there are two key differences between the object-oriented languages discussed to this point and the multi-method languages. First, there is no designated object to serve as the “receiver” of the message. Second, the selection of method body to execute may depend dynamically on the run-time type of more than one of the arguments of the message.

Another way of viewing multi-method languages is that they support \textit{dynamic} method overloading. A language like Java supports \textit{static} overloading by allowing the programmer to provide a class with multiple methods having the same name but different signatures. Thus multiple method bodies may be associated with the same name for a single object. A simple way of understanding overloading is to imagine each method has the types of its parameters appended to the method name. Java then requires these extended names to be unique. Thus Java will not allow two versions of an overloaded method that have exactly the same parameter types.

In Java, when a programmer calls an overloaded method, the compiler determines which version of the method to call based on the name and the static types of the parameters. Of course the actual method body to execute is determined dynamically based on the run-time type of the receiver, but which version (\textit{i.e.}, which extended name for the method) to use is determined statically. In a multi-method language, the selection of which method to execute is determined at run time based on the types of the values of the actual parameters.

Going back to the example above, suppose we evaluate \texttt{equal(pt1, pt2)} where at run time \texttt{pt1} is a value of type \texttt{Point} and \texttt{pt2} is a value of type \texttt{ColorPoint}. In this case, using the body of the second version of \texttt{equal} would not be possible because the first parameter has run-time type \texttt{Point}, and hence would not conform to the declaration that the first parameter have type \texttt{ColorPoint}. As a result the first method body must be chosen.

In general, multi-method languages search in a predetermined way to find the appropriate “best” match among the available method bodies. In the second example above, where both parameters were actually of type \texttt{ColorPoint} at run time, either of the method bodies might have been called.

\begin{footnote}
In a multi-method language the system may be able to make some decisions on which version of the method to execute based on statically available information as well. If one version has two formal parameters and another has three, that information can be used to eliminate method versions statically.
\end{footnote}
but the second was a better match because the run-time types were exactly the declared types of the formal parameters.

The first example, where both parameters were Points, and the third example, where the first parameter was a Point and the second parameter was a ColorPoint, were both easy because the types of the actual parameters conformed to only one of the method bodies – the first. In some cases, however, the choice of method body to use is not as obvious as for the three cases we have seen so far.

Suppose we define the following methods:

```plaintext
function compare(p1: Point, p2: ColorPoint): Boolean is
{ return ... }
```

```plaintext
function compare(p1: ColorPoint, p2: Point): Boolean is
{ return ... }
```

A call of `compare(pt1, pt2)` for `pt1` and `pt2`, where both have static type `Point`, would not be statically type safe because there is no `compare` method with signature accepting two values of type `Point`. Instead, there are method bodies corresponding to any call involving one `Point` and one `ColorPoint`. Suppose we write `compare(cpt1, cpt2)`, where `cpt1` and `cpt2` both have type `ColorPoint`. If the two definitions above are the only definitions of `compare`, we now have a problem. If the run-time types of `cpt1` and `cpt2` are `ColorPoint` (or indeed any subtype of `ColorPoint`), it is no longer clear what is the best choice for the method body to be executed. Each method choice matches one of the parameters exactly, but the other parameter is better matched by the other version of the method.

For each method call at run time, multi-method languages generally first determine the set of possible method bodies that might be executed. If there is only one, then there is no problem, otherwise an attempt is made to find the version of the method with the “best” fit. The difficulty arises when there is no clear “best” fit as in the situation described above with `compare`.

Different multi-method languages handle this problem in different ways. Some languages (CLOS [DG87], for example) provide rules to specify which method body is to be chosen when there is more than one match, while others (Cecil [Cha95], for example) require the programmer to disambiguate the calls by inserting one or more new method definitions to resolve the ambiguous cases.

If the first way of handling ambiguities is chosen, the language might specify, for example, that first a best fit among the matching method bodies is
found for the first argument, and then for successive arguments. If we were to use this rule, then the call of `compare(cpt1, cpt2)` would result in the execution of the second method body, because it provides the best match for the first argument.

If the second way of handling ambiguities is chosen, the type system might require the programmer to explicitly add a new method body for the case where both parameters are of type ColorPoint as that eliminates all ambiguities in resolving method calls for `compare`.

While Java is not a multi-method language, it does complain about overloaded methods that it cannot resolve statically, and insists that the programmer disambiguate such expressions. An example would be a class containing overloaded methods with two parameters whose corresponding types are in opposite subtyping relations like those in the `compare` methods above.

A major advantage of multi-method languages is in the handling of binary methods. Because there is no designated receiver of the message, all parameter values involved in the computation have equal precedence (aside from orders specified to resolve ambiguities as noted above). Thus if \( m \) is desired to be a binary method, programmers can simply ensure that the types for both parameters are the same for all method definitions of \( m \).

For example, our definitions of the method bodies for `equal`, above, each satisfy the constraint that the types of the two parameters are identical. As can be seen in the examples above, this does not prevent us from calling the `equal` method with parameters of different types. Thus we could write `equal(pt1, pt2)`, where the value of one of the arguments was of type `Point` and the other was of type `ColorPoint`, but we will always have that possibility in languages with subtyping.

We saw that Eiffel also supported binary methods with its `like Current` construct, but that it was not type safe. In Chapter 16, we introduce a new keyword, `MyType`, that stands for the type of `self`. This is similar to Eiffel’s `like Current`, but we will provide static type-checking rules that will ensure that it is safe. Object-oriented languages with `MyType` and a way of specifying exact types, for example the language \( V^OOL \) in Chapter 18, allow the programmer to specify that the type of the parameter is the same as that of the receiver, while also providing the capability of ruling out calls where the value of the actual parameter is a subtype of the formal parameter’s static type.

A major disadvantage of multi-method languages compared to the other object-oriented languages is that it is harder to maintain encapsulation and information hiding. When dynamic method selection depends on several pa-
rameters that may be of different types, there is no obvious place to keep the multi-method definitions. In ordinary single-dispatch object-oriented languages, method lookup takes place by going to the receiving object to find a method with the same name and signature. In multi-method languages, method lookup takes place first by finding all applicable methods with the same name and then finding the “best” match based on the types of the values at run time.

Thus in single-dispatch languages all methods logically belong with the receiver object (or its class), and the methods typically have access to non-public features of the receiver’s object (or class), but not those of its parameters.

In a multi-method language, methods with the same name logically belong together (at least in terms of the lookup strategy). In most cases, multi-methods also need access to non-public features of all parameters used to resolve the method dispatch. This if a multi-method contains has parameters of type Point and Circle, then it may need access to non-public methods and instance variables of both. As a result, it often does not make sense to consider a multi-method as a part of just one class. This also implies that methods outside of a class may need access to features that would normally be hidden outside of that class.

As a result, designers of multi-method languages struggle with modularity issues. Researchers have proposed module systems for multi-methods [CL95], as well as various schemes [Cas97, CM99] for either restricting multi-methods or combining object-oriented programming with multi-methods, but it is not yet clear whether these are sufficient to overcome the difficulties with modularity.

Many variants of multi-method systems exist, including “encapsulated multi-methods”, which blend the object-oriented and multi-method styles of programming by retaining the notion of sending a message to an object, but also dynamically resolving overloading in parameters at run time.

Castagna’s Object-Oriented Programming: A unified foundation [Cas97] is an excellent source of information on the theory of multi-method languages and hybrid systems. CLOS [DG87] and Cecil [Cha95] are interesting examples of multi-method languages.

7.1.3 What languages are truly “object-oriented”?

It seems odd for us to struggle with the definition of object-oriented languages this far into this book. One reason for postponing the discussion until
now is that endless arguments have gone on as to which systems deserve the name “object-oriented” languages. Arguments like this have some technical importance, but they often have public relations consequences as well. After all, object-oriented languages are currently “hot”, so it is an advantage to have your favorite language classified as object-oriented.

Class-based languages and object-based languages are similar in the way that method bodies are selected at execution time. Object-based languages may be syntactically simpler because they have fewer syntactic constructs. However, we believe objects in object-based languages are semantically more complex, because they play the roles of both classes and objects when compared to class-based languages. In particular, the ability to update methods dynamically in objects seems to require more sophisticated modeling in a translational semantics. Static verification of such programs may also be significantly more complex, because one cannot count on the specification of methods to be the same over the lifetime of an object.

Multi-method languages, while they share the notion of dynamic method dispatch with class-based and object-based languages, organize the lookup of method bodies entirely differently. This results in a quite different semantics for such languages. Moreover, because there is not a receiver of messages specified in a call, the notion of self is no longer meaningful. As a result, many prefer not to use the term “object-oriented” for multi-method systems.

An interesting discussion attempting to draw together the two sides of this argument can be found in [Cas95] as well as in Chapter 5 of [Cas97].

Because the use of the term “object-oriented” is so emotionally laden, we prefer not to take sides, but simply to remark that these systems are different. In this text we have chosen to restrict our discussion to class-based languages, so references to object-oriented languages in the rest of this text will refer only to class-based languages. Modeling for object-based languages can be found in [AC96] and for multi-method languages in [Cas97].

7.2 Well-known object-oriented languages

In this section we discuss some interesting features of some well-known object-oriented programming languages. Our focus is on interesting features that are related to their type systems. In particular, we highlight features not covered in the formal models presented in this book.
7.2.1 Simula and Beta

Simula 67 [BDMN73] is generally accepted to be the first object-oriented language. It was defined as an extension of ALGOL 60 to support discrete simulations. It added the notions of objects, classes, inheritance, and dynamic method invocation, but did not support information hiding of either instance variables or methods.

Beta [KMMPN87] was defined later as a modern successor to Simula 67. It uses a very different notation than most object-oriented languages, making it difficult to learn for those not familiar with the language. An important part of the language is the unification of methods and classes into a single construct called a pattern.

An interesting feature of both Simula 67 and Beta is the way that method override is accomplished. Rather than supporting a super construct to allow the programmer to invoke a method from the superclass, instead the programmer includes the keyword inner in the method body. If a new body for the method is given in the subclass, that new body is treated as a replacement for the word inner in the method definition in the superclass. That is, any invocations of the method on an element of the subclass will result in execution of the superclass code that occurs before inner, then execution of the code from the subclass, and then finally execution of the code that follows inner from the superclass.

Beta also supports virtual classes (called virtual patterns in the Beta literature). The example in Figure 7.1 gives the flavor of this construct using a syntax similar to that used in this book. To be consistent with our separation of classes from types, our example will use virtual types rather than virtual classes.

The deftype declaration of type variable A in Cell indicates that it may be a subtype of Object. This is not a type parameter to be instantiated. Instead it is a virtual type, whose declaration indicates that A will denote Object in Cell, but that A may be specialized to a subclass of Object in subclasses of Cell. (This use of “virtual” is similar to that of virtual methods in C++, as a feature that may be overridden in subclasses.) Thus in class StringCell, A is further constrained to be String, with the possibility of further specialization in subclasses of StringCell.

Many of the uses of virtual types are similar to those for polymorphic functions. In GJ (or the languages PSoOL or P.MOOL introduced later), for example, we would write a polymorphic function PolyCell taking a type parameter A, so that Cell and StringCell could be created by applying
class Cell {
  deftype A <: Object;
  value: A := null;

  function set(v: A): Void is
  { value := v }

  function get(): A is
  { return value }
}

class StringCell inherits Cell {
  deftype A <: String;
  ...
}

Figure 7.1  Cell and StringCell classes in Beta.

PolyCell to the types Object and String. In using virtual types, we need not write a polymorphic class. The class Cell defined above takes the place of both PolyCell and Cell.

Moreover, because StringCell extends Cell, and Beta subclasses are assumed to generate subtypes, objects of type StringCell may be used in any context expecting values of type Cell. By this time, the reader should recognize that this may result in type errors because the parameter of method set changes covariantly in moving to the subclass. The designers of Beta realized this, and compensated for this weakness of the static type system by inserting dynamic checks in this circumstance. Thus Beta is type safe by virtue of a combination of static and dynamic type checking.

While the simple example shown above may be easily replaced by a polymorphic class definition, there are other examples where the replacement is not as simple. Figure 7.2 provides a simple example involving mutually recursive types. We assume that ObserverClass generates objects of type Observer, and SubjectClass generates objects of type Subject. The mutually recursive definitions of the types Observer and Subject are written using virtual types.

This is an instance of the Subject-Observer pattern, which is closely re-
Figure 7.2  The Subject-Observer pattern expressed with virtual types.
Figure 7.3 Specializing Subject-Observer to windows.

related to the event-handling style for Java 1.1. Observers send a register message to a subject so that they can be notified when certain events occur. When an event occurs, the subject executes its notifyObservers message to send notify messages to each of its registered observers. Clearly Subject needs to know about Observer and vice versa. In anticipation of specializing these classes, we defined the virtual type S to be used in Observer and O to be used in Subject. Virtual type E is defined in both classes.

Now suppose we wish to specialize both SubjectClass and ObserverClass to work with a special type of event from class WindowEvent. As illustrated in Figure 7.3, the types WindowSubject and WindowObserver have simultaneously been specialized from Subject and Observer.

This usage of virtual types allowed us to simultaneously refine classes and types in ways that would have been extremely awkward without these features. Moreover, as long as we keep objects of type Subject interacting only with those of type Observer, and objects of type WindowSubject interact-
ing only with those of type WindowObserver, no type problems result.

However, if we allow objects of type WindowSubject to interact with objects of type Observer, we get the usual sorts of type insecurities that we have seen before with binary methods. As noted earlier, Beta avoids these problems by including dynamic checks to ensure type safety.

Bruce, Odersky, and Wadler [BOW98] compare the strengths and weaknesses of virtual classes with respect to F-bounded polymorphism. That paper also suggests that many of the uses of virtual classes may be seen to be a mutually recursive generalization of the $MyType$ construct introduced in Chapter 16. Bruce and Vanderwaart [BV99] present the design and translational semantics for such a construct. Igarashi and Pierce [IP98] present an alternative semantics using existential types that captures different features of virtual classes.

### 7.2.2 Java and C++

C++ and Java are two of the most popular statically typed class-based object-oriented languages in use today. C++ is actually a hybrid language that contains features of both procedural and object-oriented languages, while Java is designed to only support an object-oriented style of programming. As a result we will place more emphasis in our discussion on Java.

Java’s interfaces are quite similar to the object types discussed in this book. Java and C++ both originally supported the invariant type discipline for subclasses. That is, parameter and return types for overridden methods were required to be the same as for the method in the superclass. C++ later loosened its restrictions to allow covariant changes to return types.

C++ supports multiple inheritance on classes, while Java is restricted to single inheritance on classes. However, Java allows new interfaces to extend multiple existing interfaces. Thus Java’s rules for subclasses and subtypes are similar to those presented here. However, both Java and C++ only recognize declared extensions of either classes or interfaces. For example, even if interface B contains all of the methods of interface A (with the same signatures), B will not be recognized as an extension of A unless it is declared to be so in B’s definition. Thus subtypes are determined by declaration, rather than structurally as in this text.

Interestingly, Java does not fully support subtyping. The failure arises in conditional expressions of the form $bexp ? texp : fexp$. In that expression, $bexp$ must have type Boolean. If $bexp$ is true at run time then the value of $texp$ will be returned. Otherwise, the value of $fexp$ will be returned.
Java type-checking rules require that the types of `texp` and `fexp` either be the same or that one extends the other. For example, suppose that `ColorPoint` and `FatPoint` both extend interface `Point`, with variables `cpt` of type `ColorPoint`, `fpt` of type `FatPoint`, and `pt` of type `Point`. Then the expressions `bexp?pt:pt`, `bexp?pt: cpt`, and `bexp?fpt:pt` are well-typed, but `bexp?fpt:cpt` is not well-typed. Even though Java recognizes that the types of both `fpt` and `cpt` extend `Point`, and using expressions of type `Point` in both branches of the conditional would be fine, the expression with both branches having distinct subtypes of `Point` is not allowed.

The reason for this restriction in Java is that types (interfaces) with an upper bound need not have a least upper bound. For example, suppose interfaces `C` and `D` are both defined to extend both `A` and `B` as in Figure 7.4. Thus each of them contains at least the union of the features declared in `A` and `B`. However if `c` has type `C` and `d` has type `D`, it is not clear what type to assign to `bexp?c:d`. Either type `A` or `B` would be plausible, but there is no reason to prefer one to the other. If the system could synthesize a new interface containing exactly the features of `A` and `B`, and that was declared to extend both, it would be the obvious choice for the type of the conditional. Since that is not possible in Java, the type system is restricted to handle only the simpler cases where the type of one branch extends that of the other.\footnote{Interestingly, a similar problem arose during the design and implementation of a type system for a functional object-oriented language \cite{BCD93} in the early 1990's.}

Java does support subtyping via a subsumption rule in the context of assignment statements and with actual parameters in method calls. As a result, most programmers are unaware that Java does not fully support subtyping.

The visibility rules of Java and C++ are a bit different from those described in this book. Suppose the definition of class `C` includes the declaration of
method \( m \) which takes a parameter \( c \) of class \( C \). In the body of \( m \), it is possible to access private and protected variables of the parameter of type \( C \).

Because we do use classes as types in this book, we do not shown how to model this access. A semantics modeling this access to private features is difficult to achieve. Fisher and Mitchell [FM98] show how to obtain this access using two different kinds of classes and by nesting “existential quantifiers”, which are discussed in Section 9.3.

### 7.2.3 Smalltalk

Smalltalk [GR83] can be considered the first pure object-oriented language. It is a dynamically typed language. Therefore there is no need to restrict subtyping to subclasses or vice versa. The only issue is whether a particular object can find a method to handle a given message send at run time.

William Cook published an interesting paper [Coo92] that compared the inheritance graph of classes in the built-in Smalltalk libraries with subtype relations between objects generated by these classes. One of his interesting findings was that there was little correlation between subclass and subtype. He found a significant number of instances in which \( B \) was a subclass of \( A \), but that the object type corresponding to \( A \) was a subtype of that corresponding to \( B \).

While Smalltalk is an interesting and important object-oriented language, we do not discuss it further here because we are primarily focussing on statically typed languages in this book.

### 7.2.4 Eiffel and Sather

We presented a rather extensive analysis of Eiffel in Section 4.2. While Eiffel has a very flexible static type system, we showed that it was not sound. While several proposals [Mey92, Mey95a, Coo89b] have been made to “fix” the type system, to our knowledge none of these has yet been implemented in a publicly available compiler.

Sather [Omo91, Omo93] was designed as a more efficient variant of Eiffel. One of the goals of the Sather design was to fix the type insecurities of Eiffel. It accomplished this by only allowing contravariant changes in parameter types in subtypes, and allowing no changes in instance variables. In Sather, inheritance from abstract classes is used to define the subtype hierarchy, while inheritance from concrete classes does not result in subtypes. Thus subtyping and inheritance are somewhat independent.
Both Eiffel and Sather support multiple inheritance and parameterized classes. Multiple inheritance is an essential feature of each. For example, if a method is overridden in a subclass, multiple inheritance must be used if it is desired to call the original version of the method from the superclass.

Eiffel supports “anchor types”. Anchor types allow the programmer to declare a type to be “like” that of another identifier. For example, suppose that in class C, x is declared to be an instance variable of type A. Then the expression like x can be used as a type expression within C. Let D be a subclass of C in which the type of x is changed to be of type B (recall that Eiffel allows covariant changes to the types of instance variables). Then all inherited occurrences of like x also change type to B. This construct provides expressiveness similar to Beta’s virtual classes.

A special case of this construct is the use in Eiffel of like Current. Current is Eiffel’s name for the object we have called self, and that C++ and Java call this. Thus the type like Current is very much like the MyType construct we introduce in Chapter 16.

The uses of anchor types can be replaced by using parameterized types, but this is not as convenient as the use of the anchor types. Unfortunately, the use of anchor types can result in type errors as they can result in covariant changes in parameter and instance variable types in subclasses.

7.3 Summary

In this chapter we discussed different varieties of object-oriented languages, ranging from class-based to object-based to multi-method languages. We also discussed interesting language features from Simula 67, Beta, Java, C++, Smalltalk, Eiffel, and Sather. While several of these have interesting features not discussed earlier, they all share many of the same structures discussed in this book. Interesting work has been done in understanding many of these features using techniques similar to those presented in the rest of this book.

In the next section of the book we will introduce the lambda calculus, a formal language often used for describing the semantics of programming languages. We begin with the simply-typed lambda calculus and build to a higher-order polymorphic lambda calculus supporting bounded parametric polymorphism and a construct known as “existential types”, which are useful in modeling information hiding. In the rest of the book we will use the lambda calculus to understand the semantics of object-oriented languages and to verify the soundness of the type systems for these languages.
This page intentionally left blank
Historical Notes and References for Section I

Rather than attempting a full-scale history of object-oriented programming languages, we simply sketch some of the more influential events relating to statically-typed object-oriented languages, subtyping, and inheritance that have not already been described in earlier chapters. We postpone most of the discussion of more theoretical work until Section III.

Simula 67 is generally regarded as the first object-oriented programming language. It was designed as an extension of ALGOL 60 with features added to perform discrete simulations. It introduced the notions of class and inheritance, though no information-hiding facilities were provided in the language.

Smalltalk’s design was influenced by Simula 67 and LOGO, a language used to teach programming to children. Smalltalk was originally conceived by Alan Kay as the programming language for the Dynabook, a prototype that eventually influenced the design of laptop computers. The idea was that the Dynabook would be easily programmable by the user via a language that supported the use of graphic interfaces, communications to external databases, and audio interfaces. Smalltalk-72 was the first version of the language, while several generations of improvements resulted in Smalltalk-80, the version of the language most commonly described.

Although Smalltalk is dynamically typed, attempts have been made to create statically typed variants. One of the earliest was the system of Borning and Ingalls [BI82]. A very interesting recent effort in designing a type system for Smalltalk resulted in the language Strongtalk [BG93]. The type system includes a type `Self`, which is similar to (and was influenced by) the `MyType` construct discussed in Chapter 16. That paper contains references to earlier
attempts at providing static type systems for Smalltalk.

By the mid-80’s it had become popular to add object-oriented features to more traditional languages. This resulted in hybrid languages like Objective C [Cox86], C++ [Str86, ES90], Object Pascal [Tes85], Modula-3 [CDG+88], Oberon [Wir88], Objective CAML [RV98], and CLOS [DG87].

Eiffel [Mey88, Mey92] was designed in the mid-1980’s by Bertrand Meyer as a pure object-oriented language. It has a Pascal-like syntax, though it is not an extension of Pascal. We have already discussed many of the features of Eiffel in Section 4.2. An interesting feature of Eiffel not discussed earlier is its support for design-by-contract, with built-in features that enable run-time checking for preconditions, postconditions, variants, and invariants.

Cook [Coo89b] was one of the earliest to discuss Eiffel’s type insecurities. Problems include support for covariant type changes in instance variable and parameter types (including support for indirect changes with the `like` construct) in subclasses, and the ability to hide methods in subclasses. The Eiffel 3 language reference [Mey92] discussed one possible fix – a link-time data-flow analysis of an entire program that determines whether there are possible type insecurities. Several years later, during an invited address at the OOPSLA conference, Meyer [Mey95b] proposed a different solution, termed “No polymorphic catcalls”, that he claimed could be implemented with an incremental compiler.

At the time of the publication of this book, we know of no compiler that implements either of these solutions to the type insecurities of Eiffel, though we don’t know why. It is possible that they are overly conservative, ruling out too many useful programs that would run without type error. Alternatively, these checks may be too expensive for practical use.

As noted earlier, Sather [Omo91] was a redesign of Eiffel that emphasized efficiency and fixed the problems with type safety. The language Trellis/Owl [SCB+86] seems to be the first language to introduce the `MyType` construct that we introduce in Chapter 16. The language was restricted to ensure that subclasses always gave rise to subtypes. Thus `MyType` was only allowed as the return type of methods. The language Emerald [BHJ+87] was an interesting object-oriented language that supported subtypes but no classes or inheritance. It did include type parameters and a notion similar to F-bounded polymorphism.

Beta [KMMPN87] is a purely object-oriented language that is a successor to Simula 67. Its virtual classes, discussed in Section 7.2.1, provide an alternative (and safe) approach to the flexibility of Eiffel. Beta uses both static and dynamic type checking in order to ensure type safety. When it cannot
assure the safety of a section of code statically, it inserts dynamic checks to
guarantee the absence of undetected type errors.

The programming language Java was designed to be an object-oriented
language. While it shares most of the same syntax with C++, it differs from
that language in several ways. Two of the most important are the use of
garbage collection and its goal of platform independence. It was described
briefly in Section 4.1.2.

The GJ [BOSW98] extension of Java is based on Pizza [OW97], an earlier
extension by some of the same authors. An important feature of GJ is its
ability to “retrofit” existing classes and interfaces by assigning polymorphic
GJ types to existing non-polymorphic Java libraries. It also generates code
that can be executed by existing Java Virtual Machine implementations. It
is likely that GJ features will be included in a future version of Java. GJ is
interesting in that its designers are members of the theoretical programming
languages community. Pizza has some features normally found only in func-
tional programming languages.

Other extensions of Java that supported parametric polymorphism were
proposed in papers by Myers et al. [MBL97], Ageson et al. [AFM97], and
Bruce and his students [Bru97, Bur98, Fos01]. Extensions to Java incorporat-
ing constructs similar to Beta’s virtual classes have been proposed by Thorup
and Torgersen [Tho97, Tor97, TT97]. Other proposed extensions of Java can
be found in recent proceedings of the OOPSLA and ECOOP conferences.

Section 7.1 contained an extensive discussion of object-based and multi-
method languages. As noted earlier, Castagna’s book, Object-Oriented Pro-
gramming: A unified foundation [Cas97], provides a comprehensive discussion
of the theoretical foundations of multi-method languages.

To the earlier discussion we add that David Ungar’s language Self [US87,
CU89] has been an influential object-based programming language. Partially
because it is dynamically typed, it required clever compiler optimizations to
run efficiently. The technology developed for this has had a major impact
on the implementations of other object-oriented languages, both class-based
and object-based. Ungar’s former student Craig Chambers [Cha95, CL95,
CM99] has been one of the leading proponents of statically typed multi-
method programming languages.

Peter Wegner has written a number of papers explaining the concepts of
object-oriented programming languages. Two comprehensive surveys are
“Dimensions of Object-Oriented Language Design” [Weg87] and “Concepts
and Paradigms of Object-Oriented Programming” [Weg90].

There are several influential early theoretical papers on object-oriented
programming languages. The first is John Reynolds' 1975 paper [Rey75] “User-defined types and procedural data structures as complementary approaches to data abstraction.” That paper compared user-defined types (essentially ADT’s) with procedural data structures (essentially objects), comparing their advantages and disadvantages. Another early paper is also by Reynolds [Rey80]: “Using Category Theory to Design Implicit Conversions and Generic Operators”. That paper discusses implicit conversions of values from one type to another (like those that occur with subtypes) from the point of view of category theory. (A later paper, “An algebraic model of subtype and inheritance” [BW90], by Bruce and Wegner, discussed similar issues in object-oriented languages, using order-sorted algebras rather than category theory.)

Another important early paper is Luca Cardelli’s paper [Car88] (originally published as a conference paper in 1984), “A Semantics of Multiple Inheritance”. That paper described objects as (recursive) records and carefully set out the rules for subtyping, including both depth (specializing the types of fields of a record) and breadth (adding new fields) subtyping for records, and the correct subtyping rules for function types – covariant in return types and contravariant in parameter types. This paper also presented a type-checking algorithm for languages with subtyping.

The Emerald designers, as well as America [Ame87, AvdL90] and his collaborators, were among the earliest to argue that the subtype and class hierarchies should not be confounded. (Recall that Emerald did not even support inheritance!) The arguments for this separation were also expounded forcefully by the Abel research group at HP Labs in [CHC90]. Cook’s analysis [Coo92] of Smalltalk’s collection classes provided further evidence of the disconnect between subclasses and subtypes.

Binary methods have long created difficulties for statically typed object-oriented languages. The paper, “On binary methods” [BCC+95], was written by a group of researchers with quite different views on the subject in order to lay out different options for dealing with binary methods. It provides a useful discussion of the problem, and strengths and weaknesses of various attempted solutions. In later chapters we will go into more detail on solutions involving F-bounded polymorphism and MyType.
Part II

Foundations:
The Lambda Calculus
How can we be sure that a program is in the legal syntax for a given programming language? How can we be sure that the program will pass through the type checker without error? How do we know whether it will terminate and produce a particular result, run forever, or die with an error? Of course we could submit the program to a compiler for the language in order to answer the first two questions and then run it to obtain the answer to the third, or we could use an interpreter for the language to answer all three questions.

However, that line of reasoning simply begs the question: how do the compiler or interpreter writers know how to handle these issues? Many examples exist of languages where different compilers or interpreters give different answers to these questions for the same programs.

Language designers often try to answer these questions by providing language reference manuals written in discursive style. However, this method leaves open the possibility of ambiguity or other underspecification of the language. A more reliable way of answering these questions is to provide a formal, unambiguous description of the language at hand.

In this chapter we begin our introduction to the formal description of programming languages. We will be interested in careful formal descriptions of language syntax, type-checking rules, and semantics. While the formalism is not difficult, it does require some time to get used to the notation. As a result, in this chapter we focus on a formal description of the lambda calculus, a very simple functional language first introduced by Alonzo Church [Chu41] in order to describe notions of computability. Rather than look at the original untyped lambda calculus, we will instead focus on a typed version as this will allow us to exhibit the type-checking rules for the language.

While it may take some effort to learn to read these formal descriptions, the effort in learning to read them will pay off both with more precise and
concise descriptions of languages. In particular, we will be able to see precisely what the rules are both for forming type-correct programs and how they will execute.

Aside from our use of the lambda calculus as a simple language in which to apply our formal analysis, our primary use of the lambda calculus will be as a tool to express the semantics of object-oriented languages. We will not attempt to give an in-depth presentation of the lambda calculus, instead presenting only the essentials needed here. Instead interested readers are invited to consult more specialized texts like [Bar84] or [HS86] for a more detailed presentation of the lambda calculus.

We begin this chapter with an introduction to the simply-typed lambda calculus. Because they will be helpful in modeling common features in programming languages, in the second section we add pairs, sums, records, and references. The following chapter adds higher-order features like parametric polymorphism and information hiding via existential types. It also introduces features to support recursive types and expressions as well as subtyping.

As a result, in the next section of the book we will be prepared to use these methods of providing formal descriptions to provide a careful formal definition of a simply-typed object-oriented language, SOOL. Because we will be discussing a language and its translation into enriched forms of the lambda calculus, we will use different fonts for expressions of these languages. We continue to use a teletype-style font for the syntax of object-oriented languages, while we use an italicized font for the expressions of the lambda calculi introduced here.

8.1 The simply-typed lambda calculus

\[ \Lambda^+ \]

In this section we present the simply-typed lambda calculus, written \( \Lambda^+ \). In later sections, we add new types and constructs which make it easier to model real languages.

We can think of the typed lambda calculus as an assembly language for functional programming languages. The fundamental constructs of the language only include ways of constructing functions and applying functions to arguments. Because of our concern with typing issues, we will only consider the typed lambda calculus. The only thing that distinguishes the syntax of the typed lambda calculus from the untyped version is that formal parameters of functions must be provided with their types.
8.1 The simply-typed lambda calculus

A first simple example of a function in the typed lambda calculus is the successor function on elements of type \(\text{Integer}^1\).

\[
\lambda(x: \text{Integer}). x + 1
\]

Functions of the typed lambda calculus are anonymous – that is, they do not have an associated name. The “\(\lambda(x: \text{Integer})\)” indicates the function has a formal parameter \(x\) of type \(\text{Integer}\). Readers familiar with LISP or Scheme will recognize the use of “lambda” to introduce function expressions (the use of lambda in LISP reportedly came from the lambda calculus). Other functional languages may introduce alternative notations such as “fun(x: Integer)” or “function(x:Integer)”.

The body of the function appears after the period in lambda calculus expressions. The “.” notation separating the parameters from function body has low precedence. As a result the function body normally includes as large a subexpression as possible. In the case of the successor function above, the body is \(x+1\).

We write the type of the successor function as \(\text{Integer} \rightarrow \text{Integer}\). This type describes functions that take an argument of type \(\text{Integer}\) and return an answer of type \(\text{Integer}\). As in our earlier examples, the type \(T \rightarrow U\) represents a function with domain \(T\) and range \(U\).

We can apply a function to an argument simply by placing the argument to the right of the function. The argument may be surrounded by parentheses if desired, but this is not necessary. Thus

\[
(\lambda(x: \text{Integer}). x + 1)17
\]

and

\[
(\lambda(x: \text{Integer}). x + 1)(17)
\]

both represent the application of the successor function to the number 17. Needless to say, the type of this expression is \(\text{Integer}\).

The following example is a function which takes two numbers and adds them together:

\[
\lambda(x: \text{Integer}). \lambda(y: \text{Integer}). x + y
\]

---

1. We assume, at least temporarily, that the lambda calculus contains integer constants and the usual arithmetic functions. Later we will introduce a constant to represent the addition function. We will write constants used in the lambda calculus in the form \(c\) in order to distinguish them from variables or other expressions of the language.
Because we have written the two $\lambda$‘s successively, this actually represents a function which returns a function, rather than a function which takes two arguments. We can see this more clearly if we insert parentheses:

$$\lambda(x: \text{Integer}). (\lambda(y: \text{Integer}). x + y)$$

With the parentheses inserted, it is clearer that this is a function taking an argument $x$ with type $\text{Integer}$, and whose body is

$$\lambda(y: \text{Integer}). x + y$$

In particular, the result of applying this function to the integer constant $17$, written

$$(\lambda(x: \text{Integer}). \lambda(y: \text{Integer}). x + y)17$$

would be a function of type $\text{Integer} \rightarrow \text{Integer}$ that takes an argument, $y$, and adds $17$ to it. We will see later that it evaluates to

$$\lambda(y: \text{Integer}). 17 + y$$

In particular, we must apply the original function to two arguments in succession in order to fully evaluate it:

$$(\lambda(x: \text{Integer}). \lambda(y: \text{Integer}). x + y)(17)(13)$$

This expression evaluates to the number 30.

Thus the type of the original function is $\text{Integer} \rightarrow (\text{Integer} \rightarrow \text{Integer})$. This is the type of functions which take an argument of type $\text{Integer}$, and return a function of type $\text{Integer} \rightarrow \text{Integer}$.

That is really all there is to the typed lambda calculus. It contains constants, function definitions, and function applications. In order to specify completely the syntax of the typed lambda calculus, we need a notation to provide a formal description of the language syntax. Let us now introduce the notation to provide a formal description of the language.

The syntax of expressions of a language can be specified by context-free grammars which consist of a set of “productions” of the form:

$$a, b \in \text{Cat} ::= \exp_1 | \ldots | \exp_n$$

In this definition, $\text{Cat}$ represents the set of expressions being defined. The expressions to the left indicate typical notation used for representing arbitrary expressions from this set; in this case, “$a$” and “$b$”. The symbol “$::=$” can be
read as “is composed of”, while “|” separates alternatives on the right side of the definition. Each of the \( exp_i \) on the right side provides a template for the different kinds of elements which are contained in \( Cat \). Thus \( Cat \) consists of expressions in the forms \( exp_1 \) through \( exp_n \).

Here is a very simple grammar for non-empty strings of digits:

\[
\begin{align*}
    d & \in \text{Digit} ::= 0 \mid 1 \mid \ldots \mid 9 \\
    s & \in \text{DigString} ::= d \mid s\ d
\end{align*}
\]

A \( \text{Digit} \), which will typically be written with the identifier \( d \), is one of the integer constants representing values between 0 and 9. A \( \text{DigString} \) is either a single digit (the case \( d \)) or consists of a \( \text{DigString} \) (represented by \( s \)) followed by a digit.

From the first rule, 5 is a \( \text{Digit} \). From the first alternative in the second rule, 57 is also a \( \text{DigString} \). From the second alternative of that rule, 57 is also a \( \text{DigString} \) (because 5 is a \( \text{DigString} \) by the above and 7 is a \( \text{Digit} \) by the first rule). Longer elements of \( \text{DigString} \) can be constructed similarly.

We can use this notation to describe the syntax of the typed lambda calculus. The typed lambda calculus has two sorts of expressions: expressions representing types, and expressions representing typed values of the language. We begin with the definition of type expressions. In defining expressions of the typed lambda calculus we can start with an arbitrary collection, \( TC \), of type constants (which may include \( \text{Integer} \), \( \text{Boolean} \), etc.). The set, \( Type \), of type expressions of the simply-typed lambda calculus, \( \Lambda^\to \), is given by:

\[
T \in Type ::= C \mid T_1 \to T_2 \mid (T)
\]

where \( C \in TC \). Clearly this definition is parameterized by \( TC \), but for simplicity, we do not show this in the notation \( Type \).

This definition states that the set \( Type \) is composed of (i) type constants \( C \) from the set \( TC \), (ii) expressions of the form \( T_1 \to T_2 \) where \( T_1 \) and \( T_2 \) are any elements of set \( Type \), and (iii) expressions of the form \( (T) \) for \( T \) an element of set \( Type \). In other words, type expressions are built up from type constants, \( C \), by constructing function types, and using parentheses to group type expressions. Typical elements of \( Type \) include \( \text{Integer} \), \( \text{Boolean} \rightarrow \text{Integer} \), \( \text{Integer} \rightarrow \text{Boolean} \rightarrow \text{Integer} \), \( \text{Integer} \rightarrow \text{Integer} \), and \( \text{Boolean} \rightarrow \text{Integer} \rightarrow \text{Integer} \).

When defining the expressions of a typed programming language, we will be careful to distinguish the pre-expressions from the expressions of the language. The difference is that the pre-expressions are syntactically correct, but may not be typable, while the expressions are those that pass the type checker.
Expressions of the typed lambda calculus can include elements from an arbitrary set of constant expressions, \( \mathcal{E}C \). Each of these constants comes with an associated type. For example, \( \mathcal{E}C \) might include constants representing integers such as \( 0, 1, \ldots \), all with type \( \text{Integer} \); booleans such as \( \text{true} \) and \( \text{false} \), with type \( \text{Boolean} \); and operations such as \( \text{plus} \) and \( \text{mult} \) with type \( \text{Integer} \rightarrow \text{Integer} \rightarrow \text{Integer} \) (essentially prefix versions of \( + \) and \( \times \)). We will leave \( \mathcal{E}C \) unspecified most of the time, using constant symbols freely where it enhances our examples. As above, we will underline constants of the language to distinguish them.

The collection of pre-expressions of the typed lambda calculus, \( \mathcal{T} \mathcal{LC}E \), are given with respect to a collection of type constants, \( \mathcal{T}C \), a collection of expression identifiers, \( \mathcal{EI} \), and expression constants, \( \mathcal{EC} \):

\[
M, N \in \mathcal{T} \mathcal{LC}E \ ::= \ c \mid x \mid \lambda x : T . M \mid M N \mid (M)
\]

where \( x \in \mathcal{EI} \) and \( c \in \mathcal{EC} \). As with types, this definition is parameterized by the choice of \( \mathcal{T}C, \mathcal{EI} \), and \( \mathcal{EC} \).

Pre-expressions of \( \mathcal{T} \mathcal{LC}E \), typically written as \( M, N \) or variants decorated with primes or subscripts, are composed of constants, \( c \), from \( \mathcal{EC} \); identifiers, \( x \), from \( \mathcal{EI} \); function definitions, \( \lambda x : T . M \); and function applications, \( M N \). Also, as with type expressions, any pre-expression, \( M \), may be surrounded by parentheses, \( (M) \). All formal parameters in function definitions are associated with a type.

We treat function application as having higher precedence than lambda abstraction. Thus \( \lambda x : T . M N \) is equivalent to \( \lambda x : T . (M N) \).

In order to complete the specification of expressions of the typed lambda calculus, we need to write down type-checking rules that can be used to determine if a pre-expression is type correct. Expressions being type checked often include identifiers, typically introduced as formal parameters along with their types. In order to type check expressions we need to know what the type is for each identifier.

We use \( "\overset{\Delta}{=}" \) here and later to indicate that an expression to the left of \( \overset{\Delta}{=} \) is defined by the expression to the right.

**Definition 8.1.1** The collection of free identifiers of an expression \( M \), written \( \text{FI}(M) \), is defined as follows:

1. \( \text{FI}(c) \overset{\Delta}{=} \emptyset \), for \( c \in \mathcal{EC} \),
2. \( \text{FI}(x) \overset{\Delta}{=} \{x\} \), for \( x \in \mathcal{EI} \),
3. \( FI(\lambda(x:T).M) \triangleq FI(M) - \{x\} \),

4. \( FI(M \ N) \triangleq FI(M) \cup FI(N) \).

When an identifier is used as a formal parameter of a function, its occurrences in the function body are no longer free. Instead we say they are bound identifiers. Thus \( FI((\text{plus} \ x) \ y) = \{x, y\} \), but \( FI(\lambda(x:\text{Integer}).(\text{plus} \ x) \ y) = \{y\} \). Notice that bound identifiers are supplied with a type when they are declared as formal parameters, but free identifiers are not textually associated with types in the expressions containing them.

Thus the type-checking rules require information about the type \( s \) of free identifiers. This information is provided by a static type environment, \( \mathcal{E} \), which associates types with expression identifiers.

**Definition 8.1.2** A static type environment, \( \mathcal{E} \), is a finite set of associations between identifiers and type expressions of the form \( x: T \), where each \( x \) is unique in \( \mathcal{E} \) and \( T \) is a type. If \( x: T \in \mathcal{E} \), then we sometimes write \( \mathcal{E}(x) = T \).

**Type-checking rules** can be in one of two forms. A rule of the form

\[ \mathcal{E} \vdash M: T \]

indicates that with the typing of free identifiers given in \( \mathcal{E} \), the expression \( M \) has type \( T \).

A rule of the form

\[ \begin{array}{c}
\mathcal{E}_1 \vdash M_1: T_1, \ldots, \mathcal{E}_n \vdash M_n: T_n \\
\hline
\mathcal{E} \vdash M: T
\end{array} \]

indicates that with the typing of free identifiers in \( \mathcal{E} \), expression \( M \) has type \( T \) if the assertions above the horizontal line all hold. Thus the hypotheses of the rule all occur above the horizontal line, while the conclusion is placed below the line.

Now that we have a formalism to express type-checking rules, we can define the set of (legal) expressions of the typed lambda calculus by specifying the rules for determining how to assign types to expressions. The collection of expressions of the typed lambda calculus with respect to \( TC \) and \( EC \) is the collection of pre-expressions which can be assigned a type by the rules in Figure 8.1.

The intuitive meaning of these rules should be fairly clear. The **Identifier** rule states that if \( \mathcal{E} \) indicates that identifier \( x \) has type \( T \), then \( x \) has that type. The **Constant** rule states that a constant has whatever type is associated with
Identifier \[ \mathcal{E} \cup \{ x : T \} \vdash x : T \]

Constant \[ \mathcal{E} \vdash c : C \]

where \( C \in \mathcal{T} \mathcal{C} \) is the pre-assigned type for constant \( c \in \mathcal{E} C \).

Function \[ \begin{array}{c} \mathcal{E} \cup \{ x : T \} \vdash M : T' \\ \mathcal{E} \vdash \lambda(x : T). M : T \rightarrow T' \end{array} \]

Application \[ \begin{array}{c} \mathcal{E} \vdash M : T \rightarrow T', \quad \mathcal{E} \vdash N : T \\ \mathcal{E} \vdash MN : T' \end{array} \]

Parens \[ \begin{array}{c} \mathcal{E} \vdash M : T \\ \mathcal{E} \vdash (M) : T \end{array} \]

**Figure 8.1** Typing rules for expressions of the typed lambda calculus.

it in \( \mathcal{E} \mathcal{C} \). Skipping the Function rule momentarily, the Application rule states that a function application \( MN \) has type \( T' \) as long as the type of the function, \( M \), is of the form \( T \rightarrow T' \), and the actual argument, \( N \), has type \( T \), matching the type of the domain of \( M \).

The Function rule is the most complex. The idea is that because the formal parameter of the function occurs in the body, the formal parameter and its type need to be added to the environment when type checking the body. Thus if \( \lambda(x : T). M \) is type checked in environment \( \mathcal{E} \), then the body, \( M \), should be type checked in the environment \( \mathcal{E} \cup \{ x : T \} \). (Recall that the environment \( \mathcal{E} \cup \{ x : T \} \) is legal only if \( x \) does not already occur in \( \mathcal{E} \).)

For example, in typing the function \( \lambda(x : \text{Integer}). x + 1 \), the body, \( x + 1 \), should be type checked in an environment in which \( x \) has type \( \text{Integer} \).

Type-checking rules can also be interpreted algorithmically by reading the rules from the bottom left in a clockwise direction. For example, suppose we wish to type check an expression of the form \( MN \) under the assumptions in a type environment \( \mathcal{E} \). An expression of this form is found on the bottom left of the Application rule. Proceed clockwise to the top of the rule and we see that we need to find the types of \( M \) and \( N \) under \( \mathcal{E} \). If the first has type \( T \rightarrow T' \) and the second has type \( T \) (i.e., the domain of the first is the same as the type of the second), then the resulting type of \( MN \) is \( T' \).

The Parens rule simply states that adding parentheses has no effect on the
8.1 The simply-typed lambda calculus

type of an expression. From now on, we will omit type-checking rules for adding parentheses to expressions.

We see how these rules are used to assign types to expressions by looking at a few examples. Let plus be the constant with type Integer \( \rightarrow \) Integer \( \rightarrow \) Integer representing addition that was discussed earlier. We wish to determine the type of the expression \( \lambda(x: \text{Integer}).(\text{plus } x) x \) in the empty environment \( \mathcal{E}_\emptyset = \emptyset \).

In working through this example, we write “??” as the type for expressions whose types have not yet been determined. Thus we start with

\[ \mathcal{E}_\emptyset \vdash \lambda(x: \text{Integer}).(\text{plus } x) x: ?? \]

and use the rules to figure out the appropriate type to replace “??”.

By the rule Function, to type check this function we must check the body \( \text{(plus } x) x \) in the environment \( \mathcal{E}_1 = \{ x: \text{Integer} \} \):

\[ \mathcal{E}_1 \vdash \text{(plus } x) x: ?? \]

Because the body is a function application, we type check the function and argument to make sure their types match. Type checking the argument is easy as:

(8.1) \[ \mathcal{E}_1 \vdash x: \text{Integer} \]

by rule Identifier.

Type-checking plus \( x \) is a bit more complex, because it is a function application as well. However, by rule Constant,

(8.2) \[ \mathcal{E}_1 \vdash \text{plus: Integer } \rightarrow \text{Integer } \rightarrow \text{Integer} \]

and by rule Identifier, we again get \( \mathcal{E}_1 \vdash x: \text{Integer} \). Because the domain of the type of plus and the type of \( x \) are the same,

(8.3) \[ \mathcal{E}_1 \vdash \text{plus } x: \text{Integer } \rightarrow \text{Integer} \]

by lines 8.2, 8.1, and the Application rule.

By another use of the Application rule with lines 8.3 and 8.1,

(8.4) \[ \mathcal{E}_1 \vdash \text{(plus } x) x: \text{Integer} \]

Finally, by the Function rule and line 8.4,

(8.5) \[ \mathcal{E}_\emptyset \vdash \lambda(x: \text{Integer}).(\text{plus } x) x: \text{Integer } \rightarrow \text{Integer} \]

Now that we have typed this expression, it is easy to see how to type the following expression: \( (\lambda(x: \text{Integer}).(\text{plus } x) x) 17 \). Because the type of the
function is given in line 8.5 as $\text{Integer} \rightarrow \text{Integer}$, and $\mathcal{E}_0 \vdash 17$: $\text{Integer}$, it follows from the Application rule that

\[(8.6) \mathcal{E}_0 \vdash (\lambda(x: \text{Integer}). (\text{plus } x) x) 17: \text{Integer}\]

With the above definitions and type-checking rules, we can now determine what are the legal expressions of $\text{AX}$, and we can determine their types. However, we do not yet have any information on how to evaluate expressions. That is, our intuitive understanding of function definition and application and the intuitive understanding of the constants used leads us to believe that the result of evaluating $(\lambda(x: \text{Integer}). (\text{plus } x) x) 17$ should give us the value 34. However, we would like to provide formal rules for evaluating these expressions so that there is no ambiguity about how the evaluation takes place or the resulting answer.

The formal computation rules of the typed lambda calculus depend on a careful definition of substitution in lambda expressions. The ($\beta$) “reduction rule” for evaluating lambda expressions given later is formulated in terms of replacing all occurrences of an identifier in an expression by another expression. Rather strange errors can occur if one is not careful in defining such substitutions. As a result we give a precise definition of substitution in the lambda calculus.

**Definition 8.1.3** We write $[N/x]M$ to denote the result of replacing all free occurrences of identifier $x$ by $N$ in expression $M$.

1. $[N/x]c \equiv \lambda$, if $c \in \mathcal{E}$,
2. $[N/x]x \equiv N$,
3. $[N/x]y \equiv y$, if $y \neq x$,
4. $[N/x](L M) \equiv ([N/x] L) ([N/x] M)$,
5. $[N/x](\lambda(y: T). M) \equiv \lambda(y: T). ([N/x] M)$, if $y \neq x$ and $y \notin \text{FI}(N)$,
6. $[N/x](\lambda(x: T). M) \equiv \lambda(x: T). M$.

The first three rules should be clear: no identifiers exist in constants, so no changes are made; if substituting for an identifier, replace it if the names match, otherwise no changes should be made. The fourth rule simply states that to substitute into a function application, just substitute into the function and its argument.
The last two rules deal with the case of substitution into function definitions. What to do depends on whether or not the identifier being substituted for is the same as the name of the formal parameter. In the last rule, where \( x \) is the name of the formal parameter of the function, no replacement is made in the body of the function because all values of \( x \) in the body are “bound” by the \( \lambda \). Looked at from a traditional programming language point of view, the use of \( x \) as a formal parameter creates a “hole” in the static scope of the free \( x \), no instance of the free \( x \) can occur inside the function.\(^2\) As a result no substitutions should be made.

The fifth rule is similar to the fourth in that if the name of the formal parameter is different from the identifier being substituted for, we just make the substitution in the function body. However, this rule only holds if the name of the formal parameter is not a free identifier in \( N \), the expression being substituted in. Here is an example showing why this restriction is necessary.

Let \( M \) be the function \( \lambda(y: T).x + y \), a function which adds the formal parameter to a free identifier, \( x \). Suppose we substitute the expression \( y \ast y \), where \( y \) is a free identifier, in place of \( x \). If we blindly apply rule (5), without observing the restriction, then we would obtain \( \lambda(y: T).y \ast y + y \), an expression which squares the formal parameter and then adds it to itself.

On the other hand, if we had written \( M \) as \( \lambda(z: T).x + z \), and then performed the same substitution, we would obtain \( \lambda(z: T).y \ast y + z \). This is a function which takes a parameter and adds it to the square of the value of the free identifier \( y \).

Thus these two substitutions give quite different results, even though we think of \( \lambda(y: T).x + y \) and \( \lambda(z: T).x + z \) as being essentially the same function. Because the second result is the expected one, the restrictions on rule (5) do not allow the substitution of an expression which includes an identifier which can be “captured” by the formal parameter of the function.

However, this restriction in rule (4) leaves an important gap in the definition of substitution. Substitution of \( N \) into the body of a function with formal parameter \( y \) is not defined if it results in the capture of a free identifier \( y \) occurring in \( N \). However, we can handle this case by formalizing the intuition that functions which differ only in the names of formal parameters may be identified (i.e., treated as being identical).

We can write this identification more formally as

\[
\lambda(x: T).M \equiv \lambda(y: T).([y/x]M), \text{ if } y \notin F(I(M)).
\]

\(^2\) See a standard programming languages textbook for details.
Thus $\lambda(x: \text{Integer}).(\text{plus } x)\ x$ will be considered to be the same expression as $\lambda(y: \text{Integer}).(\text{plus } y)\ y$. In the literature these expressions are said to be $\alpha$-convertible, and the process of changing from the left side to the right is called $\alpha$-conversion.

The restriction on the new name of the formal parameter is again to avoid capture of free identifiers by the new parameter name. For example, the function $\lambda(x: \text{Integer}).(\text{plus } x)\ y$, where $y$ is the name of a free identifier, is not at all the same as $\lambda(y: \text{Integer}).(\text{plus } y)\ y$, the expression obtained by changing the formal parameter name from $x$ to $y$. The problem occurs because the first $y$ is now captured by the lambda binding of $y$ that occurs at the head of the expression. The first function adds the value of the parameter to a fixed value $y$, whereas the second doubles the value of the parameter.

The identification of expressions which differ only in the names of formal parameters can overcome the restriction in the next to last case in the definition of $[N/x] \ M$. We fix that case to be

$$5'. \ [N/x](\lambda(y: T). M) \triangleq \lambda(y': T). [N/x] M', \text{ if } y \neq x.$$  

where $y' \triangleq y$ and $M' \triangleq M$ if $y \not\in \text{Fl}(N)$. If $y \in \text{Fl}(N)$, then let $z$ be a new identifier not occurring in $M$ or $N$, and let $y' \triangleq z$ and $M' \triangleq [z/y] M$.

Thus if there is a danger that a free identifier of $N$ might be bound by making the substitution, we $\alpha$ convert the formal parameter name before making the substitution.

Now we can write down the reduction or evaluation rules for the lambda calculus. Each rule represents a single step in the evaluation of an expression. Rules are written as transformation rules where the expression to the left side of the double arrow is transformed to the expression on the right side. The label over the double arrow is the name of the rule.

**Definition 8.1.4** The reduction rules for the typed lambda calculus are given by

$$(\beta) \ (\lambda(x: T). M) N \xrightarrow{\beta} [N/x] M.$$  

$$(\eta) \ \lambda(x: T). (M \ x) \xrightarrow{\eta} M.$$  

The $\beta$ rule is the main computation rule for the lambda calculus. A function applied to an actual parameter results in evaluating the function body where all occurrences of the formal parameter have been replaced by the actual parameter (changing names of free identifiers as necessary to avoid name clashes). For example, $$(\lambda(x: \text{Integer}). (\text{plus } x)x)\ 17 \xrightarrow{\beta} (\text{plus } 17)\ 17.$$ The
\(\beta\) rule allows us to replace all free occurrences of \(x\) in the function body by 17.

The \(\eta\) rule is useful in minimizing clutter by reducing expressions formed by applying a function to an identifier and then using that identifier as a function parameter. For example if \(f = \lambda(x: \text{Integer}). (\text{plus } x)\), then \(f \xrightarrow{\beta} \text{plus}\). This should be plausible as applying \(f\) to a parameter \(M\) always results in \(\text{plus } M\) by the \(\beta\) rule.

Other rules (called \(\delta\) rules, and written with \(\text{red}\)) may be introduced to take care of evaluating constants and predefined functions (like evaluating arithmetic operations). For example, we could include the rule

\[(\text{plus } 17) \xrightarrow{\delta} 34\]

We can write \(M \rightarrow N\) when we don’t care to note which of the reduction rules is being applied. The notation \(M \rightarrow^* N\) indicates that \(N\) can be derived from \(M\) by the application of zero or more reduction steps.

In the lambda calculus one can apply a reduction rule to any subexpression of an expression. For example, if \(M\) is a lambda expression, then

\[M (((\lambda(x: \text{Integer}). (\text{plus } x) x) 17)\]

first reduces to

\[M (\text{plus } 17) 17\]

using \(\beta\)-reduction, and then to

\[M 34\]

using a \(\delta\)-rule.

The subject reduction theorem for the typed lambda calculus is an important result which implies that the type system is consistent with the computation rules. It states that reduction or evaluation of an expression results in a value with the same type as the original expression.

**Theorem 8.1.5** (Subject reduction) Let \(M\) be an expression of the typed lambda calculus such that \(E \vdash M : T\). If \(M \rightarrow^* M'\) then \(E \vdash M' : T\).

The proof of the subject reduction theorem is omitted here, but can be found in any text on the lambda calculus.

A simple consequence of the theorem is that if an expression has type \text{Integer}, then the subject reduction theorem implies that if reduction of the expression ever terminates, then the final value will be a value of type \text{Integer}. That is, it will be an integer.
Because the static type system should reflect important statically determinable properties of expression evaluation, a subject reduction theorem is generally required to show that the type system is reasonable. We will be interested in proving similar soundness properties of object-oriented languages by translating them into variants of the typed lambda calculus.

An expression is said to be in normal form if none of the reduction rules apply. All expressions of the pure typed lambda calculus with no constants or $\delta$ rules evaluate to a unique normal form. While this form is very convenient, readers who have studied the theory of effectively computable functions will recall that this property implies that the language is actually quite restricted in its expressiveness. We will increase the expressiveness of the typed lambda calculus in the following sections by adding new types and expressions.

We will need a much richer lambda calculus to model object-oriented languages. This richer calculus is needed even for languages like ML and Ada that support parametric polymorphism. In the next few sections we will define and provide the type-checking and computation rules for first an extended typed lambda calculus with pairs, records, and references. Then in Chapter 9 we introduce the polymorphic lambda calculus, an extension of the typed lambda calculus that includes both parametric polymorphism and "existential" types.

### 8.2 Adding pairs, sums, records, and references

In the last section we defined the simply-typed lambda calculus. This calculus included expressions defining functions and applying functions to arguments. While we can encode many language features in the pure lambda calculus, it is more convenient to extend the syntax to provide a closer match to real languages. In this section we define an extension of the lambda calculus, $\Lambda^+_r$, which includes tuples, records, sums, and references (variables). Ordered tuples are written in the form $(a_1, \ldots, a_n)$ and have type $T_1 \times \ldots \times T_n$, where each $T_i$ is the type of the corresponding $a_i$. Tuple types will be used to represent the domain of functions taking several parameters.

---

3. The relevant theorem is the following: For any effectively enumerable collection of total functions from integers to integers (i.e., functions defined on all integers), there is an effectively computable function not contained in that collection. Expressions of the typed lambda calculus represent total functions and can be effectively enumerated. Thus the theorem implies there is an effectively computable function not contained in the typed lambda calculus.
The projection operations, \( \text{proj}_i \), extract the \( i \)th component of a tuple. Thus \( \text{proj}_i(\langle a_1, \ldots, a_n \rangle) = a_i \).

Records are written in the form \( \{ l_1 : T_1 = M_1, \ldots, l_n : T_n = M_n \} \). Notice that each labeled field is provided with its type. The type of a record of this form is written as \( \{ l_1 : T_1 ; \ldots ; l_n : T_n \} \). Dot notation is used to extract the value of a field from a record: \( \{ l_1 : T_1 = M_1, \ldots, l_n : T_n = M_n \} . l_j = M_i \).

A sum type, \( T_1 + \ldots + T_n \), represents a disjoint union of the types, where each element contains information to indicate which summand it comes from, even if several of the \( T_i \) are identical. Suppose \( M \) is an expression from a type \( T_i \). The expression \( \text{in}_{T_i}^{T_1 ; \ldots ; T_n}(M) \) injects the value \( M \) into the \( i \)th component of the sum \( T_1 + \ldots + T_n \). If \( M \) is an expression of type \( T_1 + \ldots + T_n \), then an expression of the form

\[
\text{case } M \text{ of } x_1 : T_1 \text{ then } E_1 \mid \ldots \mid x_n : T_n \text{ then } E_n
\]

represents a statement listing the possible expressions to evaluate depending on which summand \( M \) is a part of. Thus if \( M \) was created by \( \text{in}_{T_i}^{T_1 ; \ldots ; T_n}(M') \) for some \( M' \) of type \( T_i \), then evaluating the case statement will result in evaluating \( E_i \) using \( M' \) as the value of \( x_i \).

The expression

\[
\text{in}_{T_1}^{\text{Integer}, \text{Integer} \to \text{Integer}}(47)
\]

is an expression with type \( \text{Integer} + (\text{Integer} \to \text{Integer}) \). It represents injecting the number 47 into the sum type.

The following function takes elements of the sum type, \( \text{Integer} + (\text{Integer} \to \text{Integer}) \), and uses case to compute an integer value. The value depends on whether the element of the sum type arose by injecting an integer or by injecting a function from integers to integers:

\[
\text{isFirst} = \lambda(y : \text{Integer} + (\text{Integer} \to \text{Integer})). \text{case } y \text{ of } \\
x : \text{Integer} \text{ then } x \times 2 \mid \\
f : \text{Integer} \to \text{Integer} \text{ then } f(0)
\]

In this example, the parameter \( y \) comes from a sum type, the first of whose summands is \( \text{Integer} \), while the second is the function type, \( \text{Integer} \to \text{Integer} \). If the parameter comes from the first summand, then it must represent an integer, \( x \), and one is added to the value. If it comes from the second summand, then it represents a function from integers to integers, denoted \( f \), and the function is applied to 0. This example shows how the value originally injected in the sum is represented by an identifier in the appropriate branch of the case, and thus can be used in determining the value to be returned.
Note that the case expression is a more secure alternative than providing expressions of the form \( \text{out}_i(M) \). The problem with using an \( \text{out}_i \) expression is that if, for example, a value was injected into the first summand, but \( \text{out}_i \), for some \( i \neq 1 \) is applied to the resulting value, an error has occurred. To preserve type safety, however, some default value of the appropriate type would have to be returned. The case expression appears better because all alternatives must be clearly specified and no default values are necessary.

Finally we introduce reference types in order to represent updatable variables. If \( M \) has type \( \text{Ref } T \), then we can think of it as denoting a location which can hold a value of type \( T \). The expression \( \text{val } M \) will denote the value stored at that location. In most procedural languages, programmers are not required to distinguish between variables (representing locations) and the values they denote. Instead, the compiler or interpreter automatically selects the appropriate attribute (location or l-value versus value or r-value) based on context without requiring the programmer to annotate the variable. The functional language ML (which supports references) does require the programmer to write \( !x \) when the value stored in \( x \) is required, while \( x \) alone always denotes the location.

The null expression is a special constant representing the null reference, a reference that does not point to anything. Evaluating the expression \( \text{val } \text{null} \) will always result in an error. If \( M \) is a reference, with type \( \text{Ref } T \), and \( N \) has type \( T \), then the expression \( M := N \) denotes the assignment of \( N \) to \( M \). That is, the value of \( N \) is stored in the location denoted by \( M \).

With that informal introduction to the extension, we are ready to define the set of types in \( \Lambda_r^+ \).

**Definition 8.2.1** Let \( \mathcal{L} \) be an infinite collection of labels, and let \( \mathcal{T} \mathcal{C} \) be a collection of type constants. The type expressions of \( \Lambda_r^+ \) are given by the following grammar:

\[
T \in \text{Type ::= } C \mid \text{Void} \mid T_1 \to T_2 \mid T_1 \times \ldots \times T_n \mid T_1 + \ldots + T_n \mid \mid l_i : T_1 ; \ldots ; l_n : T_n \mid \text{Ref } T \mid \text{Command}
\]

where \( l_i \in \mathcal{L} \), and, as before, \( C \in \mathcal{T} \mathcal{C} \) represents type constants (like Integer, Double, etc.). The type Void is the type of zero tuples. It will be used as the return type of commands or statements and as the parameter type for parameterless functions. It will have only one (trivial) value, \( \langle \rangle \). The other types represent function types, product (tuple) types, sum (or disjoint union) types, record types, and reference types (the types of variables), as discussed above. The final type, \( \text{Command} \), represents the type of statements, expressions like assignments that are evaluated simply for their side effects.
We next define the collection of pre-expressions of $\Lambda^r_r$. As before, they are pre-expressions, because the only valid expressions are those which type-check properly:

\[
M \in \mathcal{RLCE} ::= x \mid c \mid \langle \rangle \mid \lambda(x:T).M \mid M N \mid \langle M_1, \ldots, M_n \rangle \mid \text{proj}_i(M) \mid \\
\text{case } M \text{ of } x_1 : T_1 \text{ then } E_1 \ldots \mid x_n : T_n \text{ then } E_n \mid \\
\text{in}_1^{T_1} \ldots \text{in}_n^{T_n}(M) \mid \{ l_1 : T_1 = M_1, \ldots, l_n : T_n = M_n \} \mid M.l_i \mid \\
\text{ref } M \mid \text{null } \mid \text{val } M \mid \text{if } B \text{ then } \{ M \} \text{ else } \{ N \} \mid \\
\text{nop } \mid N := M \mid M; N
\]

In the above definition, $x \in \mathcal{E}I$, $c \in \mathcal{E}C$, and $l_i \in \mathcal{L}$. As noted above, the expression "\langle \rangle" represents a zero-tuple with type Void. The expressions involving function definition and application are familiar from the lambda calculus. The expressions for dealing with tuples, sums, records, and references were introduced above. The expression if $B$ then $\{ M \}$ else $\{ N \}$ represents the usual conditional statement, returning either the value of $M$ or $N$, depending on whether or not $B$ is true.

The last three expressions represent statements. The expression nop is a constant that represents a statement that has no effect. The expression $N := M$ represents an assignment statement, while $M; N$ indicates the sequencing of the two statements. Its intended meaning is similar to that of the language ML: do the first statement for the side effect and then return the value of the second. We can sequence as many expressions as we like. Most of the time, all of the expressions in the sequence will have type Command, in which case the expressions are evaluated as statements for their side effects, just as in Pascal, C, C++, and Java.

We could have added other expressions to the language such as various loop constructs; however it will be most convenient to work with a small language. We will add recursion later, which will provide the necessary expressive power to model real programming languages.

As before, the type-checking rules for these expressions require information about the types of free identifiers in an expression, provided through a static type environment, $\mathcal{E}$. The type-checking rules for expressions of $\Lambda^r_r$ are given in Figures 8.2 and 8.3.

As we observed earlier, these rules can be read algorithmically by starting with the bottom left side and proceeding clockwise around the rule. Thus to find the type of a pre-expression of the form $M.l_i$ with respect to a static type assignment $\mathcal{E}$, first find the type of $M$. If it is of the form $\{ l_1 : T_1 ; \ldots ; l_n : T_n \}$,
<table>
<thead>
<tr>
<th>Identifier</th>
<th>$\mathcal{E} \cup {x: T} \vdash x: T$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constant</td>
<td>$\mathcal{E} \vdash c: N$</td>
</tr>
<tr>
<td>Void</td>
<td>$\mathcal{E} \vdash \emptyset: \text{Void}$</td>
</tr>
<tr>
<td>Function</td>
<td>$\dfrac{\mathcal{E} \cup {x: S} \vdash M: T}{\mathcal{E} \vdash \lambda x: S. M: S \to T}$</td>
</tr>
<tr>
<td>Application</td>
<td>$\dfrac{\mathcal{E} \vdash M: S \to T, \mathcal{E} \vdash N: S}{\mathcal{E} \vdash MN: T}$</td>
</tr>
<tr>
<td>Tuple</td>
<td>$\dfrac{\mathcal{E} \vdash M_i: T_i, \text{ for all } 1 \leq i \leq n}{\mathcal{E} \vdash \langle M_1, \ldots, M_n \rangle: T_1 \times \ldots \times T_n}$</td>
</tr>
<tr>
<td>Projection</td>
<td>$\dfrac{\mathcal{E} \vdash M: T_1 \times \ldots \times T_n}{\mathcal{E} \vdash \text{proj}_i(M): T_i \text{ for all } 1 \leq i \leq n}$</td>
</tr>
<tr>
<td>Sum</td>
<td>$\dfrac{\mathcal{E} \vdash \text{in}_i^{T_1 + \ldots + T_n}(M): T_1 + \ldots + T_n}{\mathcal{E} \vdash M: T_1 + \ldots + T_n}$, $\mathcal{E} \cup {x_i: T_i} \vdash E_i: U_i \text{ for all } 1 \leq i \leq n$</td>
</tr>
<tr>
<td>Case</td>
<td>$\dfrac{\mathcal{E} \vdash \text{case } M \text{ of } x_1: T_1 \text{ then } E_1 \parallel \ldots \parallel x_n: T_n \text{ then } E_n: U}{\mathcal{E} \vdash {l_1: T_1; \ldots; l_n: T_n}: T_i \text{ for all } 1 \leq i \leq n}$</td>
</tr>
</tbody>
</table>

**Figure 8.2** Type-checking rules for $\Lambda^\to_{\text{cc}}$, part 1.
8.2 Adding pairs, sums, records, and references

Reference

\[ \varepsilon ⊢ M: T \]
\[ \varepsilon \vdash \text{ref } M: \text{Ref } T \]

Null

\[ \varepsilon \vdash \text{null}: \text{Ref } T, \text{ for any type } T \]

Value

\[ \varepsilon \vdash M: \text{Ref } T \]
\[ \varepsilon \vdash \text{val } M: T \]

No op

\[ \varepsilon \vdash \text{nop}: \text{Command} \]

Assignment

\[ \varepsilon \vdash N: \text{Ref } T, \quad \varepsilon \vdash M: T \]
\[ \varepsilon \vdash N := M: \text{Command} \]

Conditional

\[ \varepsilon \vdash B: \text{Boolean}, \quad \varepsilon \vdash M: T, \quad \varepsilon \vdash N: T \]
\[ \varepsilon \vdash \text{if } B \text{ then } \{ M \} \text{ else } \{ N \}: T \]

Sequencing

\[ \varepsilon \vdash M: S, \quad \varepsilon \vdash N: T \]
\[ \varepsilon \vdash M; N: T \]

Figure 8.3 Type-checking rules for \( \Lambda'_{\rightarrow} \), part 2.

then the type of \( M.l \) is \( T \). Otherwise no rule applies and the type-checking algorithm fails.

The case and if-then-else expressions require that the types of the branches all be the same type. This way a result of the same type is returned no matter which branch is selected.

If we have a sequence of two or more statements, the Sequencing rule indicates that the type of the sequence is the type of the last expression. Normally, all statements in the sequence will have type Command, but we also allow more general types.

The collection of expressions of \( \Lambda'_{\rightarrow} \) with respect to \( \varepsilon \) is the set of pre-expressions that can be assigned types with respect to the type-assignment axioms and rules.

We will sometimes find it convenient to use certain abbreviations in writing expressions of the lambda calculus. Two important abbreviations have to do with writing n-ary functions and let expressions. For each of these abbreviations, we will provide the corresponding derived typing rule.
We will find it convenient to write
\[ \lambda(id_1: T_1, \ldots, id_n: T_n). M \]
as an abbreviation for
\[ \lambda(\text{arg}: T_1 \times \ldots \times T_n). [\text{proj}_i(\text{arg})/id_i]_{i=1,\ldots,n} M. \]

Thus an \( n \)-ary function is an abbreviation for a function of a single argument that takes an \( n \)-tuple. When expanded, each of the individual parameters is replaced by an appropriate projection from the \( n \)-tuple. For example
\[ \lambda(x: \text{Integer}, y: \text{Integer}). \text{plus} \ x \ y \]
abbreviates
\[ \lambda(p: \text{Integer} \times \text{Integer}). \text{plus} \ (\text{proj}_1(p)) \ (\text{proj}_2(p)) \]

The derived typing rule for \( n \)-ary functions is
\[
\begin{align*}
\text{n-ary Function} & \quad \mathcal{E} \cup \{id_1: T_1, \ldots, id_n: T_n\} \vdash M: U \\
& \quad \therefore \mathcal{E} \vdash \lambda(id_1: T_1, \ldots, id_n: T_n). M: T_1 \times \ldots \times T_n \rightarrow U
\end{align*}
\]

We will write \( \text{let} \ x:T = M \text{ in } N \text{ end} \) as an abbreviation for \( (\lambda(x: T). N) \ M \). Thus, introducing an identifier for an expression is modelled by writing a function with that identifier as the parameter, and then applying the function to the intended value for the identifier. The \( \beta \)-rule verifies our intuition that let-expressions allow us to use the identifier to stand for the appropriate expression.

We have the following derived typing rule for let expressions:
\[
\begin{align*}
\text{Let} & \quad \mathcal{E} \cup \{x: T\} \vdash N: S, \quad \mathcal{E} \vdash M: T \\
& \quad \therefore \mathcal{E} \vdash \text{let} \ x:T = M \text{ in } N \text{ end}: S
\end{align*}
\]

We leave it as an exercise for the reader to verify that the typing rules for \( n \)-ary functions and let expressions are derivable from the other typing rules when the abbreviations are expanded.

Along with the \( \beta \) and \( \eta \) conversion rules from the last section, there are now similar rules for expressions of the new types introduced here. Some of these rules can be found in Figure 8.4.

At this point we warn the reader that, because of the addition of assignment statements, the reduction rules provided in the figure are not sufficient
8.2 Adding pairs, sums, records, and references

\[ \lambda (x: T). (M \ x) \rightarrow M \]

\[ \text{proj}_i ((M_1, \ldots, M_n)) \rightarrow M_i \]

\[ \text{case (inj}_{T_1}, \ldots, T_n(M)) \text{ of } x_1 : T_1 \text{ then } E_1 \ | \ \ldots \ | \ x_n : T_n \text{ then } E_n \rightarrow \]

\[ [M/x_i] E_i \text{, for } 1 \leq i \leq n. \]

\[ (\text{val (ref } M)) \rightarrow M \]

\[ \text{true if true then } M \text{ else } N \rightarrow M \]

\[ \text{false if false then } M \text{ else } N \rightarrow M \]

\[ (x := M; \text{val } x) \rightarrow M \]

\[ \text{let } x : \text{Ref Integer} = \text{ref } 5 \text{ in} \]
\[ \text{let } y : \text{Ref Integer} = \text{ref } 2 \text{ in} \]
\[ x := \text{val } x + 2; y := \text{val } x - 2; \text{val } x \]
\[ \text{end} \]
\[ \text{end} \]

Figure 8.4 Computation rules for \( \Lambda^{+}_{\pi}. \)

to perform the computations that one would normally expect an interpreter for this language to perform. The problem is that we have not provided an explicit representation of memory. Consider the following example

\[ \text{let } x : \text{Ref Integer} = \text{ref } 5 \text{ in} \]
\[ \text{let } y : \text{Ref Integer} = \text{ref } 2 \text{ in} \]
\[ x := \text{val } x + 2; y := \text{val } x - 2; \text{val } x \]
\[ \text{end} \]
\[ \text{end} \]

where we use the \textit{let} clause abbreviation just introduced. None of the computation rules apply to the sequence of expressions in the body of the \textit{let} clauses because the assignment to \( y \) is between the assignment to \( x \) and the \textit{val } \textit{x} expression. Yet it is clear that the entire expression should evaluate to 6.

If we were interested in writing an interpreter, we could write computation rules that are based on taking the triple of an expression, an environment representing the current values of identifiers (where the value of a variable would be a location in memory), and an expression representing the values held in memory. Evaluating an expression would result in a pair consisting
of the value of the expression and the new memory configuration obtained after evaluating the expression.

Because the intuition behind the evaluation of such expressions should be clear, we do not bother to write down the formal computation rules, but instead leave that as an exercise for the reader. Thus the computation rules provided so far, as well as those to come, should be treated as a very incomplete set of computation rules that are provided only to give intuition for the meaning of expressions.

8.3 Summary

In this chapter we introduced formal notations for describing the syntax of a language and a set of type-checking rules. We applied this notation to describe the typed lambda calculus, including extensions to add tuples, sums, records, and references (variables).

We will use the lambda calculus to describe the meaning of the constructs of object-oriented languages. However, we need more advanced features in order to describe some of the special features of object-oriented languages like objects, classes, subclasses, and information hiding. We introduce the polymorphic lambda calculus in the next chapter in order to more accurately model these features. We also introduce mechanisms to define recursive functions and types.
In the previous chapter we introduced a version of the lambda calculus with type and value expressions similar to those in most common functional programming languages, including simple imperative features. We wish to use the lambda calculus to explain object-oriented language constructs. However, the version of the lambda calculus introduced so far is not adequate to model these constructs. To model these features it will be necessary to add to the lambda calculus expressions supporting parametric polymorphism, existential types (to model information hiding), and the definition of recursive functions and types.

The polymorphic lambda calculus, $\Lambda^p$, extends the typed lambda calculus with pairs, records, and references, $\Lambda^{\tau_\pi}$, defined in the previous section. It adds functions that are parameterized over types, as well as a mechanism to hide type information in an expression. We begin with parametric polymorphism, and then add support for recursion and information hiding. Finally we extend the polymorphic lambda calculus to $\Lambda^p_{\leq}$, which supports subtyping and bounded polymorphism.

### 9.1 Parameterized types and polymorphism

Polymorphic functions are functions that operate uniformly on values of different types. These functions typically are defined to work with parameterized types, like $\text{Stack}(T)$, $\text{Tree}(T)$, $\text{Array of } T$, etc. As can be seen from these examples, parameterized types typically represent structured collections of values. The parameter $T$ typically indicates the type of values stored in the structure.

We begin by examining parameterized types. Parameterized types can best be modeled as functions from types to types. But rather than restricting our-
selves only to functions from types to types, we will add higher-order functions on types as well. These will be classified by a kind structure that is, in many ways, quite similar to the types that categorize expressions. However, the kinds will categorize functions from types to types.

The simplest kind is $\star$, representing the kind of all types, while more complex kinds are built up by defining functions from kinds to kinds.

$$\kappa \in \text{Kind} ::= \star | \kappa \Rightarrow \kappa'$$

The kind $\star \Rightarrow \star$ thus represents functions from $\star$ to $\star$, like Tree and Stack. We use the double arrow for function kinds to distinguish them from function types. Higher kinds represent more complex functions involving types. For example, $(\star \Rightarrow \star) \Rightarrow \star$ represents the kind of functions that take a function from types to types and return a type. We will presume that function kinds, like types, associate from the right. That is, $\star \Rightarrow \star \Rightarrow \star$ will be treated as being the same as $\star \Rightarrow (\star \Rightarrow \star)$.

The elements that are classified by kinds are called type constructors or simply constructors. Let $CT$ be a collection of constructor identifiers, each labelled with its kind, where $CT$ contains an infinite number of identifiers for each kind. Let $CC$ be a collection of constructor constants, where again each element is associated with a constructor.

**Definition 9.1.1** Let $C$ be an infinite collection of labels, and let $CC$ be a collection of type constants. The type constructor pre-expressions of $\Lambda^P_C$, are given by the following grammar.

$$\mu, \nu \in \text{Constructor} ::= v^\kappa | c^\kappa | \text{Void} | \mu \rightarrow \nu | \mu_1 \times \ldots \times \mu_n | \mu_1 + \ldots + \mu_n | \{l_1; \mu_1; \ldots; l_n; \mu_n\} | \text{Ref} \mu | \text{Command} | \lambda v^\kappa. \mu | \mu \nu$$

where $v^\kappa \in CT$, $c^\kappa \in CC$, and $l_i \in C$.

The constructor pre-expressions include all of the type expressions introduced in the previous chapter, but include more expressions dealing with constructors of higher kinds. The constructor identifiers are labeled with their kinds. Thus $v^\kappa$ is an identifier representing types, while $v^\kappa \Rightarrow \star$ is an identifier representing a function from types to types. For notational simplicity, we will usually omit the $\star$ as a superscript for type identifiers, and simply write $t$, $u$, $t'$, or $l_i$. We will do the same for constructor constants of kind $\star$ (i.e., type constants).

If $CT$ is non-empty, the system includes type constructor constants. For example rather than thinking of $\rightarrow$ and $\times$ as operators, we could consider
them as type constructor constants, each of which has kind $\star \Rightarrow \star \Rightarrow \star$. However, we will prefer to continue to treat them as type operators.

Type constructor pre-expressions of kind $\star$ (i.e., type expressions) can be built up as in the last chapter to form function, record, sum, product, and reference types. The type constructor pre-expressions also include the types Void and Command.

Functions from kinds to kinds are written in the form $\lambda \sigma. \mu$, where $\mu$ is a type constructor. The application of a constructor function to an argument of the appropriate kind is written in the form $\mu \nu$, where $\mu$ and $\nu$ are constructors.

We will write kind assignments to constructors in the form "$\mu::\kappa". kind assignments do not depend on "static kind environments", unlike the type-checking rules that depended on static type environments. The reason is that all of the constructor identifiers are provided with kinds. kind checking is straightforward. The rules can be found in Figure 9.1. The type constructor expressions are the type constructor pre-expressions that can be assigned a kind by the kinding rules.

Constructor identifiers are assigned the kind in their label. Void and Command have kind $\star$. That is, they are types. The kinding rules assign kind $\star$ to expressions built up from types using $\rightarrow$, sums, products, records, and reference types. Notice these expressions are not well-kinded if any of the components fail to have kind $\star$.

Function expressions result in higher kinds, while applications of constructors of kind $\mu \Rightarrow \nu$ to type constructors of kind $\mu$ result in constructors of kind $\nu$.

This figure also includes a “congruence rule” for simplifying constructor applications. A congruence rule indicates when one object may be treated as being the same as another. In particular, if two constructors are congruent, then if one is substituted for the other in any constructor then the new constructor will be congruent with the old. In combination with a forthcoming typing rule, this will allow us to simplify constructor applications appearing in expressions to be type-checked.

A simple example of a constructor with kind $\star \Rightarrow \star$ is $\text{Pair} = \lambda t. (t \times t)$. (Recall that we usually drop the kind from type variables.) This is a function that takes a type as an argument and returns the product of the type with itself. Thus, $\text{Pair}(\text{Integer}) = \text{Integer} \times \text{Integer}$ and $\text{Pair}(\text{Boolean}) = \text{Boolean} \times \text{Boolean}$.

$\text{AppPair} = \lambda t. \lambda u.((t \times (t \Rightarrow u)) \Rightarrow u)$ is an example of a constructor with kind $\star \Rightarrow \star \Rightarrow \star$. This constructor can be applied to two successive type
\[\begin{array}{l}
c^\varepsilon:: \kappa \\
\varnothing^\varepsilon:: \kappa \\
\text{Void}:: \ast \\
\text{Command}:: \ast \\
\frac{\mu:: \ast \quad \nu:: \ast}{\mu \to \nu:: \ast} \\
\frac{\mu_1:: \ast, \ldots, \mu_n:: \ast}{\mu_1 \times \ldots \times \mu_n:: \ast} \\
\frac{\mu_1:: \ast, \ldots, \mu_n:: \ast}{\mu_1 + \ldots + \mu_n:: \ast} \\
\frac{\mu_1:: \ast, \ldots, \mu_n:: \ast}{\{l_1:: \mu_1; \ldots; l_n:: \mu_n\}:: \ast} \\
\frac{\mu:: \ast}{\text{Ref}\mu:: \ast} \\
\frac{\mu:: \kappa'}{\lambda\nu^\kappa.\mu:: \kappa \Rightarrow \kappa'} \\
\frac{\mu:: \kappa \Rightarrow \kappa', \nu:: \kappa}{\mu\nu:: \kappa'} \\
\frac{\nu:: \kappa}{(\lambda\nu^\varepsilon.\mu)\nu \equiv [\nu/\varepsilon]\mu}
\end{array}\]

Figure 9.1  Kind checking and equivalence rules.
9.1 Parameterized types and polymorphism

expressions. For example, \( \text{AppPair}(\text{Integer})(\text{Boolean}) = ((\text{Integer} \times (\text{Integer} \rightarrow \text{Boolean})) \rightarrow \text{Boolean}) \).

Because we will be adding constructs over the next few sections, we will not bother to repeat in our new definitions the constructs introduced in previous sections of this chapter. Instead we will indicate their omission by the use of “...”.

In order to introduce polymorphic functions, we first enrich our type system by adding an expression representing the type of a polymorphic function. This will be added as a type constructor with kind \( \kappa \).

\[
\mu, \nu \in \text{Constructor} ::= \ldots \mid \forall \nu^\kappa. T
\]

The constructor expression \( \forall \nu^\kappa. T \), where \( \nu^\kappa \) may occur in \( T \), represents the type of a function that takes a type constructor with kind \( \kappa \) and returns a value of type \( T \). Because of the use of the universal quantifier in the notation for polymorphic types, these are called universal types.

The kinding rule for this constructor is:

\[
\frac{T ::= \star}{\forall \nu^\kappa. T ::= \star}
\]

For example, a polymorphic function that takes a type identifier, \( t \), and then a value, \( x \), of type \( t \), returning the pair \( \langle x, x \rangle \) will have type \( \forall t. t \rightarrow \text{Pair}(t) \), where \( \text{Pair} \) is the constructor introduced earlier.

In order to avoid notational overhead, from now on we will identify types with type constructors of kind \( \kappa \), treating the type constructors as containing all of the types in \( \Lambda^\kappa_{胎} \). Similarly the expressions of the polymorphic lambda calculus, \( \Lambda^P_{胎} \), include all expressions of \( \Lambda^\kappa_{胎} \).

We now add two new pre-expressions to our language to represent polymorphic function construction and polymorphic function application:

\[
M \in \mathcal{P}_{胎} ::= \ldots \mid \forall \nu^\kappa. M \mid M[\mu]
\]

The notation \( \mathcal{P}_{胎} \) stands for the set of polymorphic lambda calculus (pre-)expressions.

The first expression in the definition is a polymorphic function taking a parameter of kind \( \kappa \) (we use capital \( \Lambda \) to distinguish polymorphic functions that take types to values from constructor functions that take types to types). The second represents the application of a polymorphic function to a constructor parameter of the appropriate kind. As above, when the kind of the
Continuing our example above, we write the polymorphic function that
takes a value of type \( t \) and returns a pair with that value duplicated as fol-
" "

\[ \text{polyPair} = \lambda t. \lambda x. t. \langle x, x \rangle \]

The reader should be sure to understand why the \( \lambda t \) is used in this def-
nition (to indicate a polymorphic function that returns a value) and the \( \lambda t \)
was used in the earlier definition of \( \text{Pair} \) (a function that returns a type rather
than a value).

The type-checking rules for these new expressions are given in Figure 9.2.
The first two rules are similar to those for regular functions. The main twist
is that, after the application of a polymorphic function, \( M \), to a constructor
expression, \( \mu \), the type of the result is obtained by replacing all occurrences
of the type identifier \( t \) in the polymorphic type by \( \mu \).

**Exercise 9.1.2** Extend Definition 8.1.1 to define the set of free identifiers (regular
and type constructor) for an expression \( M \).

**Exercise 9.1.3** Extend Definition 8.1.3 and define substitution for type constructor
identifiers.

Using type-checking rule \( \text{PolyFunction} \) we can show the type of the expres-
sion \( \text{polyPair} \) defined above is \( \forall t. (t \to t \times t) \). By the \( \text{PolyApplication} \) typing
rule, the type of \( \text{polyPair} [\text{Integer}] \) will be \( \text{Integer} \to \text{Integer} \times \text{Integer} \).

We also add a **Congruence** rule that ensures that if an expression has a type
\( S \), then it can also be assigned any other type congruent to \( S \).
As before, we will find it convenient to use let expressions as abbreviations. We will write \( \text{let } \nu^\kappa = \mu \text{ in } M \text{ end} \) as an abbreviation for \([\nu^\kappa / \mu] M\). The derived typing rule for this let expression on types is:

\[
\frac{\mathcal{E} \vdash (\nu^\kappa / \mu) M : S \quad \mu : : \kappa}{\mathcal{E} \vdash \text{let } \nu^\kappa = \mu \text{ in } M \text{ end} : S}
\]

We are unable to treat this as an abbreviation for a type application because the type information will generally be used in typing the rest of the expression. For example, the expression

\[\text{let } t = \text{Integer in } x = 7 \text{ in } x + 2 \text{ end end}\]

may only be type checked properly if it is known that \(t\) stands for \text{Integer} at the time the declaration is checked. This was not necessary for regular let expressions as the value was not needed in type-checking.

Again we will identify type constructors and polymorphic functions that differ only by the name of the formal parameter. Rewriting rules analogous to \(\beta\) and \(\eta\) are added to the calculus.

\[
\pi\beta \quad (\Lambda \nu^\kappa. M)[\mu] \rightarrow (\mu/\nu^\kappa)M, \text{ if no free identifier of } \mu \text{ is bound in } M.
\]

\[
\pi\eta \quad \Lambda \nu^\kappa. (M[\nu^\kappa]) \rightarrow M
\]

In Section 9.4 we further enrich the polymorphic lambda calculus to support bounded polymorphism. However, we will first discuss recursion and existential types.

### 9.2 Recursive expressions and types

Recursive functions and procedures are supported in almost all modern programming languages. Just as we have provided \(\lambda\)-notation to provide the description of anonymous functions, we would like to do the same for recursive functions. This seems difficult since the essence of a recursive definition is that the definition uses the name of the function. However, there is a standard way of working around this difficulty using what are known as fixed-point operators.

A common example of a recursively defined function is the factorial function:

\[\text{fact}(n: \text{Integer}) \triangleq \text{if } n = 0 \text{ then } 1 \text{ else } n * \text{fact}(n-1)\]
or, more accurately,

\[ \text{fact} \overset{\Delta}{=} \lambda(n: \text{Integer}). \text{if } n = 0 \text{ then } 1 \text{ else } n \times \text{fact}(n-1) \]

If we think of the above as an equation to be solved for \text{fact}, it is apparent that we need to find a function, \( f \), that, when substituted for \text{fact} on the right side of the equation, results in exactly the same function as \( f \). We can formalize this by writing the following higher-order function definition:

\[ G \overset{\Delta}{=} \lambda(f: \text{Integer} \to \text{Integer}). \lambda(n: \text{Integer}). \text{if } n = 0 \text{ then } 1 \text{ else } n \times f(n-1) \]

We can apply \( G \) to any function from integers to integers. For example, \( G(\lambda(n: \text{Integer}). 0) \) results in a function that returns \( 1 \) when applied to \( 0 \) and \( 2 \) for all other input values. (Try it!)

However, if \text{fact} is defined above, then

\[ G(\text{fact}) = \lambda(n: \text{Integer}). \text{if } n = 0 \text{ then } 1 \text{ else } n \times \text{fact}(n-1) = \text{fact} \]

Thus we can change our point of view and say that what we are really looking for is a function \( f_0 \) such that \( G(f_0) = f_0 \).

An expression \( e \) is a fixed point of a function \( G \) if \( G(e) = e \). Because \( G \) applied to \text{fact} returns itself, \text{fact} is a fixed point of \( G \). We call \( G \) a generator of the fixed point.

We can use the information that \text{fact} is a fixed point of \( G \) in order to compute the value when \text{fact} is applied to a number. The key is that any occurrence of \text{fact} may be replaced by \( G(\text{fact}) \). Thus we compute

\[
\begin{align*}
\text{fact}(2) & = G(\text{fact})(2) \\
& = (\lambda(n: \text{Integer}). \text{if } n = 0 \text{ then } 1 \text{ else } n \times \text{fact}(n-1))(2) \\
& = \text{if } 2 = 0 \text{ then } 1 \text{ else } 2 \times \text{fact}(2-1) \\
& = \text{if false then } 1 \text{ else } 2 \times \text{fact}(2-1) \\
& = 2 \times \text{fact}(1) \\
& = 2 \times G(\text{fact})(1) \\
& = 2 \times (\text{if } 1 = 0 \text{ then } 1 \text{ else } 1 \times \text{fact}(1-1)) \\
& = 2 \times (\text{if false then } 1 \text{ else } 1 \times \text{fact}(1-1)) \\
& = 2 \times (1 \times \text{fact}(0)) \\
& = 2 \times (1 \times G(\text{fact})(0)) \\
& = 2 \times (1 \times (\text{if } 0 = 0 \text{ then } 1 \text{ else } 0 \times \text{fact}(0-1))) \\
& = 2 \times (1 \times (\text{if true then } 1 \text{ else } 0 \times \text{fact}(0-1))) \\
& = 2 \times (1 \times 1) \\
& = 2
\end{align*}
\]
We can generalize this construction so that, given a recursive definition of a function \( f \), we can define a higher-order function like \( G \) above, so that \( G(f) = f \). We can then use the fact that \( f \) is a fixed point to compute the values of the function.

Unfortunately, there are some recursive function definitions that do not result in values when expanded as above. For example, the recursive function definitions:

\[
\begin{align*}
    f & \triangleq \lambda (n: \text{Integer}). f(n) + 1 \\
    g & \triangleq \lambda (n: \text{Integer}). g(n + 1)
\end{align*}
\]

are everywhere undefined, though for very different reasons.

These non-convergent function definitions will be handled by introducing partial functions. We interpret the function type \( T \rightarrow U \) as the set of partial functions from \( T \) to \( U \). That is, they represent the functions whose domain is a subset of \( T \).

We introduce a special constant \( \text{fix} \) to represent a polymorphic fixed point operator. It is applied first to a type \( T \), and then to a higher-order function, \( H \), with type \( T \rightarrow T \). The result will be a value of type \( T \) that is a fixed point of \( H \).

Thus in the above example defining the factorial function,

\[
\text{fact} \triangleq \text{fix}[\text{Integer} \rightarrow \text{Integer}](G)
\]

results in a function of type \( \text{Integer} \rightarrow \text{Integer} \).

The corresponding typing rule is:

\[
\text{FixElt} \quad \emptyset \vdash \text{fix} \forall t. (t \rightarrow t) \rightarrow t
\]

In this rule, \( t \) represents the type of the function being defined, so is generally itself a function type.

We also add the following computation rule. If \( G \) has type \( t \rightarrow t \), then:

\[
\mu \text{Elt} \quad \text{fix}[t](G) \rightarrow G(\text{fix}[t]G)
\]

This is exactly the computation rule used earlier in order to compute \( \text{fact}(2) \).

1. Though we won’t provide the details here, we intend \( \text{fix} \) to provide what is usually called the least fixed point, as that will correspond to the computation rule.
recursive definition. As in the example above, unrolling the definition typically results in an expression headed by a lambda that can be used in a $\beta$ reduction to make progress in the computation.

Many of the types of interest to programmers are also defined recursively. A common example is lists, where the definition of a list of integers might be given as

$$IntList \triangleq Void + Integer \times IntList$$

That is, an element from $IntList$ is either an empty tuple or is the pair of an integer with an element from $IntList$. Examples of elements of this type include $()$, $\langle 17, () \rangle$, $\langle 2, \langle 17, () \rangle \rangle$, and $\langle 47, \langle 2, \langle 17, () \rangle \rangle \rangle$.

We can also define parameterized recursive types using functions from types to types:

$$List(t) = Void + t \times List(t)$$

or, more accurately,

$$List \triangleq \lambda t.\ (Void + t \times List(t))$$

We introduce a new operator, $\text{Fix}$, on type constructors in order to create fixed points of functions from kinds to kinds, just as $\text{fix}$ creates fixed points of regular functions. We could make $\text{Fix}$ polymorphic on kinds, but that would require us to make our kind language more complex. We will avoid that by treating $\text{Fix}$ similarly to $\text{Fix}$, that is, as an operator used in creating new constructor expressions. The kinding rule for $\text{Fix}$ is as follows:

$$\frac{\mu : \kappa \Rightarrow \kappa}{\text{Fix}(\mu) : \kappa}$$

Thus if

$$\nu = \lambda (IL : \ast).\ (Void + Integer \times IL)$$

then $\text{Fix}(\nu)$ is a type that will be equivalent to the recursively defined type $IntList$ above.

We can define the parameterized type function $List$ similarly by defining

$$\nu' = \lambda (L : \ast \Rightarrow \ast).\ \lambda t.\ (Void + t \times L(t))$$

and letting $List = \text{Fix}(\nu')$.

As with regular recursive functions, we can also unfold recursively defined types. We will treat recursively defined type constructors (i.e., those defined using $\text{Fix}$) as being congruent to their unfoldings:

$$\frac{\mu : \kappa \Rightarrow \kappa}{\text{Fix}(\mu) \cong \mu(\text{Fix}(\mu))}$$
While we do not bother to write the explicit rules here, we treat \( \cong \) as a true congruence relation, so that if we replace a constructor \( \mu \) in \( \nu \) by \( \mu' \cong \mu \) to obtain \( \nu' \), then \( \nu' \cong \nu \).

The Congruence typing rule from Figure 9.2 allows us to replace recursive types in the type-checking rules by their foldings or unfoldings.

We will now use recursive definitions freely when working with the polymorphic lambda calculus, with the understanding that we can formalize all definitions as shown in this section.

### 9.3 Information hiding and existential types

Sometimes we wish to use abstraction to hide information outside of a particular context. This is one of the main reasons for defining modules in programming languages, but information hiding is also of great importance in the use of object-oriented languages. For example, we will generally want to restrict instance variables from being visible outside of an object.

A traditional example of information hiding is when we want to have defined expressions and types treated as though they are built-in primitives. For example, it is possible to define a type and operations in the polymorphic lambda calculus that behave like the booleans.

Expressions representing true and false can be defined as follows:

\[
\text{True} \triangleq \lambda t. \lambda y: t. \lambda n: t. y \\
\text{False} \triangleq \lambda t. \lambda y: t. \lambda n: t. n
\]

They both have type \( \forall t. t \rightarrow t \rightarrow t \), a type we can think of as an abbreviation for Boolean.

Conditional expressions can be defined as follows:

\[
\text{IfThenElse} \triangleq \lambda u. \lambda b: \forall t. t \rightarrow t \rightarrow t. \lambda (M: u). \lambda (N: u). b[u] M N
\]

with type \( \forall u. (\forall t. t \rightarrow t \rightarrow t) \rightarrow u \rightarrow u \rightarrow u \). The type parameter \( u \) in the definition of \( \text{IfThenElse} \) determines the result type of the conditional expression.

With these definitions, it is easy to see that if \( M \) and \( N \) both have type \( T \), then \( \text{IfThenElse}[T] \text{True} M N \rightarrow M \) and \( \text{IfThenElse}[T] \text{False} M N \rightarrow N \).

**Exercise 9.3.1** Define \( \text{not} \) in the polymorphic lambda calculus. Show that \( \text{not}(\text{True}) \) reduces to \( \text{False} \) and that \( \text{not}(\text{False}) \) reduces to \( \text{True} \). Define \( \text{and} \) and \( \text{or} \) and show that they behave appropriately. All three of these are most easily defined using the definitions of \( \text{True} \), \( \text{False} \), and \( \text{IfThenElse} \).
Exercise 9.3.2 Define expressions in the polymorphic lambda calculus for each of the integers, \( n \) as

\[
\mathit{\lambda} \triangleq \forall t. \lambda(f: t \to t). \lambda(x: t). f^{(n)}(x)
\]

where \( f^{(n)} \) represents applying the function \( f \) a total of \( n \) times. What is the type of integers? Write functions \( \text{Plus} \) and \( \text{Mult} \), and show they have the appropriate behaviors when applied to encodings of integers. Subtraction is much more difficult to define, and requires a very clever encoding.

Because all of these expressions are definable, and expressions for the integers as well, we could go ahead and use them informally, knowing that the informal expressions can be replaced by the corresponding formal expressions. However, we have earlier added the types \( \text{Integer} \) and \( \text{Boolean} \), as well as an \( \text{if-then-else} \) expression to our language. Why did we do introduce these as constants and operators if it is possible to define these expressions?

One reason for introducing these as primitives rather than just using the expressions is that it is painful, for example, to have to write down the type \( \forall t. t \to t \to t \) whenever we have a boolean parameter or variable. We would prefer to be able to just write down a name \( \text{Boolean} \), and not have to worry about the actual definition. Similarly for the expressions representing \( \text{True} \), \( \text{False} \), and conditionals.

One solution to this problem is to introduce existential types, which will allow us to hide certain types and definitions, while providing sufficient typing information that we can use the expressions and types. For instance we can write the type

\[
\text{BoolSpec} \triangleq \exists \text{Bool. } \exists \text{Bool. } \exists \text{Bool. } \exists \text{Bool. } \exists \text{Bool. } \exists \text{Bool. }
\]

that provides the specification of a package representing booleans. The specification includes a hidden type \( \text{Bool} \), exported features \( \text{True} \) and \( \text{False} \) with type \( \text{Bool} \), and polymorphic operation \( \text{IfThenElse} \) that takes a type parameter \( u \), a value of type \( \text{Bool} \), two values of type \( u \), and returns a value of type \( u \).

The idea is that users should be able to see this specification and understand how to use the components of a package satisfying the specification without knowing what the implementation type, \( \text{Bool} \), of the package really is. (Of course, the typing information only provides information on how the elements of this specification can be used without generating type errors. Further semantic specifications would need to be given to ensure that the components behave as one would expect with booleans.)
We should be able to package up the definitions of the boolean expressions given above to satisfy \( \text{BoolSpec} \) if we let the implementation of \( \text{Bool} \) be the type \( \forall t. t \rightarrow t \rightarrow t \). After all, if we let \( \text{Bool} \) stand for that type then the earlier definitions of \( \text{True}, \text{False}, \text{and IfThenElse} \) all have the types specified in \( \text{BoolSpec} \).

We could just put all of these definitions in a record, but then the user could take advantage of the details of the implementation to perform operations we had not intended. For example \( \text{True} \) could be treated as a function and applied to \( \text{Integer}, 7, \text{and 15} \). Another problem with making the implementations of these expressions available to the programmer is that we might later want to change the implementations, for example to get more efficiency or extra behaviors. If the user took advantage of the implementation of these expressions then changing the implementation would likely cause problems throughout the code that was using it.

The reasons just given are the classic reasons for the introduction of abstract data types (ADT’s) in programming languages. The formal mechanism for supporting this kind of information hiding is the introduction of existential types.

Thus we add existential type expressions to our collection of constructor expressions:

\[
\mu, \nu \in \text{Constructor} ::= \ldots \mid \exists \nu^e. T
\]

The constructor expression \( \exists \nu^e. T \), where \( \nu^e \) may occur in \( T \), represents the type of a package specification that hides the representation of a type constructor (represented by \( \nu^e \)) and exports components with type \( T \). As usual, we do not bother to write a superscript on constructor identifiers with kind \( * \).

The kinding rule for this constructor is:

\[
\frac{T :: *}{\exists \nu^e. T :: *}
\]

The following expressions provide operations to pack up and open existentials:

\[
M \in \mathcal{P} \cdot \mathcal{L} \cdot \mathcal{E} ::= \ldots \mid \text{pack} \langle \mu, M \rangle \text{ as } \exists \nu^e. T \mid \text{open } M \text{ as } \langle \nu^e, x \rangle \text{ in } N
\]

The first expression takes a value, \( M \), and creates a packed version of existential type by hiding certain occurrences of construction expression, \( \mu \), and
replacing them with the constructor identifier, \( v^e \). After replacing appropriate occurrences of \( \mu \) in the type of \( M \) by \( v^e \), the resulting type is given by \( T \).

The second expression “opens” an existential, revealing that it is composed of a pair of a type constructor and value, and uses the given type and value names in the expression \( N \). While \textit{open} provides names \( v^e \) and \( x \) for the hidden type and the value abstracted, it does not provide any information about the actual representation of the type or expression.

Returning to our example of creating a package representing boolean values, we can use the pack expression along with the definitions in the previous subsection to define

\[
\text{BoolPack} \triangleq \text{pack} \left\langle \forall t. t \rightarrow t \rightarrow t,\right.
\]

\[
\{ \begin{align*}
\text{True} &= \Lambda t. \lambda(y: t). \lambda(n: t). y \\
\text{False} &= \Lambda t. \lambda(y: t). \lambda(n: t). n \\
\text{IfThenElse} &= \Lambda u. \lambda(b: \forall t. t \rightarrow t \rightarrow t). \\
& \quad \lambda(M: u). \lambda(N: u). b[u]M N
\end{align*}\}
\]

\[
\right\rangle
\]

as \textit{BoolSpec},

where we omitted the type information in the record of values to reduce the clutter in the expression. All of the details of the implementation of the type \textit{Boolean} are now hidden inside \textit{BoolPack}, whose type will be \textit{BoolSpec}. The pack expression takes the type \textit{Boolean} and the record of implementations of the values and implementations and converts them to a value of existential type. The components are no longer directly accessible now that they have been packed.

We can use \textit{BoolPack} as follows:

```plaintext
open BoolPack as \langle \text{Bool}, \text{BOps} \rangle in
let bval = \text{BOps.True}
\ldots \text{BOps.IfThenElse}[\text{Integer}](bval)(\text{3})(17) \ldots
end
```

Thus we only get access by opening the package and referring to the components through the record whose name, \textit{BOps}, and type, \textit{Bool} are provided in the open expression. We know \textit{Bops.IfThenElse} is legal because the existential type \textit{BoolSpec} guarantees that the implementation has a field \textit{IfThenElse} with type \( \forall u. \text{Bool} \rightarrow u \rightarrow u \rightarrow u \). As a result we are guaranteed that the type of \textit{BOps.IfThenElse}[\text{Integer}](bval)(3)(17) is \textit{Integer}. 


9.3 Information hiding and existential types

The typing rules in Figure 9.3 are a bit more complex than the earlier ones. The first rule is relatively straightforward. One can simply replace as many occurrences of $\mu$ as one would like in the type of $M$ by a constructor identifier, and report the packed version of the element as being of an existential type.

The body of the $\text{pack}$ is type checked using the knowledge of the constructor, $\mu$, but the final type uses the constructor identifier. Clearly $\text{BoolPack}$ has type $\text{BoolSpec}$ by this rule.

The second rule is a bit more subtle. The expression $M$ is of an existential type, so it has been packed as a pair. The $\text{open}$ expression reveals the pair that can be used in $N$. However, statically $N$ doesn’t have access to the exact type of the value $x$ in the pair. Thus we need to be able to determine the type of $N$ without knowing anything about the constructor identifier $v^\kappa$, aside from the fact that it represents a constructor of kind $\kappa$ and that the type of $x$ is $T$.

The side condition on the rule ensures that the constructor $v^\kappa$ does not escape from the scope of the “open” expression. This is essential both because we want the type to remain hidden, and because outside the expression, $v^\kappa$ is a free constructor identifier with no meaning associated with it.

For example, the earlier expression that opens and uses $\text{BoolPack}$ is well-typed using this rule as long as the expression surrounded by the $\text{open}$ clause has a type that does not mention $\text{Bool}$. For example, it could be a statement (with type $\text{Command}$) or return an integer value. However if it returned a value of type $\text{Bool}$ we would not know what to do with it as the identifier $\text{Bool}$ is introduced only as part of the open expression and has no meaning outside of that expression.

Computation with existentials is relatively straightforward. If you first pack something and then open it, it is as though you used it directly in a let statement. Thus,

$$o \quad \text{open} \ (\text{pack} \ (\mu, M) \ as \ \exists v^\kappa . T) \ as \ (u^\kappa, x) \ in \ N \rightarrow [M/x][\mu/u^\kappa]N$$
As usual we identify existential types and open expressions that differ only in the names of bound constructor identifiers.

We can relate this to our example using `BoolPack`. In that example, `BoolPack` was opened and the `IfThenElse` component was applied to the type `Integer`, `bval`, and the integers 3 and 17. By the computation rule, opening `BoolPack` allows access to the hidden type and the definitions of all of the components. Thus the computation using `IfThenElse` proceeds just as if we hadn’t hidden the type and component values.

Existential types are a suitable mechanism for defining abstract data types (ADT’s). In our example above, as with other abstract data types, values of existential type provide enough hooks that they can be used without knowing anything about the hidden implementation.

### 9.4 Adding subtypes to the polymorphic lambda calculus

In Chapter 5 we presented the intuition behind subtyping rules for standard type constructors. In this section we add subtypes to the polymorphic lambda calculus to form the bounded polymorphic lambda calculus, $\Lambda^P$, and formalize the rules for subtypes.

We will use subtyping in the bounded polymorphic lambda calculus in two ways. First, we shall use a “subsumption” rule to allow us to treat a value of a type as if it were a value of any of its supertypes. Second, we will use subtyping to constrain type and other constructor identifiers in polymorphic functions. For example, if we are defining a routine to calculate areas, we will only wish to apply it to records containing height and width fields. We will be able to constrain the function in this way with bounded polymorphism.

Let $\text{TwoD} = \{ \text{height: Integer; width: Integer} \}$. Then we can define

$$\text{Area} = \Lambda (R <: \text{TwoD}). \lambda (obj: R). \text{obj.height} \ast \text{obj.width}$$

The restriction on type identifier $R$ guarantees than any record with that type has `height` and `width` fields of type `Integer`, and thus the operations of extracting fields will succeed.

We define the types and expressions of the language first. Then we will define the subtyping rules, and finally the type-checking rules of this extended calculus. Because we have added types and expressions many times in this chapter, and this will be our final extension of the lambda calculus, we will provide complete definitions rather than just indicating incremental changes.
We will not make changes to the kinds, so we do not bother to repeat those definitions from Section 9.1. We will assume that the constructor constants include \textit{Fix}, introduced in Section 9.2 to generate recursive types, which has kind $(\star \Rightarrow \star) \Rightarrow \star$.

\begin{figure}[h]
\begin{equation}
\begin{array}{c}
T : \star \quad \mu : \kappa \\
\exists (\nu^\kappa \triangleleft \mu). \ T : \star \\
T : \star \quad \mu : \kappa \\
\forall (\nu^\kappa \triangleleft \mu). \ T : \star
\end{array}
\end{equation}
\caption{Kind checking rules for bounded polymorphic and existential types.}
\end{figure}

\textbf{Definition 9.4.1} Let $\mathcal{L}$ be an infinite collection of labels, and let $\mathcal{CC}$ be a collection of type constructor constants. The type constructor pre-expressions of $\Lambda r^P$ are given by the following grammar.

\begin{align*}
\mu, \nu \in \text{Constructor} &::= \nu^\epsilon \mid \epsilon^\epsilon \mid \text{Void} \mid \mu \rightarrow \nu \mid \mu_1 \times \ldots \times \mu_n \mid \\
&\mid \mu_1 + \ldots + \mu_n \mid \{ l_1 : \mu_1 ; \ldots ; l_n : \mu_n \} \mid \text{Ref} \ \mu \mid \\
&\mid \text{Command} \mid \lambda \nu^\mu. \mu \mid \mu \ \nu \mid \\
&\mid \forall \nu^\mu. \ T \mid \exists \nu^\mu. \ T \mid \forall (\nu^\mu \triangleleft \mu). \ T \mid \exists (\nu^\mu \triangleleft \mu). \ T
\end{align*}

where $\nu^\epsilon \in \mathcal{CL}$, $\epsilon^\epsilon \in \mathcal{CC}$, and $l_i \in \mathcal{L}$.

All but the last two type forms were introduced earlier. The last two are variations of the universal and existential types, where now the constructor identifiers are bounded by other constructors of the same kind as the bound identifier. The intuition is that the bound constructor identifiers represent constructors that are in the subtype relation to the given bounds.

The kinding rules for the new bounded polymorphic and existential types are given in Figure 9.4. We do not repeat the kinding rules for the other constructors given earlier in this chapter.
The expressions of the full language are given below:

\[
M \in \mathcal{PLC} : = x \mid c \mid \langle \rangle \mid \lambda (x: T). M \mid M N \mid \langle M_1, \ldots, M_n \rangle \mid \text{proj}_i(M) \mid \\
\text{case } x_1 : T_1 \text{ then } E_1 \mid \ldots \mid x_n : T_n \text{ then } E_n \mid \\
\text{in} \ T_1, \ldots, T_n(M) \mid \| \| \mid l_i : T_1 = M_1, \ldots, l_n : T_n = M_n \mid M l_i \mid \\
\text{ref } M \mid \text{null } \mid \text{val } M \mid \text{nop } \mid N := M \mid \\
\text{if } B \text{ then } \{ M \} \text{ else } \{ N \} \mid M; N \mid \\
\Lambda \nu^e. M \mid \Lambda (\nu^e <: \mu). M \mid M[\mu] \mid \\
\text{pack } \langle \mu, M \rangle \text{ as } \exists \nu^e . T \mid \text{pack } \langle \mu, M \rangle \text{ as } \exists (\nu^e <: \nu). T \mid \\
\text{open } M \text{ as } \langle \nu^e , x \rangle \text{ in } N
\]

We assume that the constants of \( \mathcal{PLC} \) include the constant \texttt{fix} with type \( \forall t. (t \to t) \to t \), which was introduced in Section 9.2 to support recursive definitions. Notice that we have added two new expressions representing the formation of bounded polymorphic functions and of existential packages with bounded existential types.

The axioms and rules for type-checking expressions of \( \Lambda \nu^e \), as well as those for \(<: \rangle \), are given with respect to a set, \( C \), of simple type constraints, that provide information about type identifiers. For example, in type-checking the body of the function \texttt{Area} defined earlier, we will need to use the fact that the type identifier \( R \) is a subtype of record type \texttt{TwoD}.

**Definition 9.4.2** Relations of the form \( \nu^e <: \mu \), where \( \nu^e \) is a type constructor identifier and \( \mu \) is a constructor expression of kind \( \kappa \), are said to be simple type constraints. A constraint system is defined as follows:

1. The empty set, \( \emptyset \), is a type constraint system.

2. If \( C \) is a type constraint system, and \( \nu^e \) is a constructor identifier that does not appear in \( C \), then \( C \cup \{ \nu^e <: \mu \} \) is a type constraint system.

Before collecting together all of the type-checking rules for the bounded polymorphic lambda calculus, we write down the rules for subtyping in Figures 9.5 and 9.6. Because types may involve type identifiers, the subtyping relations may depend on a constraint system, \( C \).

The rules \texttt{Reflex} \( <: \rangle \) and \texttt{Transitivity} \( <: \rangle \) are very natural for a partial ordering. Rule \texttt{Identifier} \( <: \rangle \) simply allows one to infer any constraint contained in the type constraint system.

Rules \texttt{Function} \( <: \rangle \) and \texttt{Record} \( <: \rangle \) reflect our earlier discussion of subtyping in Chapter 5. Function subtyping is covariant in the return type and contravariant in the parameter type. Record subtyping allows both breadth
9.4 Adding subtypes to the polymorphic lambda calculus

- **Reflex** \( < \): 
  \[
  \frac{\mu : \kappa}{C \vdash \mu <: \mu}
  \]

- **Transitivity** \( < \): 
  \[
  \frac{C \vdash S <: T, \quad C \vdash T <: U}{C \vdash S <: U}
  \]

- **Identifier** \( < \): 
  \[
  C \cup \{v^e <: \mu\} \vdash v^e <: \mu
  \]

- **Function** \( < \): 
  \[
  \frac{C \vdash S <: S', \quad C \vdash T <: T}{C \vdash S \rightarrow T <: S \rightarrow T}
  \]

- **Product** \( < \): 
  \[
  \frac{C \vdash S_i <: T_i, \text{ for } 1 \leq i \leq n}{C \vdash \prod_{i=1}^{n} S_i \times \cdots \times S_{n+m} <: T_1 \times \cdots \times T_n}
  \]

- **Sum** \( < \): 
  \[
  \frac{C \vdash S_i <: T_i, \text{ for } 1 \leq i \leq n}{C \vdash S_1 + \cdots + S_n <: T_1 + \cdots + T_n + \cdots + T_{n+m}}
  \]

- **Record** \( < \): 
  \[
  C \vdash \{l_1; S_1; \ldots; l_n; S_n; \ldots; l_{n+m}; S_{n+m}\} <: \{l_1; T_1; \ldots; l_n; T_n\}
  \]

- **Poly** \( < \): 
  \[
  \frac{C \vdash S <: T}{C \vdash \forall v^e. S <: \forall v^e. T}
  \]

- **BdPoly** \( < \): 
  \[
  \frac{C \cup \{w^e <: \mu\} \vdash S <: T}{C \vdash \forall(v^e <: \mu). S <: \forall(v^e <: \mu). T}
  \]

- **Exist** \( < \): 
  \[
  \frac{C \vdash S <: T}{C \vdash \exists v^e. S <: \exists v^e. T}
  \]

- **BdExist** \( < \): 
  \[
  \frac{C \cup \{w^e <: \mu\} \vdash S <: T}{C \vdash \exists(v^e <: \mu). S <: \exists(v^e <: \mu). T}
  \]

**Figure 9.5** Subtyping rules for \( \Lambda^e_{\mathcal{D}} \): Part 1.
and depth subtyping by supporting both the addition of more fields and subtyping of individual fields. Products behave like records whose fields are accessed by the $\text{proj}_i$ operations. Width and depth subtyping are supported by rule $\text{Product} <:$. The order of tuple components is significant because slots are accessed by location rather than name.

The rule $\text{Sum} <:$. for sums is new. It asserts that an element of type $S_1 + \ldots + S_n$ can masquerade as having type $T_1 + \ldots + T_n + \ldots + T_{n+m}$. The only operation applicable to an element of a sum type is the case statement. For an expression $M$ to successfully masquerade as having type $T_1 + \ldots + T_n + \ldots + T_{n+m}$ in a case statement (having $m+n$ alternatives) it is only necessary that each summand in the type of $M$ match one of the alternatives in the case statement. Hence the actual type of $M$ may involve fewer summands than expected in the case statement. Because the order of the summands is important in the interpretation of the case statement, the summands of the type of $M$ must be subtypes of the corresponding summands.

The subtyping rules for polymorphic and bounded polymorphic types, $\text{Poly} <:$ and $\text{BdPoly} <:$, are simplifications of the rule $\text{Function} <:$ in that they allow covariant changes to the return type, but no change to the bound on the type identifier in bounded polymorphic types. Similarly, the rules for existential and bounded existential types, $\text{Exist} <:$ and $\text{BdExist} <:$, indicate that these types are also covariant in the result type.

It is possible to generalize the rules $\text{BdPoly} <:$ and $\text{BdExist} <:$ to allow changes in the polymorphic bounds (contravariant in the first case and co-
variant in the second), but subtyping, and hence type checking, becomes undecidable [Pie94]. As a result we will work with this more restricted set of subtyping rules.

The four rules in Figure 9.6 define subtyping (or subconstructoring?) for higher-order type constructors and for fixed points. Rule ConstructorFcons \(<:\) allows subtyping of constructor functions if the result types change covariantly.

The rule ConstructorApp \(<:\) has the results of constructor function applications be subtypes as long as the functions are subtypes and the arguments are the same. It might seem that we could also allow the arguments to vary in a covariant way, but that would not be sound.

Suppose \(\mu = \lambda t. t \to t, \nu = S\), and \(\nu' = S'\) for some \(S' \triangleq S\). Then \(\mu \nu' = S' \to S'\), which is not a subtype of \(\mu \nu = S \to S\) (or vice versa). Without knowing extra information about type functions (e.g., without knowing they are covariant in their type parameter), one may not make deductions about subtyping when the actual parameters are subtypes. In practice, it is unusual for type functions to be either covariant or contravariant in their parameters, so we do not pursue this kind of rule further.

Recall that \(\cong\) is used to relate recursive types and their unfoldings in Section 9.2. The Cong \(<:\) rule allows one to substitute congruent types in a subtype relationship. In particular, if recursive types are subtypes, then so are their recursive unfoldings.

The last rule allows us to determine when two recursive types are subtypes of each other. This rule is the most complex to understand of all.

One might expect to be able to show that if \(C \vdash \mu' \triangleq \mu\), that would be sufficient to show that \(C \vdash \text{Fix}(\mu') \triangleq \text{Fix}(\mu)\). But that is false.

Let
\[
\mu \triangleq \lambda t. \{m: t \to \text{Boolean}\}
\]
and
\[
\mu' \triangleq \lambda t. \{x: \text{Integer}; m: t \to \text{Boolean}\}.
\]

Clearly \(\mu' \triangleq \mu\). However, we show that \(S' = \text{Fix}(\mu')\) is not a subtype of \(S = \text{Fix}(\mu)\). Define
\[
s': S' = \{x: \text{Integer} = 0; m: S' \to \text{Boolean} = \lambda (u: S'). (u.x = 0)\},
\]
and
\[
s: S = \{m: S' \to \text{Boolean} = \lambda (u: S). \text{true}\},
\]
and
\[
f \triangleq \lambda (v: S). v.m(s).
\]
Suppose, in order to find a contradiction, that $\mathcal{C} \vdash S' <: S$. By the fact that values of subtypes can masquerade as supertypes (supported more formally by the Subsumption rule later in this section), it follows that $\mathcal{C} \vdash s': S$, and $f(s')$ is well-typed. But $f(s') = s'.m(s) = (s.x = 0)$, and $s$ has no $x$ field. Thus the assumption that $\mathcal{C} \vdash S' <: S$ must be false. This shows that the intuitive subtyping rule for recursive types is not correct. We need the stronger assumption given in rule $\text{Fix} <: \mu$ to obtain correct conclusions.2

While verifying the correctness of this rule would take us too far afield, we present the following intuition, which may be helpful. If we hope to show that

$$\mathcal{C} \vdash S' = \text{Fix}(\mu') <: \text{Fix}(\mu) = S,$$

then we will need

$$\mathcal{C} \vdash S' \equiv \mu'(S') <: \mu(S) \equiv S.$$

If the hypothesis of the $\text{Fix} <:$ rule holds, then we already know that

$$\mathcal{C} \cup \{v'' <: v''\} \vdash \mu'(v''') <: \mu(v''').$$

As a result, if $\mathcal{C} \vdash S' <: S$, it will follow from the hypothesis of the rule that $\mathcal{C} \vdash \mu'(S') <: \mu(S)$. That is, if $\mathcal{C} \vdash S' <: S$, then the unfoldings of $S'$ and $S$ are also in the subtyping relation.

Of course, this is not a proof — the implications go the wrong direction! Rather it shows that a possible proof by contradiction fails to reveal a contradiction. Instead one can treat this as a plausibility argument. An actual proof of correctness would require either a co-inductive proof or arguments using infinite trees [AC93], neither of which we wish to pursue.

Stronger rules are possible for subtyping with recursive types. However, we will only need this one rule for our work with object-oriented languages.

The typing rules for expressions in Figures 9.7 and 9.8 are a straightforward extension of those given earlier. The main difference between these and the earlier type-checking rules is the inclusion of $\mathcal{C}$ as an added hypothesis for type-checking. This is necessary in order to type-check the new expressions in Figure 9.9.

The only difference between type-checking bounded polymorphic functions and regular (unbounded) polymorphic functions is that with bounded polymorphism we type-check the body under the assumption (added to $\mathcal{C}$) that the type parameter is constrained by the bound of the type identifier.

2. The alert reader will notice the similarity of this argument to the failure of subtyping with binary methods in Eiffel.
Similarly when applying a bounded polymorphic function, we must ensure that the actual type parameter is a subtype of the upper bound expressed in the type.

Bounded existentials are handled similarly to bounded polymorphic functions. We must ensure the hidden type is a subtype of the declared bound before packing the expression into one whose type is a bounded existential. When unpacking such an expression, one can count on the fact that the hidden type is bounded above by the upper bound included in the type.

The subsumption rule states that if an expression has a type \( S \), then it also has as type any supertype of \( S \). Another way of stating this is that an element of a type can successfully masquerade as an element of any supertype. This rule provides the formal justification of subtype polymorphism.

Computation rules for the bounded polymorphic and existential expressions are similar to those for their unbounded counterparts in Section 9.1 and 9.3. Their formulation is left as an exercise for the reader. It is possible to show a subject reduction theorem for this calculus (see [Pie02]):

**Theorem 9.4.3** Suppose \( C, E \vdash M : T \) and \( M \rightarrow^* M' \). Then \( C, E \vdash M' : S \) for some \( S \) such that \( C \vdash S \ll T \).

In essence, this theorem states that computation preserves types. One might think that computation might not result in a subtype, but consider the following simple example:

\[
(\lambda (x : T). x) (y) \rightarrow y
\]

Suppose \( y : S \), where \( C \vdash S \ll T \). It is easy to see that the left side has type \( T \) because the function has type \( T \rightarrow T \) and the type of \( y \) can be promoted to \( T \) by subsumption. It is easy to see that the expression cannot have a smaller type because subsumption applied to the function can only result in a larger return type. On the other hand, the result of the computation, \( y \), has type \( S \). Thus evaluating an expression can result in a value whose minimum type is a subtype of the original.

Constructing semantic models of the bounded polymorphic lambda calculus and proving subject-reduction is non-trivial and beyond the scope of this text. For our purposes it is sufficient to understand the syntax, type-checking, and some of the computation rules for this extended typed lambda calculus. We hope the reader will accept that the typing and computation rules discussed in this chapter are intuitive and that the subject-reduction theorem is plausible. See the references for suggestions as to where to find the details of these developments.
identifier $\vdash C, E \cup \{x : T\} \vdash x : T$

Constant $\vdash C, E \vdash c : N$

Void $\vdash C, E \vdash \emptyset : \text{Void}$

Function $\vdash C, E \vdash \{x : S\} \vdash M : T$

$$\vdash C, E \vdash \lambda (x : S). M : S \rightarrow T$$

FuncApp $\vdash C, E \vdash M : S \rightarrow T$, $\vdash C, E \vdash N : S$

$$\vdash C, E \vdash MN : T$$

Product $\vdash C, E \vdash M_i : T_i$, for $1 \leq i \leq n$

$$\vdash C, E \vdash \langle M_1, \ldots, M_n \rangle : T_1 \times \ldots \times T_n$$

Proj $\vdash C, E \vdash M : T_1 \times \ldots \times T_n$

$$\vdash C, E \vdash \text{proj}_i (M) : T_i \text{ for } 1 \leq i \leq n$$

Sum $\vdash C, E \vdash M : T_1$

$$\vdash C, E \vdash \text{in}_i^{T_1 \ldots T_n} (M) : T_1 + \ldots + T_n \text{ for } 1 \leq i \leq n$$

Case $\vdash C, E \vdash \{x_i : T_i\} \vdash E_i : U_i$, for $1 \leq i \leq n$

$$\vdash C, E \vdash \text{case } M \text{ of } x_i : T_i \text{ then } E_i \mid \ldots \mid x_n : T_n \text{ then } E_n : U$$

Record $\vdash C, E \vdash M_i : T_i$, for $1 \leq i \leq n$

$$\vdash C, E \vdash \{l_1 : T_1, \ldots, l_n : T_n \} : \{l_1 : T_1, \ldots, l_n : T_n \}$$

Selection $\vdash C, E \vdash M : \{l_1 : T_1, \ldots, l_n : T_n \}$

$$\vdash C, E \vdash M.l_i : T_i \text{ for } 1 \leq i \leq n$$

Figure 9.7 Type-checking rules for $\Lambda^C$, part 1: revised rules.
### 9.5 Summary

We have now completed the preliminary background material needed to understand the semantics of object-oriented languages. In the next chapter we will use the machinery we have built up here to explain objects and classes. We will use pairs and records to represent objects as a pair of the records of meth-

---

#### Reference

\[
\frac{C, E \vdash M : T}{C, E \vdash \text{ref } M : \text{Ref } T}
\]

#### Null

\[
E \vdash \text{null} : \text{Ref } T
\]

#### Value

\[
\frac{C, E \vdash M : \text{Ref } T}{C, E \vdash \text{val } M : T}
\]

#### Assignment

\[
\frac{C, E \vdash N : \text{Ref } T, \quad C, E \vdash M : T}{C, E \vdash N := M : \text{Command}}
\]

#### Conditional

\[
\frac{C, E \vdash B : \text{Boolean}, \quad C, E \vdash M : T, \quad C, E \vdash N : T}{C, E \vdash \text{if } B \text{ then } \{ M \} \text{ else } \{ N \} : T}
\]

#### Sequencing

\[
\frac{C, E \vdash M : S, \quad C, E \vdash N : T}{C, E \vdash M; N : T}
\]

#### PolyFunc

\[
\frac{C, E \vdash M : T}{C, E \vdash \lambda v^\varepsilon. M : \forall v^\varepsilon. T}
\]

#### PolyApp

\[
\frac{C, E \vdash M : \forall v^\varepsilon. T, \quad \mu : \kappa}{C, E \vdash M[\mu/\varepsilon]T}
\]

#### Pack

\[
\frac{C, E \vdash M : [\mu/\varepsilon]T, \quad \mu : \kappa}{C, E \vdash \text{pack } \langle \mu, M \rangle : \exists v^\varepsilon. T}
\]

#### Unpack

\[
\frac{C, E \vdash M : \exists v^\varepsilon. T, \quad C, E \uplus \{ x : T \} \vdash N : S}{C, E \vdash \text{open } M \text{ as } \langle v^\varepsilon, x \rangle \text{ in } N : S}
\]

where \( v^\varepsilon \) does not appear in \( S \)

**Figure 9.8** Type-checking rules for \( \Lambda^p_{<} \), part 2: revised rules.
BdPolyFunc \[
\frac{\mathcal{C} \cup \{\nu^e \prec \mu\}, \mathcal{E} \vdash M : T, \; \mu : \kappa}{\mathcal{C}, \mathcal{E} \vdash \lambda(\nu^e \prec \mu). M : \forall(\nu^e \prec \mu). T}
\]

BdPolyApp \[
\frac{\mathcal{C}, \mathcal{E} \vdash M : \forall(\nu^e \prec \mu). T, \; \mathcal{C} \vdash \mu' \prec \mu}{\mathcal{C}, \mathcal{E} \vdash M[\mu'/\nu^e] : T}
\]

BdPack \[
\frac{\mathcal{C}, \mathcal{E} \vdash M : [\mu/\nu^e]T, \; \mathcal{C} \vdash \mu \prec \nu}{\mathcal{C}, \mathcal{E} \vdash \text{pack}(\mu, M) : \exists(\nu^e \prec \mu). T : \exists(\nu^e \prec \nu). T}
\]

BdUnpack \[
\frac{\mathcal{C}, \mathcal{E} \vdash M : \exists(\nu^e \prec \mu). T, \; \mathcal{C} \cup \{\nu^e \prec \mu\}, \mathcal{E} \cup \{x : T\} \vdash N : S}{\mathcal{C}, \mathcal{E} \vdash \text{open} M \text{ as } \langle \nu^e, x \rangle \text{ in } N : S}
\]

where \(\nu^e\) does not appear in \(S\)

Subsumption \[
\frac{\mathcal{C}, \mathcal{E} \vdash M : S, \; \mathcal{C} \vdash S \prec T}{\mathcal{C}, \mathcal{E} \vdash M : T}
\]

Congruence \[
\frac{\mathcal{E} \vdash M : T, \; T \equiv T'}{\mathcal{E} \vdash M : T'}
\]

**Figure 9.9** Type-checking rules for \(\bar{\Lambda}_\kappa^\mu\), part 3: new rules.

...ods and instance variables. Existential types will be used to hide the record of instance variables outside of the methods of an object, and bounded polymorphic functions in turn will be critical in ensuring that methods defined in a class can safely be inherited in subclasses.
Alonzo Church developed the untyped lambda calculus in the 1930’s to aid in the study of mathematical logic [Chu32], and in order to define computable functions [Chu36, Chu41]. Church’s student Kleene discovered the remarkable expressiveness of the untyped lambda calculus, proving in 1936 [Kle36] that the collection of functions from natural numbers to natural numbers definable in the untyped lambda calculus was the same as that definable from recursive functions. A year later, Turing [Tur37] proved that these sets were the same as the set of Turing machine computable functions.

The typed lambda calculus [Chu40, Hen50] was introduced several years later as a restriction of the untyped lambda calculus. However, as a result of research into the solution of recursive domain equations in the 1970’s, the simply typed lambda calculus with recursive types and terms is now typically seen as more general, with the untyped lambda calculus treated as a special case whose models satisfy a particular domain equation. See Mitchell [Mit96] or Gunter’s [Gun92] texts for details. We follow the modern approach here by working with the typed lambda calculus. Good references on the lambda calculus include the books and articles of Hindley [Hin97], Hindley and Seldin [HS86] and Barendregt [Bar84, Bar92]. The recent text [Pie02] by Pierce is an excellent reference for the kinds of extensions of the lambda calculus discussed here. Pierce’s survey article, “Foundational Calculi for programming languages” [Pie97] provides a short introduction to both the lambda calculus and the pi calculus.

Interest in using the lambda calculus to model high level programming languages was spurred by the work of Landin [Lan65, Lan66] and Strachey [Str67] in the 1960’s. Initially there was some skepticism as to the sound-
ness of the untyped lambda calculus because of the lack of models for its use in denotational semantics, but work by Scott [Sc76] and others showed that both the untyped lambda calculus and the typed lambda calculus with recursive domain equations had very rich sets of models.

The polymorphic lambda calculus, sometimes referred to as second-order lambda calculus or System F, was invented independently by Girard [Gir71, Gir86] and Reynolds [Rey74]. They came to this system through two different viewpoints, Girard through a study of the proof theory of mathematical logic, and Reynolds from a programming languages perspective. McCracken [McC79], building on results of Scott, produced the first model of the polymorphic lambda calculus.

MacQueen, Plotkin, and Sethi invented the ideal model [MPS86] for the implicit polymorphic lambda calculus. This model is useful in understanding languages like ML in which the system infers polymorphic types for untyped terms rather than having types declared as part of the language syntax. The importance of this model is that subtyping in these languages could be interpreted as subsets of ideals. While ideal models are not sound for the explicitly typed polymorphic lambda calculus, they did provide an important semantic understanding of the notion of subtype.

While existential types were included in Girard’s System F, Mitchell and Plotkin’s paper, “Abstract types have existential types” [MP88], was the first to tie these types to programming language constructs. While the initial application was to model information hiding and abstract data types, we will see later that they are extremely useful in modeling objects with instance variables.


Mitchell’s 1984 paper, “Coercion and type inference” [Mit84], also contained suggestions that bounded polymorphism would be necessary for type inference to obtain most general types.

HP Lab’s Abel group along with Mitchell [CCH89] were among the first to notice that many of the applications of bounded polymorphism (especially those involving binary methods) required an extension to F-bounded polymorphism. The designers of Emerald also recognized this problem, providing similar mechanisms.

The first models of bounded polymorphic lambda calculus were by Bruce
Historical Notes and References for Section II

and Longo [BL90] and by Breazu-Tannen et al. [BTCGS91]. In each case subtyping was not modeled by subset of the interpretation of types, but instead by particular kinds of mappings from the interpretation of a subtype to the interpretation of the supertype.

In their paper, Bruce and Longo extended the definition of the bounded second-order lambda calculus by generalizing the definition of the subtyping relation on bounded polymorphic types to allow contravariant changes in type bounds. While the models were sound, it turned out that this extension destroyed the decidability of subtyping, as Pierce showed in his paper, “Bounded quantification is undecidable” [Pie94]. Both subtyping and type checking for Cardelli and Wegner’s original version of the language are decidable. As a result, the original version is generally used even though subtyping of bounded polymorphic types is more restricted than is strictly necessary. Most models for the bounded polymorphic lambda calculus are easily extended to handle F-bounded polymorphism.

Amadio and Cardelli provided the first careful theoretical analysis of subtyping rules for recursive types. Kozen et al. [KPS93] provided a more efficient algorithm for deciding subtyping. A recent tutorial, “Recursive Subtyping Revealed” [GLP00], provides a more modern, coinductive treatment of recursive types.
This page intentionally left blank
Part III

Formal Descriptions of Object-Oriented Languages
This page intentionally left blank
In the last two chapters we introduced the formal notation necessary in order to specify programming languages. Now we are ready to present the formal specification of a simple object-oriented programming language, SOOL, in terms of the formal notation of λC. The language SOOL is intended to share the core features of most statically typed class-based object-oriented programming languages. It is essentially the same language used in the examples in the earlier chapters of this book. It does not include the extensions discussed in Chapter 4. These will be added in Part IV, starting with Chapter 15.

Analyzing this language and its extensions will be the focus of most of the remaining chapters of this text. We will first carefully give formal specifications of the syntax, type-checking rules, and semantics of the SOOL. With that specification complete we will be able to prove that the type system is sound.

In order to make it simpler to write down semantics and type-checking rules for our language, the syntax for the language will be somewhat more cluttered than most common object-oriented languages. However, we will also introduce abbreviations which provide a more friendly and familiar programming style. One can imagine the early steps of a compiler expanding the code from the language with abbreviations to the full language.

10.1 Informal description and example

An early example of the programming notation introduced in Chapter 2 was the program repeated in Figure 10.1. That will be a valid class definition in our language with abbreviations (as will the version that drops all of the "self." subexpressions), but the formal language specification will define
class CellClass {
    x: Integer = 0;

    function get(): Integer is
    { return self.x }

    function set(nuVal: Integer): Void is
    { self.x := nuVal }

    function bump(): Void is
    { self <= set(self <= get()+1) }
}

Figure 10.1 Cell class.

a more verbose version of the language which allows function and classes
to be defined as “anonymous” expressions. That is, similar to the way the
lambda calculus allows one to define unnamed function expressions using
the lambda notation, we will be able to write down unnamed expressions
that represent functions and classes.

In Figure 10.2, CellClass is shown as written in the fully expanded lan-
guage. We explain the changes in notation between the two versions below.

In the fully expanded language we will adopt some of the notation of the
lambda calculus with references described in Section 5.1.3. Types of the form
Ref T are reference types, denoting the types of variables holding values
of type T. An expression with type Integer (e.g., 3+7) denotes an integer
value, while an expression x of type Ref Integer denotes an updatable
variable holding values of type Integer.

An expression of the form ref e denotes a reference or location holding
the value of e. These expressions are used to initialize variables. If x is
declared to be a variable of type T with initial value e in the abbreviated
language, then x will be defined as a constant with type Ref T and value
ref e in the expanded language.

We will use the val qualifier to distinguish between the value of a variable
as a location (e.g., as used on the left side of assignment statements) and the
value stored in the variable (e.g., as used on the right side of expressions).
Thus if x is an integer variable (identifier with type Ref Integer), we will
10.1 Informal description and example

```
CellClass = class ( ||
  x: Ref Integer = ref 0
  ||
  get: Void → Integer =
    function(v: Void): Integer is {
      return val self.x
    };

  set: Integer → Command =
    function(nuVal: Integer): Command is {
      self.x := nuVal;
      return nop
    };

  bump: Void → Command =
    function(v: Void): Command is {
      self := set(self := get()+1);
      return nop
    }
  ||)
```

**Figure 10.2** CellClass in the fully expanded language.

write

```
x := (val x) + 1;
```

rather than the more usual `x := x + 1`.

We will generally not bother to include the `Ref` types, `ref` expressions, and `val` qualifiers in example code, but we will use them in formal specifications in order to distinguish variables of type `T` from values of type `T` and the two kinds of values of variables. Having these distinctions show up in the language will make it significantly easier to write type-checking rules and semantics for SIOLEC.

In order to avoid having to distinguish between functions and procedures in the semantics, the expanded language will represent procedures as functions which return a value `nop` of type `Command`, the type of commands. The expression `nop` stands for a command which does nothing. In the abbreviated language, we will allow `return nop` to be omitted.

Because of this convention, the return type of procedures will also be given as `Command`, the type of all other statements, rather than the `Void` we have been using in the abbreviated language. While this differs from the conven-
tions used in C-style syntax, it is helpful to distinguish between a function which takes a default value () of type \texttt{Void}, and a procedure whose return type indicates that it is executed to change the state of the computer.

Invoking a parameterless method involves applying it to (). In order to make method definitions and invocations more uniform, parameterless methods will be considered to be abbreviations for methods which take a new (and unused) parameter of type \texttt{Void}. Because () has type \texttt{Void}, this will work smoothly with our type-checking rules.

Other differences between the original and the non-abbreviated version of the notation for class definitions are that names of classes and methods are separated from the expressions that define them, and method names are provided with their types. One reason for specifying this somewhat more verbose version of the language is that it will allow us to have anonymous class and function expressions.

The method \texttt{get} in Figure 10.2 illustrates these differences. The method name with its signature (typing) occurs before the equals sign. After the equals sign is the definition of the anonymous function represented by \texttt{get}.

Another change is that the collections of instance variables and methods are grouped together using the symbols “"]“ and “[””. These symbols are chosen to be distinct from the curly brackets used to enclose sequences of statements. As we will see in the formal specification, records are not considered expressions or types of \texttt{SOOL}. However, notation representing classes as being composed of a record of instance variable declarations and another record of method definitions will make it easier to specify type-checking rules.

These changes are included to make the formal syntax more uniform and allow us to write fewer formal rules for grammar, type checking rules, and semantics. It should be clear that it is easy to automatically translate a program in the abbreviated language into one of the expanded language. Again, all formal specification will be of the expanded language.

### 10.2 Syntax and type-checking rules

In this section we will provide formal specifications of the syntax and type-checking rules for our simple language, \texttt{SOOL}. Later on we will add more constructs to the language to increase its expressiveness. The expressions of \texttt{SOOL} are limited to those which type check correctly. As with the typed lambda calculus, we will introduce these expressions in two steps. In this
section we provide the syntactic specification of pre-expressions of the language, some examples, and then the type checking rules. The legal expressions of the language will be comprised of those pre-expressions which can be assigned a type via the type-checking rules.

10.2 Syntax and type-checking rules

10.2.1 Syntax of types and expressions

The syntax of \( S\OOL \) types and expressions will be given by a context-free grammar. Because expressions contain type expressions, we begin by specifying the syntax of types.

Syntax of types

The definition of type expressions will be built from a collection of built-in types (or type constants), \( TC \). We assume that \( TC \) contains at least the type constants \( Integer \) for the set of integers, \( Real \) for the real numbers, \( Boolean \) for the booleans, \( Character \) for the characters, \( String \) for the strings, \( Void \) for a type with only a single value, written (), and \( Command \) for the type of statements or commands. When convenient, we will assume \( TC \) has other built-in types.

**Definition 10.2.1** The set, \( TYPE_{S\OOL} (TC, L, TI) \), of type expressions of \( S\OOL \) over a set \( TC \) of type constants, a set \( L \) of record labels, and a set \( TI \) of type identifiers is given by the following context-free grammar (where we assume \( \in TI, c \in TC, l_i \in L, and T, T_i \in TYPE_{S\OOL} (TC, L, TI) \) :

\[
T \in Type :::= C | t | T_1 \times \ldots \times T_n \rightarrow T_{n+1} | Ref T | ObjectType RT | VisObjectType(RT_i, RT_m) | ClassType(RT_i, RT_m)
\]

\[
RT \in RTType ::= \{ l_1 : T_1 ; \ldots ; l_n : T_n \ |
\]

Items generated by \( RTType \) are not themselves considered legal types of \( S\OOL \). Instead these record types are used only to build up legal object and class types. Record types consist of a list of labels and their associated types.

Types of the form \( T_1 \times \ldots \times T_n \rightarrow T_{n+1} \) are function types, denoting functions which take \( n \) arguments of types \( T_1 \) through \( T_n \) and return a value of type \( T_{n+1} \). Object types are provided with a record type that specifies the names and types of methods.

As discussed earlier, types of the form \( Ref T \) are reference types, denoting the types of variables holding values of type \( T \). An expression with type \( In-\)
integer (e.g., 3+7) denotes an integer value, while an expression \( x \) of type \( \text{Ref Integer} \) denotes an updatable variable holding values of type Integer.

Types of the form \( \text{ObjectType \ M} \) are the types of objects with method names and types given by the record type \( M \). (We often use identifiers \( M \) and \( IV \) to represent record types, where \( M \) is used as the type of the record of methods in a class or object, while \( IV \) is used as the type of the record of instance variables.) The names and types of instance variables are not visible in object types.

The keyword \( \text{self} \) provides access to both the instance variables and methods of the object. As a result, we need a type for \( \text{self} \) that includes the names and types of instance variables, reflecting the fact that they are visible or accessible via \( \text{self} \). Thus the type of \( \text{self} \) will be a “visible” object type rather than \( \text{ObjectType \ M} \). Types of the form \( \text{VisObjectType}(IV_{ref}, M) \) are introduced explicitly to serve as the types of \( \text{self} \). In definitions of this form, \( IV_{ref} \) will represent the type of the record of instance variables (all of which will be references), whereas \( M \) will represent the type of the record of methods (all of which will be functions).

We have so far only encountered class types briefly in Section 5.3, where we discussed the lack of subtyping between class types. Class types are normally not explicitly written in programs of \( \text{SOCOL} \). However, because we will support class expressions, it is possible to have variables that hold class values or to pass class expressions as parameters to functions.

While objects hide their record of instance variables, the instance variables will be visible in classes, as they may be used in subclasses. (Thus our instance variables behave like protected members of Java or C++, and are visible in subclasses.) Thus the types of instance variables must be specified in class types.

In a class type of the form \( \text{ClassType}(IV, M) \), \( IV \) is the record type specifying the names and types of the initial values of instance variables, while \( M \) specifies the names and types of the methods. Notice the difference between the types in visible object types and class types. In the former, \( IV_{ref} \) represents the type of the record of instance variables. That is, it is a record of reference types. In class types, \( IV \) represents the type of a record of initial values of instance variables. Typically these will not be explicit references.

**Syntax of expressions**

The syntax of \( \text{SOCOL} \) programs is given by a context-free grammar. Rather than specifying all of the built-in constants and pre-defined functions of the
language, we will simply assume that there is a collection, $\mathcal{E}$, of such expressions. This will allow us to introduce such built-in expressions as needed rather than give a complete specification now. We also assume a collection, $\mathcal{L}$, of labels (used for method and instance variable names), and a set, $\mathcal{E}$, of expression identifiers.

**Definition 10.2.2** The pre-expressions, $\mathcal{P}$, of $\mathcal{E}$, $\mathcal{L}$, and $\mathcal{E}$, of $\mathcal{S}$, over a set $\mathcal{E}$ of expression constants, a set $\mathcal{L}$ of labels, and a set $\mathcal{E}$ of identifier names, is given by the following context-free grammar (where we assume $id \in \mathcal{E}$, $t \in \mathcal{T}$, $c \in \mathcal{E}$, $l$, $m \in \mathcal{L}$, and $T$, $T_i \in \mathcal{E}$):

$$
\begin{align*}
\text{Prog} &::= \text{Program id; Blk} \\
\text{Blk} \in \text{Block} &::= \text{TDefs CDs \{ } S \text{ return } E \} \\
\text{TDefs} \in \text{TDefS} &::= \epsilon \mid \text{type TDL} \\
\text{TDL} \in \text{TDefLst} &::= t = T \mid t = T; \text{TDL} \\
\text{CDefs} \in \text{CDefS} &::= \epsilon \mid \text{const CDL} \\
\text{CDL} \in \text{CDefLst} &::= \text{id: } T = E \mid \text{id: } T = E; \text{CDL} \\
\text{E} \in \text{Exp} &::= \text{id } \mid c \mid \text{nil } ( \) \mid \text{val E } \mid \text{ref E } \mid E(E_1, \ldots, E_n) \mid \\
& \quad \text{function{id: } T_i, \ldots, id_n: T_n}: T_{n+1} \text{ is Blk } \\
& \quad \text{class}(R_i, R_m) \mid \text{new E } \mid E \leftarrow m \mid E.1 \mid \text{class inherits E modifies } l_{j_1}, \ldots, l_{j_m} (R_i, R_m) \\
\text{R} \in \text{Rec} &::= \{ l_1: T_1 = E_1; \ldots; l_n: T_n = E_n \} \\
\text{S} \in \text{Stmt} &::= \text{nop } \mid \text{id:= } E \mid \text{if } E \text{ then } S_1 \text{ else } S_2 \mid \\
& \quad \text{while } E \text{ do } S \mid S_1 ; S_2
\end{align*}
$$

A program in $\mathcal{S}$ consists of a Program statement with the name of the program, followed by a block. The block consists of a list of type and constant definitions (which include definitions of functions and classes), and finally by a statement and return expression surrounded by curly brackets. For programs, the block will be the main program and will return the default value nop. In function bodies, the return value can be any expression of the appropriate type.

The symbol $\epsilon$ indicates that a construct may be replaced by an empty string. In other words, the $\epsilon$ in type and constant definitions indicates that these constructs are optional.

Expressions in $\mathcal{S}$ consist of identifiers, constants, the keyword nil, val expressions, ref expressions, function applications, function expressions, class expressions, new expressions (for creating new objects), message sends, extracting a labeled field, and subclass expressions. The keyword nil represents an uninitialized object.
We have already discussed \texttt{ref} and \texttt{val} expressions. As stated earlier, if \( x \) is declared to be a variable of type \( T \) in the abbreviated language, then \( x \) will be defined as a constant with type \texttt{Ref} \( T \) in the expanded language. It will be initialized with an appropriate default value for the type. If \( T \) is an object type then it will be initialized with \texttt{nil}. Variables of number types will be initialized to the zero value of the type, characters to the \texttt{”NUL”} character, and strings to the empty string. Functions will be initialized to functions returning the default value of the result type.

For example, a variable declaration of the form

\[
x: \text{Integer};
\]

will be transformed into the constant definition

\[
x: \text{Ref Integer} = \text{ref} 0;
\]

Thus \( x \) is initialized to be a reference to a location that initially holds 0.

Assignment statements can update the value held (referred to) by \( x \), but do not change the reference (location) itself. Hence it may be included in with the other constant definitions.

Function applications consist of a function expression followed by the list of actual parameters. Function expressions contain a list of formal parameters and their types, the return type (which is \texttt{Command} for procedures), and the function body.

Class expressions contain a record of initial values of instance variables and a record of method definitions. Subclass expressions differ from class expressions by including the superclass and a list of methods that are to be overridden by the subclass. \texttt{”New”} expressions contain the class that is used as a template to generate a new object.

Message sends include the target object and the name of the message. Recall that we use \( \Leftarrow \) to distinguish message sends from accesses to instance variables, which use the \texttt{“.”} notation. Sending a message to \texttt{nil} should result in a run-time error. Labelled field projection normally appears only in the bodies of method expressions. It typically appears in the form \texttt{self.x}, where \( x \) is the name of an instance variable of the class.

The statements of the language include \texttt{nop}, assignment statements, if-then-else expressions, while loops, and sequencing of statements. The expression \texttt{nop} represents a statement that has no effect (a “no op” or “skip”).

Two keywords of the language do not appear in the grammar above. They are \texttt{self}, a name for the object currently executing a method, and \texttt{close}, a
function used to convert the type of self in order to hide the instance variables. We do not introduce them as constants because they may appear only inside method definitions, and cannot be assigned types independently of those contexts. In particular their meanings will vary depending on the context. To be consistent with our type-checking rules presented below, these keywords will instead be introduced as “bound” identifiers. The type of self will always be the “visible” object type generated by the class definition that contains it. Recall this is necessary so that access is allowed to instance variables.

We have not yet discussed the close keyword. Its sole purpose is to convert objects of type VisObjectType(IVref, M) to type ObjectType M once we no longer need access to instance variables. If the value of self is provided outside of the class definition, either by returning it from a method or using it as a parameter in another method, the instance variables should no longer be accessible. In particular, we do not wish to have either the return type or the parameter type of a method be a visible object type that includes the types of instance variables.

Thus, before passing self as a parameter or returning it from a method, we will apply close in order to convert it to an object type. As with val, the abbreviated form of the language will not require the programmer to insert applications of close, instead they will automatically be inserted where needed.

10.2.2 Examples

Figure 10.3 contains a very simple example of a complete program in SOOL. It defines a Point class, creates an object from Point, and moves it. It uses the syntax abbreviations that allow the program to look more like it was written in a typical object-oriented language. Notice that we do not require the section headings type and const, and variable declarations are written in a form similar to those found in most languages.

If we remove all of the abbreviations, we get the program in Figure 10.4. For this program, the syntax is defined exactly by the context-free grammar given in the previous section.

We have not yet discussed constructors for classes. We will omit constructors in SOOL, replacing them by functions that return classes. In the language with abbreviations we will write these as parameterized classes, but these can be replaced with functions returning classes.

In Figure 10.5, a variation of the program PointExample from Figure
Program PointExample;

PointType = ObjectType {
    move: Integer × Integer → Command;
    getx: Void → Integer;
    gety: Void → Integer
};

class Point {
    x: Integer = 0;
    y: Integer = 0;
    function move(dx: Integer,dy:Integer): Command is {
        x := x + dx;
        y := y + dy
    }
    function getx(): Integer is { return x }
    function gety(): Integer is { return y }
};

pt: PointType;

{ pt := new Point;
  pt ☣ move(3,2) }

Figure 10.3  PointExample program in language with abbreviations.

10.3 illustrates the use of parameterized classes. The class PPoint takes two integers as parameters. The two integer parameters are used to initialize the instance variables x and y.

In Figure 10.6 we remove the abbreviations to reveal the parameterized class to be a function returning a class. In the assignment to pt in the main program, the application of PPoint to the arguments 2 and 7 has higher priority than the new operator.
Program PointExample;

```
type
  PointType = ObjectType {{
    move: Integer × Integer → Command;
    getx: Void → Integer;
    gety: Void → Integer
  }};
  PtClassType = ClassType {{
    x: Integer;
    y: Integer
  }, {{
    move: Integer × Integer → Command;
    getx: Void → Integer;
    gety: Void → Integer
  }});
  const
  Point: PtClassType = class {{
    x: Integer = 0;
    y: Integer = 0;
  }, {{
    move: Integer × Integer → Command =
      function(dx: Integer, dy: Integer): Command is {
        self.x := (val self.x) + dx;
        self.y := (val self.y) + dy;
        return nop
      }
    getx: Void → Integer = function(): Integer is {
      return val self.x
    }
    gety: Void → Integer = function(): Integer is {
      return val self.y
    }
  }});
  pt: Ref PointType = ref nil;
{
  pt := new Point;
  pt ← move(3, 2);
  return nop
}
```

Figure 10.4  PointExample program in language without abbreviations.
Program PPointExample;

... class PPoint(x0: Integer, y0: Integer) {
  x: Integer = x0;
  y: Integer = y0;
  ...  
};

pt: PointType;

{
  pt := new PPoint(2,7);
  pt move(3,2)
}

Figure 10.5 Parameterized Point class in language with abbreviations.

10.2.3 Type-checking rules

Most programming languages do not provide explicit formal rules for statically type-checking programs to the programmer. Instead they are typically implicit in the language definition and then in the code of compilers. However, these rules are not difficult to formulate. As in the typed lambda calculus, the type-checking rules can be given inductively based on the productions of the context-free grammar that generates the language. In this section we provide the formal type-checking rules for SOOL.

As in the typed lambda calculus, the type checking rules require information about the types of free identifiers in an expression and information about the meaning of type identifiers. The static type environment \( \mathcal{E} \) associates expression identifiers with types.

**Definition 10.2.3** A static type environment, \( \mathcal{E} \), is a finite set of associations between identifiers and type expressions of the form \( x: \mathcal{T} \), where each \( x \) is unique in \( \mathcal{E} \) and \( \mathcal{T} \) is a type. If the relation \( x: \mathcal{T} \in \mathcal{E} \), then we sometimes write \( \mathcal{E}(x) = \mathcal{T} \).

An important difference between SOOL and the typed lambda calculus is that SOOL can include type definitions. These definitions are recorded
Program PPointExample;

type
  ...

const

  PPoint: Integer × Integer → PtClassType =
  function(x0: Integer, y0: Integer): PtClassType is {
    return class ( ||
      x: Integer = x0;
      y: Integer = y0;
      ||
      ... ||
    )
  }

  pt: Ref PointType = ref nil;

{
  pt := new PPoint(2,7);
  pt ⇔ move(3,2);
  return nop
}

Figure 10.6 Parameterized Point class in language without abbreviations.

for type checking purposes in a type constraint system that associates type identifiers with their definitions. That way the identifiers can be replaced by their definitions wherever necessary.

Definition 10.2.4 Relations of the form τ = T, where τ is a type identifier and T is a type expression, are said to be type definitions. A type constraint system, C, is defined as follows:

1. The empty set, , is a type constraint system.

2. If C is a type constraint system and τ is a type identifier that does not appear in C or T, then C ∪ {τ = T} is a type constraint system.
For example, in type checking the body of the program `PointExample`, $C$ would contain type definitions for both `PointType` and `PtClassType`.

Recall that a type constraint system for the polymorphic lambda calculus kept track of subtyping assumptions rather than type definitions. We retain the same name here because we will add subtyping definitions to type constraint systems when we add polymorphism to the language.

The function $C(T)$ returns the type expression formed by replacing all type identifiers in $T$ by their definitions in $C$ (recursively, if necessary).

**Definition 10.2.5** $C(T)$, for $T$ a type expression, is defined inductively as follows:

1. If $t$ is a type identifier, then $C(t) = C(t)$ if $(t = T) \in C$, otherwise $C(t) = t$.
2. If $C$ is a type constant, then $C(C) = C$.
3. $C(T_1 \times \ldots \times T_n \rightarrow T_{n+1}) = C(T_1) \times \ldots \times C(T_n) \rightarrow C(T_{n+1})$.
4. $C(\text{Ref } T) = \text{Ref } C(T)$.
5. $C(\text{ObjectType } RT) = \text{ObjectType } C(RT)$.
6. $C(\text{VisObjectType}(RT_1, RT_m)) = \text{VisObjectType}(C(RT_1), C(RT_m))$.
7. $C(\text{ClassType}(RT_1, RT_m)) = \text{ClassType}(C(RT_1), C(RT_m))$.
8. $C(\{ l_1 : T_1; \ldots; l_n : T_n \}) = \{ l_1 : C(T_1); \ldots; l_n : C(T_n) \}$

The function $C(T)$ is guaranteed to terminate because the restrictions in clause (2) of the definition of $C$ rule out recursive (or mutually recursive) type definitions. If all free identifiers of $T$ are contained in the domain of $C$, then $C(T)$ will be a type expression with no free identifiers. A recursive definition is necessary because type definitions occurring later in a program may use type identifiers defined earlier.

We begin by introducing rules that process declarations. Processing a type or constant definition or variable declaration results in adding type information about the new identifiers introduced into the type constraint system, $C$, or type environment, $\mathcal{E}$.

The symbol $\diamond$ is used to separate a declaration in \texttt{SOOL} from the resulting constraint system and type environment. A judgement of the form $C, \mathcal{E} \vdash \text{Dec} \diamond C', \mathcal{E}'$ asserts that, under the assumptions in $C$ and $\mathcal{E}$, processing the declaration $\text{Dec}$ results in an enhanced type constraint system, $C'$, and type
10.2 Syntax and type-checking rules

This notation makes it clear that the result of processing a declaration is changes to the type constraint system and type environment.

The processing rules for declarations are written similarly to the type-checking rules for the lambda calculus given in Chapter 8:

\[ \text{AxiomName} \quad \text{C, } \mathcal{E} \vdash \text{Dec} \circ C', \mathcal{E}' \]

if the rule is not dependent on hypotheses, or

\[ \text{RuleName} \quad \frac{C, \mathcal{E} \vdash \text{Dec}_1 \circ C_1, \mathcal{E}_1, \ldots, C_{n-1}, \mathcal{E}_{n-1} \vdash \text{Dec}_n \circ C_n, \mathcal{E}_n}{C, \mathcal{E} \vdash \text{Dec} \circ C_n, \mathcal{E}_n} \]

where the conclusion is written below the line, and the hypotheses are above the line. Typically, the expressions appearing in judgements above the line in rules are subexpressions of the expressions appearing in judgements below the line.

We read the rules starting from the bottom and then proceeding from left to right across the top. For example, the sample rule can be read as stating, “from the type assumptions in \(C\) and \(\mathcal{E}\), processing \(\text{Dec}\) results in a richer collection of type assumptions \(C_n, \mathcal{E}_n\), if processing \(\text{Dec}_1\) from \(C\) and \(\mathcal{E}\) results in \(C_1, \mathcal{E}_1, \ldots\), and processing \(\text{Dec}_n\) from \(C_{n-1}, \mathcal{E}_{n-1}\) results in \(C_n, \mathcal{E}_n\).”

The type-checking rules for declarations can be found in Figure 10.7. They are relatively straightforward. By rule TypeDef, processing a type definition results in adding the definition to \(C\), but no changes to the type environment, \(\mathcal{E}\). In contrast, rule ConstDef indicates that processing a constant declaration results in adding the constant name and its type to the type environment, but no change to the type constraint system, \(C\). A list of type or constant declarations is processed by processing the first declaration to get an enriched type environment, and then processing the rest of the declarations recursively to get the final type environment.

Type-checking rules for non-declarations are written similarly. A judgement of the form \(C, \mathcal{E} \vdash \text{exp}: T\) asserts that, under the type assumptions in \(C, \mathcal{E}\), the expression \(\text{exp}\) has type \(T\). We introduce a special notation to make writing type-checking rules easier. If \(\text{RType}\) is a record type of the form \([x_1: T_1; \ldots; x_n: T_n]\), we let \(\text{RType}^\ref = [x_1: \text{Ref } T_1; \ldots; x_n: \text{Ref } T_n]\).

We will give type-checking rules for each of the other syntactic categories of \(S\text{OOL}\). We start by providing the type-checking rules for expressions in Figure 10.8.

Most of these rules are straightforward. By rule Identifier, an identifier can be deduced to have the type that is specified for it in the type environment,
### Figure 10.7  Type checking rules for declarations.

<table>
<thead>
<tr>
<th>Rule</th>
<th>Context</th>
</tr>
</thead>
<tbody>
<tr>
<td>TypeSec</td>
<td>( C, E \vdash T \text{DefLst} \circ C', E' )</td>
</tr>
<tr>
<td></td>
<td>( C, E \vdash \text{type } T \text{DefLst} \circ C', E' )</td>
</tr>
</tbody>
</table>
|TypeDefLst| \( C, E \vdash t = T \circ C_1, E_1 \quad C_1, E_1 \vdash T \text{DefLst} \circ C_2, E_2 \)
|          | \( C, E \vdash t = T; T \text{DefLst} \circ C_3, E_3 \)            |
|TypeDef   | \( C, E \vdash t = T \circ C \cup \{t = T\}, E \quad \text{if } t \notin \text{dom}(C) \) |
|ConstSec  | \( C, E \vdash \text{const } C \text{DefLst} \circ C', E' \)        |
|          | \( C, E \vdash \text{const } C \text{DefLst} \circ C', E' \)    |
|ConstDefList| \( C, E \vdash \text{id}: T = E \circ C, E_1 \quad C, E_1 \vdash C \text{DefLst} \circ C, E_2 \)
|          | \( C, E \vdash \text{id}: T = E; C \text{DefLst} \circ C, E_2 \) |
|ConstDef  | \( C, E \vdash E: T \quad \text{id} \notin \text{dom}(E) \)          |
|          | \( C, E \vdash \text{id}: T = E \circ C, E \cup \{\text{id}: T\} \) |

\( E \). In reporting the type of the identifier, all of the type identifiers in the type are replaced by their definitions in \( C \). This is done by writing the resulting type as \( C(T) \). Because all user-introduced names will be replaced by their definitions during type-checking, \textit{SOOL} uses structural rather than name equivalence of types [Lou93].

By rule \textit{Const}, built-in constants are assigned the type that is provided in the language definition. The expression \( () \) is the only value of type \textit{Void}. The expression \textit{nil} is unusual in that it can have any object type. Which type it is assigned will depend on context. If \( E \) has type \( T \), then \textit{ref} \( E \) is a reference to \( E \), and hence has type \textit{Ref} \( T \). If \( E \) has type \textit{Ref} \( T \), \textit{i.e.}, is a variable holding values of type \( T \), then \textit{val} \( E \) has type \( T \).

Type checking function definitions is only a bit more complicated. By the \textit{Function} rule, a function has type \( T_1 \times \ldots \times T_n \rightarrow T_{n+1} \) iff the body has type \( T_{n+1} \) under the added assumptions (in \( E \)) that the parameters have types \( T_1 \) through \( T_n \). The assumption on the types of formal parameters is necessary since the function body may involve the formal parameters. Similarly if a function has type \( T_1 \times \ldots \times T_n \rightarrow T_{n+1} \), and the actual parameters have types \( T_1 \) through \( T_n \), the \textit{Application} rule indicates that the result of applying the function to the actual parameters has type \( T_{n+1} \).
10.2 Syntax and type-checking rules

\[ C, E \cup \{ \text{id}: T \} \vdash \text{id}: C(T) \]

\[ C, E \vdash \text{c}: C \]

where \( C \) is the pre-assigned type for built-in constant \( c \).

\[ C, E \vdash (): \text{Void} \]

\[ C, E \vdash \text{nil}: \text{ObjectType} \]

\[ C, E \vdash \text{E}: \text{Ref} T \]

\[ C, E \vdash \text{E}: \text{Ref} T \]

\[ C, E \vdash \text{inst: IV, } C', E' \vdash \text{meth: M} \]

\[ C, E \vdash \text{class inst, meth: ClassType(IV,M)} \]

where

- \( C' = C \cup \{ \text{SelfType = VisObjectType(IV''d,M)} \} \),
- \( E' = E \cup \{ \text{self: SelfType, close: SelfType \rightarrow ObjectType M} \} \), and
- \( \text{SelfType does not occur in IV or M} \).

\[ C, E \vdash \text{E: ClassType(IV,M)} \]

\[ C, E \vdash \text{new E: ObjectType M} \]

\[ C, E \vdash \text{E: ObjectType } [\text{m_1: T_1, ..., m_n: T_n}] \]

\[ C, E \vdash \text{E \leftarrow m_i: T_i} \]

Figure 10.8 Type-checking rules for expressions of S O O L, part 1.
The type checking of classes is more challenging. A class of the form `class inst, meth` has type `ClassType(IV, M)`, if two conditions hold. First, the record of instance variables, `inst`, must have type `IV`. Second, the record of methods, `meth`, must have type `M`.

There is a complication, however. Because method bodies may contain occurrences of `self`, we need to type check methods in a context that allows us to assign a type to all expressions involving `self`. We assign the type `SelfType` to `self`, where `SelfType = VisObject`\(\text{Type}(\text{IVref}, \text{M})\)\footnote{Ref before the type of each field type. That is, if \(\text{IV} = \{\text{l_1:T_1}; \ldots; \text{l_n:T_n}\}\), then \(\text{IV} = \{\text{l_1:Ref T_1}; \ldots; \text{l_n:Ref T_n}\}\).}

While `inst` with type `IV` is the record of initial values of instance variables, the type `SelfType` of `self` includes the types of the instance variables themselves. That is, if `x` is an instance variable and the class contains an initial value of `x` of type `T`, then the corresponding type in the visible object type is `Ref T`.

Because `self` will be assigned type `SelfType` in \(\mathcal{E}\), we will be able to type check expressions involving sending messages to `self` and extracting instance values from `self` inside the bodies of methods. (The type checking rules for these operations will be introduced in the next figure.) As discussed earlier, `close` is introduced as a function to convert values (like `self`) from `SelfType` to `ObjectM`. Its typing is inserted into \(\mathcal{E}'\) in order to type check the bodies of methods.

Applying rule `Class` to type check a class definition requires that we know the types of the instance variables and methods before we begin type checking, because this information must be added to \(\mathcal{C}'\) and \(\mathcal{E}'\) when type checking methods of the class. Our syntax for classes ensures that each instance variable and method definition inside a class does appear with its type information, so no difficulties arise with having to guess what these types might be if `inst` and `meth` are written as record expressions. However it is possible that `inst` or `meth` could be provided in some other way, e.g., as a formal parameter or the value of a variable, so that we only know a type, rather than the most explicit type.

For example, we might only have partial information on the instance variables or methods in type-checking an expression. Suppose \(\text{m}\) is a method that is never referenced in any of the other methods of the class, and let \(\text{M}'\) be the type of the record consisting of all of the methods of the class aside from \(\text{m}\). We could then type check the methods under the assumption that `SelfType= VisObject`\(\text{Type}(\text{IVref}, \text{M}')\)` rather than `VisObject`\(\text{Type}(\text{IVref}, \text{M})\).
If the type checking succeeds, then the resulting type of the class would be \texttt{ClassType(IV, M')} rather than \texttt{ClassType(IV, M)}.

Of course the type of objects generated from the class with this typing would be \texttt{ObjectType M <: ObjectType M}. Thus there is little advantage to obtaining a weaker type for the class. As a result, a type-checking algorithm will normally deduce the most specific type possible for the class. Most languages do not allow a programmer to pass a record of methods or even an individual method into a class. Thus this issue rarely arises.

Finally, note the restriction that \texttt{SelfType} may not occur in the types of either instance variables or methods. “Visible object types” are not really part of S0OL, they are simply introduced as an aid in type-checking programs. As a result it may not appear explicitly anywhere in a program, though it is used when type checking expressions involving \texttt{self}.

Expressions of the form new \texttt{E} generate new objects from a class. The new object has an object type with methods corresponding to those given in the class type of \texttt{E}. By rules \texttt{Message}, a message send has the type of the corresponding method of the object type.

The type-checking rules for subclasses, records, and blocks are given in Figure 10.9.

The type-checking rule for subclasses is longer than any we have seen so far, but it is simpler than it may first appear. Recall from Chapter 6 that a subclass may add new instance variables and methods, but that the types of instance variables may not be changed, and that the types of overridden methods may only be replaced by subtypes. Thus parameters may only change in a contravariant way, and return types in a covariant way.

As a result, to type-check the expression

\begin{verbatim}
class inherits E modifies l_1, ..., l_m(inst, meth),
\end{verbatim}

we must first determine the type of superclass, \texttt{E}, the new instance variables in \texttt{inst}, and the new methods in \texttt{meth}. We also must ensure that the names of the new instance variables do not overlap with those of \texttt{E}, and that if any of the new method names overlap with those of \texttt{E}, then the new types in the subclass will be subtypes of those for the superclass \texttt{E}. (We do not bother to check whether instance variable and method identifiers overlap since they are referred to in different ways.)

If the type of the subclass is expected to be \texttt{ClassType(IV, M)}, then we determine the types of the new methods under the assumption that the type of \texttt{self} is \texttt{SelfType \rightarrow VisObjectType(IV', M)}, and that the type of \texttt{close} is \texttt{SelfType \rightarrow ObjectType M}, as with the type checking rule for classes.
Figure 10.9 Type-checking rules for expressions of SOOL, part 2.
10.2 Syntax and type-checking rules

The notation $R \oplus R'$ represents the combination of two record types. The result is a record type with all of the fields of both record types. If there are overlapping labels, then the type associated with each such label in the result is the corresponding type from $R'$. Thus we can think of $\oplus$ as a right dominant operation that combines record types.

The next two rules provide type-checking rules for message sends and instance variable extraction from elements of type $\text{VisObject}Type(IVR,M)$. Recall that the type of self in type checking methods in classes and subclasses is of the form $\text{VisObject}Type(IVR,M)$. The result of extracting an instance variable has the same type as the corresponding field of the type of the record of instance variables. The type checking rule for message sends to visible object types is essentially the same as for object types and is based only on the type of the methods.

The type of a record, $E$, is determined by the types associated with the individual fields. Notice that record expressions include type expressions along with the values for each field.

A block consists of the declarations and body of either a program or function. By rule $\text{Block}$, it is type checked by processing the declarations to obtain a new type environment, which is then used to type check the body and return expression. The body should be a statement, and the type of the return expression determines the type of the block. The type of the block should not involve any local type definitions as they will not be visible outside of the scope of the block.

By using the rule $\text{Identifier}$, we end up typing expressions by first removing all user-introduced type identifiers. That is, when $x:T$ is included in $E$, we assigned $x$ the type $C(T)$ rather than just $T$. However, as a convenience, we would also like to report the types of expressions with type expressions that use these abbreviations. By rule $\text{Type Abbrev}$, we can provide an expression with a type expression involving abbreviations once we determine that the expression has the type obtained by removing the abbreviations.

Figure 10.10 contains type-checking rules for statements. Recall that well-typed statements have type $\text{Command}$. The special constant, $\text{nop}$, is a command, by rule $\text{No Op}$. By rule $\text{Assn}$, an assignment statement is well-typed if the type of the left side is a reference to the type of the expression appearing on the right side. If-then-else and while statements are well-typed if the guard expressions have type $\text{Boolean}$ and the component statements are themselves well-typed. Individual statements in statement lists should be well-typed. However, unlike the lambda calculus, we do not allow non-statements, i.e., expressions with
Figure 10.10 Type-checking rules for statements of SOOL.

10. SOOL, a Simple Object-Oriented Language

No Op
\[ C, E \vdash \text{nop}: \text{Command} \]

Assn
\[ C, E \vdash \text{id}: \text{Ref T}, \quad C, E \vdash E: T \]
\[ C, E \vdash \text{id} := E: \text{Command} \]

Cond
\[ C, E \vdash E: \text{Boolean}, \quad C, E \vdash S_1: \text{Command}, \quad C, E \vdash S_2: \text{Command} \]
\[ C, E \vdash \text{if } E \text{ then } \{ S_1 \} \text{ else } \{ S_2 \}: \text{Command} \]

While
\[ C, E \vdash E: \text{Boolean}, \quad C, E \vdash S: \text{Command} \]
\[ C, E \vdash \text{while } E \text{ do } \{ S \}: \text{Command} \]

StmtList
\[ C, E \vdash S_1: \text{Command}, \quad C, E \vdash S_2: \text{Command} \]
\[ C, E \vdash S_1; S_2: \text{Command} \]

Program
\[ C, E \vdash \text{Block}: \text{Command} \]
\[ C, E \vdash \text{Program id; Block}: \text{Command} \]

type different from \text{Command}, to occur in statement lists. Finally, a program is well-typed if the block it contains is well-typed.

We have now defined type-checking rules for the complete language, but we have not yet mentioned subtyping. Rather than using the very simple invariant subtyping rules used in Java and early versions of C++, we instead use the more general rules discussed in Chapter 5. The SOOL subtyping rules can be found in Figure 10.11.

Because \( C \) does not contain subtyping assumptions, we need \( C \) only to expand user-defined type definitions. Rule \text{TypeDef} \( \prec \) specifies that to determine if two types are subtypes under the assumptions in \( C \), simply expand the type definitions and determine whether the expanded types are subtypes using no assumptions.

The remaining subtyping rules make no direct use of \( C \). Rule \text{Reflex} \( \prec \) specifies that a type is always a subtype of itself, while rule \text{Trans} \( \prec \) asserts that subtyping is closed under transitivity. Subtyping for functions is covariant in the argument type and contravariant in the argument types by rule \text{Function} \( \prec \), just as we would expect based on our earlier discussions of subtyping for function types in Section 5.1.2.
TypeDef $\preceq$: \[
\frac{\emptyset \vdash C(\emptyset) \preceq C(T)}{C \vdash S \preceq T}
\]

Reflex $\preceq$: 

Trans $\preceq$: \[
\frac{C \vdash S \preceq T, \quad C \vdash T \preceq U}{C \vdash S \preceq U}
\]

Function $\preceq$: 

Record $\preceq$: \[
\frac{m \leq n \text{ and } C \vdash S_i \preceq T_i \text{ for all } 1 \leq i \leq m}{C \vdash \langle 1_1 : S_1, \ldots, 1_n : S_n \rangle \preceq \langle 1_1 : T_1, \ldots, 1_m : T_m \rangle}
\]

Object $\preceq$: \[
\frac{C \vdash \text{ObjectType} \preceq \text{RType}}{C \vdash \text{ObjectType}' \preceq \text{RType'}}
\]

Subsumption \[
\frac{C, E : S, \quad C \vdash S \preceq T}{C, E : T}
\]

**Figure 10.11** Subtyping rules for SOOL.

Rule Record $\preceq$ indicates that record subtyping includes both width and depth subtyping. Finally, rule Object $\preceq$ specifies that two object types are subtypes if the types of their records of methods are subtypes.

The last rule, Subsumption, is actually a type-checking rule rather than a subtyping rule. It states that if an expression has a type, then it can also be assigned any supertype of that type. In Chapter 2 we defined types ColorCellType and CellType where ColorCellType $\preceq$ CellType. If colorCell has type ColorCellType, then by the Subsumption rule it also has type CellType.

### 10.2.4 A type checking example

In the previous section we presented a number of formal type-checking rules for SOOL. In this section we present an example showing how these rules can be used to type-check SOOL programs. At the end of the section we in-
lude a brief discussion of how the type-checking rules can be used to create a type-checking algorithm for SOOL.

We show how to use the type-checking rules by verifying that the program PointExample in Figure 10.4 is type-correct. We start with the program as a whole, and with a type constraint system and type environment, $C_0$, $E_0$, that are both empty.

It follows from the $Program$ rule in Figure 10.10 that the program as a whole is well-typed iff its block has type $Command$. Looking back at Figure 10.9, we see that a block is type-checked by processing the type and constant definitions to get an enriched type constraint system and type environment. Then the body of the block is type-checked to make sure that it is well-typed. Finally the type of the entire block is the type of the return expression. In this case the return expression is $nop$, which by rule $NoOp$ has type $Command$. Thus the block as a whole will be well-typed if we can first successfully type-check the constant definitions to obtain an enriched type environment, and then use that enriched type environment to ensure the body of the block is well-typed.

There are two type definitions in the program. By the type-checking rules, both $PointType$ with its definition, and $PtClassType$ with its definition are added to $C_0$, resulting in

$$C_1 = C_0 \cup \{ PointType = ObjectType \mid \ldots \},$$

$$PtClassType = ClassType(\ldots)$$

There are only two constant definitions in the program. The first is of the constant $Point$, which is defined to be a class of type $PtClassType$. We must show that the class expression in Figure 10.12 has type $PtClassType$:

For simplicity, we will use the abbreviations:

$$PIV = \{ \mid x: \text{Integer}; y: \text{Integer} \}$$

$$PIVR = \{ \mid x: \text{Ref Integer}; y: \text{Ref Integer} \}$$

$$PM = \{ \mid \text{move: Integer} \times \text{Integer} \rightarrow \text{Integer};$$

$$\text{getx: Void} \rightarrow \text{Integer};$$

$$\text{gety: Void} \rightarrow \text{Integer} \}$$

We begin by showing that

$$C_1, E_0 \vdash \{ \mid x: \text{Integer} = 0; y: \text{Integer} = 0 \} : PIV.$$
### 10.2 Syntax and type-checking rules

```plaintext
class (∥
    x: Integer = 0;
    y: Integer = 0;
∥,
    move: Integer × Integer → Command =
        function(dx: Integer, dy: Integer): Command is
        { self.x := (val self.x) + dx;
          self.y := (val self.y) + dy;
          return nop};

    getx: Void → Integer =
        function(): Integer is
        { return val self.x };

    gety: Void → Integer = function(): Integer is
        { return val self.y }
∥)

Figure 10.12  Class definition from PointExample.

But this is easy as the Constant rule implies that $C_1, \mathcal{E}_0 \vdash 0: \text{Integer}$.

The desired typing of the record of instance variables follows from rule Record.

We must next show that the record of methods in Point has the type PM
under the type assumption

$$\mathcal{E}_1 = \mathcal{E}_0 \cup \{\text{self: SelfType, close: SelfType} \rightarrow \text{ObjectType PM}\}$$

and type constraint system

$$C_2 = C_1 \cup \{\text{SelfType} = \text{VisObjectType(PIVR, PM)}\}.$$

By the Record rule, it is sufficient to show that each individual method has
the appropriate type. We will only type check the move method, as the others
are similar, but simpler.

We must show that the function expression,

```plaintext
function(dx: Integer, dy: Integer): Command is {
```

\footnote{We did not list all constants of the language in $\mathcal{E}$, but instead assume familiar constants (like numerals) have the obvious types.}
self.x := (val self.x) + dx;
self.y := (val self.y) + dy;
return nop 
}

has type Integer \times Integer \rightarrow Command under assumption \mathcal{C}_2, \mathcal{E}_1.

We type check the function using the Function rule by checking the body under type constraint \mathcal{C}_2 and type assumption
\[
\mathcal{E}_2 = \mathcal{E}_1 \cup \{ dx: \text{Integer}, dy: \text{Integer} \}.
\]
The first line of the function is an assignment. By rule Inst Vble and the assumption on the type of self in \mathcal{E}_2, it follows that
\[
\mathcal{C}_2, \mathcal{E}_2 \vdash \text{self.x: Ref Integer}.
\]
By the Value rule, the type of val self.x is Integer. The type of dx is also Integer by the Identifier rule, because dx: Integer is in \mathcal{E}_2.

At this point we must cheat a bit since we did not include a type-checking rule for operations like "+". The operation "+" should actually be written as a prefix function Plus with type Integer \times Integer \rightarrow Integer. Then val self.x + dx abbreviates Plus(val self.x, dx), which, by the Application rule, has type Integer. Now the Assignment rule allows us to infer from \mathcal{C}_2, \mathcal{E}_2 that statement
\[
\text{self.x := (val self.x) + dx;}
\]
has type Command, because the left side has type Ref Integer, and the right side has type Integer. Showing the other assignment statement has type Command is similar. By the rule StmtList, the sequence of the two assignments also has type Command.

The expression nop has type Command by rule No Op, so by rule Block the whole function body has type Command when typed with respect to \mathcal{C}_2, \mathcal{E}_2. By rule Function, the function expression associated with method move has type Integer \times Integer \rightarrow Command when typed with \mathcal{C}_2, \mathcal{E}_1.

Similar arguments show that the other two method bodies have the desired types using the assumptions in \mathcal{C}_2, \mathcal{E}_1. Hence the record of methods has type \text{PM} under \mathcal{C}_2, \mathcal{E}_1. Finally the Class rule implies that the entire Point body has type PtClassType. By rule ConstDef,
\[
\mathcal{E}_3 = \mathcal{E}_0 \cup \{ \text{Point: PtClassType} \}
\]
results from processing the definition of PointClass, starting with \mathcal{C}_1, \mathcal{E}_3.

The other definition in the program is
pt: Ref PointType = ref nil;

By rule Nil, nil can be assigned type PointType (this is the one place where we use information about context in type checking). By rule Reference, ref nil has type Ref PointType. Therefore by rule ConstDef,

$$\mathcal{E}_4 = \mathcal{E}_3 \cup \{pt: \text{Ref PointType}\}$$

results from processing the definition of pt, starting with $\mathcal{C}_1, \mathcal{E}_3$.

By rules ConstSec and ConstDefList and the results of processing the two constant definitions above, processing the const section of the program results in static type environment $\mathcal{E}_4$. We use this environment along with $\mathcal{C}_1$ to type check the body of the program.

The first assignment statement is well-typed because new Point has type PointType by the New rule. The message send of pt $\leftarrow$ move has type Integer $\times$ Integer $\rightarrow$ Command by rule Message, and, by rule Application, pt $\leftarrow$ move(3, 2) has type Command. As before, nop has type Command by rule No Op. Combining rules StmtList and Block, it follows that the block consisting of the program declarations and code has type Command. Thus by rule Program, the entire program is well-typed.

As the description above shows, type checking a program is fairly mechanical. All constants are declared with their types, so we need only verify that the definitions are of the appropriate types. The expressions representing class and function definitions include sufficient type information in the expressions that the type can be extracted from the expression. For classes (and subclasses) this information can be extracted from the types of the record of instance variables and the record of methods. The types of individual components of records are included in record expressions, so all necessary type information can be extracted from those subexpressions. Then it is only required to check that the subexpressions have the appropriate types.

Most of the other expressions have types that can be simply composed from the types of their subexpressions. For example, the type of a function application will always be the return type of the function, the type of an expression of the form $\text{val} \ E$ is obtained by stripping the Ref off of the type of $E$, etc. Identifiers' types can be found in the type environment and constants' types are predefined.

That really leaves only nil expressions as problematic. Their types may depend on the context, as with the definition of pt in the sample program. There it was necessary to look at the type declared with the constant in order to determine which type nil should be assigned (it was PointType). We
could get around this by creating a new type, Bottom, which is a subtype of all object types, and give nil type Bottom. The Subsumption rule could then be used to promote the type of nil to whatever object type is needed in the context. We will forgo adding this new type and simply presume enough information is available from context to allow us to infer an appropriate type for nil.

10.3 Summary

In this chapter we provided a formal definition of the language SOOL via a context-free grammar and type-checking rules. As noted earlier, we will usually use an abbreviated version of the language that is closer to existing common object-oriented languages, as it will be easy to algorithmically convert from the simplified language to the stricter formal language.

Simply providing a formal language definition and type-checking rules does not guarantee that a statically typed language is type-safe. For example, how do we know that at run time an expression of some type T will actually hold a value of type T? How do we know that all message sends at run time will be to objects that have methods that can execute in response to the message?

We will address such questions after we have defined the semantics of SOOL. In the next chapter, we begin our investigation of a translational semantics for SOOL.
In this chapter we provide a first approximation to the semantics of an object-oriented language via a translation into the language $\lambda^\pi$. We will see that our first simple translation works well through the definition of classes, but fails to work with subclasses. Nevertheless, it provides a useful first step toward a correct definition. In the next chapter we show how to modify this translation in order to model subclasses correctly.

In this and the following sections we will use the language Sool introduced in the last chapter as our source language. We will use the fully expanded language, rather than the abbreviated version, both because it matches the formal description of syntax and type-checking rules given in the last chapter, and because it will be easier to translate.

As discussed earlier, we will presume that all instance variables are hidden from clients (protected in Java and C++ notation) and all methods are visible (public). In Section 14.4 we will refine the notation, type-checking rules, and semantics so that we can allow the hiding of selected methods as well.

11.1 Representing objects at runtime

Before launching into a description of the formal translation from Sool to $\lambda^\pi$, we discuss the representation of objects in real object-oriented systems.

One possible representation for objects is as pairs of a record of instance variables and a record of methods. We illustrate this with an object generated by class CellClass from Figure 10.2 in the last chapter. The record of instance variables for the object has a single field, which represents a location to hold the value of the instance variable $x$. The record of methods has fields for each of the methods, \textit{get}, \textit{set}, and \textit{bump}. In these methods, references
to \texttt{self.x} would be replaced by the address of the slot in the record holding the value of \texttt{x}.

Specializing the methods to directly access the instance variables provides for more efficient access to these instance variables in methods. However, it results in great waste of space in class-based systems, as a separate copy of each method must be created for each object existing at runtime because each method has been specialized to only reference its own instance variables. Such a representation may make sense for a delegation-based object-oriented system, but, because of the space inefficiencies, it is not used in implementations of any major class-based languages that we know of. Instead, only one copy of the methods of a class is created, and it is shared by all objects generated by the class. In this case, a convenient representation for objects is as a record of its instance variables along with a pointer to the method suite of the object. Access to the record of instance variables from a method body is obtained by an indirect access of one sort or another.

In our formal representation, we will provide access to instance variables inside method bodies by requiring the record of instance variables to be passed as a parameter to the methods. This could be done in two ways. One might add an extra parameter representing the record of instance variables to each method of the class. Alternatively, we could parameterize the entire record of methods by the record of instance variables. From a formal point of view, these are essentially equivalent. We will choose the second of these for notational simplicity. It might be interesting to investigate which choice would provide the best average execution-time efficiency.

In the next section we present our “implementation” of objects by providing a translation from \texttt{SOOL} into \texttt{A<}. That translation results in a representation of objects as pairs. The first component of the pair is a record of instance variables. The second component is a function that takes a record of instance variables as a parameter, and returns the message suite specialized to that record of instance variables. (See details below.) While our objects will actually contain the parameterized method suites as part of the object representation, it should be clear that these functions are identical for all objects created from the same class, and hence they could be stored in a central location (e.g., with the class) and accessed via references from the object.

As suggested earlier, it would be possible to specialize the method suite to each particular object, essentially by compiling into the method suite the references to each instance variable. However, we prefer the representation where the method suites for all objects created from a given class are identical, and thus can be shared.
11.2 Modeling SOOL types in $\Lambda^P_\leq$

In this section we provide a formal mapping from SOOL types to $\Lambda^P_\leq$ types in which the representation of objects corresponds to the implementations described in the previous section. The translation function, denoted $T_c[]$, will provide the translation of both types and expressions of the language SOOL to types and expressions of $\Lambda^P_\leq$. We include the type constraint $C$ as a subscript, because the translation may involve type variables that should be replaced by the translations of their definitions in $C$. The translation of types is given in Figure 11.1.

![Figure 11.1](https://example.com/figure111.png)

**Figure 11.1** Translation of types of SOOL to types in $\Lambda^P_\leq$.

### 11.2 Modeling SOOL types in $\Lambda^P_\leq$

In this section we provide a formal mapping from SOOL types to $\Lambda^P_\leq$ types in which the representation of objects corresponds to the implementations described in the previous section. The translation function, denoted $T_c[]$, will provide the translation of both types and expressions of the language SOOL to types and expressions of $\Lambda^P_\leq$. We include the type constraint $C$ as a subscript, because the translation may involve type variables that should be replaced by the translations of their definitions in $C$. The translation of types is given in Figure 11.1.

Constant types are translated to fixed types in $\Lambda^P_\leq$ with no free type identifiers. For example we could let $T_c[Integer] \triangleq Integer$, where $Integer$ is the type of $\Lambda^P_\leq$ representing the collection of integers, and $T_c[Void] \triangleq Void$.

Because $\Lambda^P_\leq$ includes both product and function types, the function types of SOOL are translated by translating the component types and then combining them using the corresponding type constructors in $\Lambda^P_\leq$. Record types of SOOL are similarly translated by creating the corresponding record types in $\Lambda^P_\leq$, replacing the types of fields by their translations using $T_c[]$.

To simplify notation in the definition of object types, we introduce the following abbreviations. If $IV = \{l_1: T_1; \ldots; l_n: T_n\}$, let $IV_{ref} = \{l_1: Ref T_1; \ldots; l_n: Ref T_n\}$. Similarly, if $inst = \{l_1: T_1:= E_1; \ldots; l_n: T_n:= E_n\}$, then let $inst_{ref} = \{l_1: Ref T_1:= ref E_1; \ldots; l_n: Ref T_n:= ref E_n\}$. This notation is useful because classes will provide only the initial values of instance variables, while objects will contain variables that are initialized to those values.
Thus if $IV$ is the type of the record of initial values given by a class, $IV^\text{ref}$ will be the type of the actual record of instance variables for objects generated by that class.

The types in Figure 11.1 include the type $\text{VisObjectType}(IVR, M)$ that was introduced in the last chapter. Again, think of $\text{VisObjectType}(IVR, M)$ as presenting the view of an object as seen from the “inside”, with both the instance variables and methods accessible.

As described earlier, we represent objects as pairs in which the first component is the record of instance variables and the second component is a function taking the record of instance variables and returning the record of methods that access these instance variables. Let $IVR$ be the type of the record of instance variables of an object and let $M$ be the type of the record of methods of the same object. As presented in Figure 11.1, the translation of $\text{VisObjectType}(IVR, M)$ is obtained by using $\mathcal{T}_C[\ ]$ to translate the types $IVR$ and $M$, and then creating the type of pairs whose first component has type $\mathcal{T}_C[IVR]$ and whose second component has type $\mathcal{T}_C[IVR] \rightarrow \mathcal{T}_C[M]$. Note that if $IV$ is the type of the record of initial values for instance variables in a class and $M$ is the type of its methods, then the internal view of objects built from the class will have type $\text{VisObjectType}(IV^\text{ref}, M)$. Thus we replace the $IV$ in the class definition by $IV^\text{ref}$ in the actual object.

With this representation we can obtain access to instance variables of an object by accessing the appropriate field of the first component of the pair representing the object. We obtain access to a method of the object by first applying the second component of the pair (the parameterized methods) to the first (the record of instance variables). Now that the record of methods has been provided access to the instance variables, it is only necessary to select the appropriate method.

We illustrate the translation with the example of objects generated from the class $\text{CellClass}$. The type of the values of instance variables in objects generated from $\text{CellClass}$ is

$$\text{CellIV} \triangleq \{x: \text{Integer}\}$$

while the type of the record of methods is

$$\text{CellM} \triangleq \{\text{get: Void} \rightarrow \text{Integer};\text{set: Integer} \rightarrow \text{Void};\text{bump: Void} \rightarrow \text{Void}\}$$

The type of $x$ in $\text{CellIV}$ is $\text{Integer}$ rather than $\text{Ref Integer}$ because it
11.2 Modeling SOQL types in \( \Lambda C \)

refers only to the values stored in the instance variables. The references will be added in forming the meaning of the object type.

Because CellIV and CellM include only non-object types, their translations are simple. Let CellIV \( \triangleq T_C[CellIV] \), RfCellIV \( \triangleq T_C[CellIV^{ref}] \) and CellM \( \triangleq T_C[CellM] \). Then

\[
\text{CellIV} = \{ |x: \text{Integer}| \}, \\
\text{RfCellIV} = \{ |x: \text{Ref Integer}| \},
\]

and

\[
\text{CellM} = \{ |\text{get: Void} \rightarrow \text{Integer}; \text{set: Integer} \rightarrow \text{Void}; \text{bump: Void} \rightarrow \text{Void}| \}
\]

The types of instance variables in the translations have been converted to the corresponding reference types. The structures of the original and translated types of methods are identical, with base types like Void and Integer of SOQL replaced by the corresponding types, Void and Integer, of \( \Lambda C \).

It follows that

\[
T_C[\text{VisObjType}(\text{CellIV}^{ref}, \text{CellM})] = \text{RfCellIV} \times (\text{RfCellIV} \rightarrow \text{CellM}).
\]

We will use the following abbreviation to simplify writing the interpretations of visible object types:

\[
\text{VisObj}(IV, M) \triangleq IV \times (IV \rightarrow M)
\]

Thus we can now write

\[
T_C[\text{VisObjType}(\text{CellIV}^{ref}, \text{CellM})] = \text{VisObj}(\text{RfCellIV}, \text{CellM}).
\]

If \( o \) has type \( T_C[\text{VisObjType}(\text{CellIV}^{ref}, \text{CellM})] \), then \( \text{inst} \triangleq \text{proj}_1(o) \) represents the record of instance variables of \( o \), while \( \text{inst}.x \) returns (the location of) instance variable \( x \) of \( o \). The expression \( \text{methfun} \triangleq \text{proj}_2(o) \) represents the function which takes a record of instance variables and returns the suite of methods that use that record of instance variables. Thus \( \text{methfun}(o) \) represents the method suite instantiated with \( o \). As a result, \( \text{methfun}(o).\text{bump}(\cdot) \) extracts the method body of \( \text{bump} \) and applies it to \( \cdot \), providing a command that increases the value of instance variable \( x \) of \( o \) by 1.

This leaves us with only the translation of \( \text{ObjectType} \) \( M \) in Figure 11.1 left to explain. Recall that the difference between \( \text{VisObjType}(\text{IVR}, \text{IVM}) \).
and ObjectType M is that instance variables should not be accessible in values of type ObjectType M. If the type of the object were to include the type of the instance variables, then clients could get access to the instance variables of the object.

In particular, we would like it to be possible for several distinct classes (e.g., with different collections of instance variables) to generate objects with the same type. Thus it is important that the type of instance variables not appear in object types.

We introduced existential types exactly so that we could hide this sort of information. Thus the only change we need to make to our representation of VisObjectType (IVR, M) is to hide the type of the instance variables by using an existential type as shown in the definition in Figure 11.1. Using the abbreviation introduced above,

$$T_C[\text{VisObject}Type(\text{CellIV}, \text{CellM})] = \text{VisObj} (\text{RfCellIV}, \text{CellM})$$

while

$$T_C[\text{ObjectType} \text{CellM}] = \exists Y. \text{VisObj} (Y, \text{CellM})$$

Intuitively, the object type admits to the fact that the type of the record of instance variables occurs in the representation of the object type, but it does not reveal exactly what that type is.

We can transform objects from a visible object type into a regular object type by applying the function closeobj, which, when provided with the translations of the types IVR and M of the record of instance variables and methods, takes values of type $$T_C[\text{VisObject}Type(\text{IVR}, \text{M})]$$ to values of type $$T_C[\text{ObjectType} \text{M}]$$ by “packing” the value to hide the type of the record of instance variables:

$$\text{closeobj} \triangleq \lambda IVR. \lambda M. \lambda (\text{vo} : IVR \times (IVR \to M)). \quad \text{pack (IVR, vo) as } \exists Y. Y \times (Y \to M)$$

Thus closeobj is a polymorphic function of type

$$\text{closeobj} : \forall IVR. \forall M. (IVR \times (IVR \to M)) \to (\exists Y. (Y \times (Y \to M)))$$

or, equivalently,

$$\text{closeobj} : \forall IVR. \forall M. \text{VisObj} (IVR, M) \to (\exists Y. \text{VisObj} (Y, M))$$

Going back to our example, the type of objects generated by CellClass will be CellType $\triangleq$ ObjectType CellM, and

$$T_C[\text{CellType}] = \exists Y. Y \times (Y \to \text{CellM}).$$
11.3 Modeling SOOL expressions in $\Lambda_\vartriangleleft^P$.

Sending messages to values of object type is now just slightly more work than with visible types. We must first “open” the existential before applying the second component to the first and then extracting the appropriate method. We show more details in the next section.

The observant reader will have noticed that we have not yet provided a translation for class types. We will postpone all discussions of classes and their types until later in this chapter.

We end this section with the observation that our translation of SOOL types into $\Lambda_\vartriangleleft^P$ types preserves subtyping.

**Theorem 11.2.1** If $S$ and $T$ are types in SOOL such that $C \vdash S \vartriangleleft: T$, then $\emptyset \vdash T_C[s] \vartriangleleft: T_C[t]$ in $\Lambda_\vartriangleleft^P$.

**Proof.** Most cases are trivial because nearly all of the types of SOOL translate directly into similar types in $\Lambda_\vartriangleleft^P$. The use of type abbreviations goes through because the abbreviations are removed by the definition of the translation of type identifiers. The only other interesting case is for object types, as visible object types have no non-trivial subtypes.

Suppose $C \vdash \text{ObjectType } M' \vartriangleleft: \text{ObjectType } M$. By the subtyping rules for SOOL this can only occur if $C \vdash M' \vartriangleleft: M$. By induction on the subtyping derivation, $\emptyset \vdash T_C[M'] \vartriangleleft: T_C[M]$. By the function and product type subtyping rules of $\Lambda_\triangleleft^P$, it follows that $\emptyset \vdash Y \times (Y \rightarrow T_C[M']) \vartriangleleft: Y \times (Y \rightarrow T_C[M])$. Finally, by rule $\text{Exist } \vartriangleleft$,

$$\emptyset \vdash \exists Y. Y \times (Y \rightarrow T_C[M']) \vartriangleleft: \exists Y. Y \times (Y \rightarrow T_C[M]).$$

11.3 Modeling SOOL expressions in $\Lambda_\vartriangleleft^P$.

In this section we provide translations of most of the expressions and statements of SOOL into $\Lambda_\vartriangleleft^P$. We will see later that the translations of classes and subclasses depend on the types of some of these expressions. As a result, our translation is actually based on the typing of the expressions, rather than just the expressions themselves. If $C, E \vdash E: T$ then we write $T_C[C, E: T]$ to represent the translation of that typing of the expression.

Two different typings of an expression may result in two different translations of the expression. Of more concern, because the translation actually depends on the actual typing derivation of a type for an expression, it appears that two different derivations of the same type for an expression may result
in two different translations, even though they both have the same final type. Thus $T_C[C, \xi \vdash e : \tau]$ may not be well-defined if two different derivations of $C, \xi \vdash e : \tau$ result in different translations. As a result it appears we should write $T[C, D, \xi \vdash e : \tau]$, where $D$ is a derivation of the typing, instead.

However, we will see in Section 13.2, that with only very slight restrictions on $SOOL$ to ensure that classes have unique types, the translation will be independent of the actual typing derivation. The reader can check that all of the translations given in this section may easily be shown to be independent of the type of the expression, let alone its typing derivation, though that will change in the next section when we introduce the translation of classes. In order to reflect that dependency we will write $T_C[C, \xi \vdash e : \tau]$ for the translation from $SOOL$ to $\Lambda^F$.

We postpone the translations of the class, subclass and new expressions until later. See Figure 11.2 for the translations of the typings of other $SOOL$ expressions.

The translation of non-object-oriented expressions is relatively straightforward. Identifiers of $SOOL$ are translated to corresponding identifiers of $\Lambda^F$. Constants are translated into closed expressions of $\Lambda^F$ of the appropriate type. (An expression is closed if it has no free identifiers.) For example, the constant 2 of $SOOL$ could be translated to the expression $2$ of $\Lambda^F$. Application of a function to $n$ arguments is translated as the application of the translated function to an $n$-tuple of the translated arguments. Similarly for the definition of a function of $n$ arguments. Translation of val, ref, and record expressions is straightforward. We postpone the discussion of nil until Section 14.2.

Translation of the blocks that form the bodies of functions and also form the main program is a bit tricky because blocks introduce new type and constant definitions. We use the let expressions introduced as abbreviations in Chapters 8 and 9 to make the translation more readable. Type definitions are translated into type let expressions introducing the appropriate names for the translations of the type definitions. Similarly, regular let expressions bind names for constant expressions in the $SOOL$ program. All of these are used in the translation of the body of the Block.

If we did not use the let expressions, the definition would have to be given in the following, less readable, style:
\[ T_c^\mathcal{C}\mathcal{E} \triangleq id \]
\[ T_c^\mathcal{C}\mathcal{E}c \triangleq c \]
\[ T_c^\mathcal{C}\mathcal{E}\lambda \triangleq \langle \rangle \]
\[ T_c^\mathcal{C}\mathcal{E} \triangleq \text{val } T \triangleq \text{val } T_c^\mathcal{C}\mathcal{E}E : T \]
\[ T_c^\mathcal{C}\mathcal{E} \triangleq \text{ref } E : T \]
\[ T_c^\mathcal{C}\mathcal{E}E_1 \ldots \times T_n \triangleq T \]
\[ T_c^\mathcal{C}\mathcal{E}\text{function}(id_1 : T_1, \ldots, id_n : T_n) : T \text{ is Block:} \]
\[ T_1 \times \ldots \times T_n \triangleq \lambda (\langle id_1 : T_1, \ldots, id_n : T_n \rangle). T_c^\mathcal{C}\mathcal{E}' \text{ Block: } T \]
where \( \mathcal{E}' = \mathcal{E} \cup \{ \text{id}_1 : T_1, \ldots, \text{id}_n : T_n \} \)
\[ T_c^\mathcal{C}\mathcal{E}\text{Block: } T \triangleq \text{let } t_1 = T_c^\mathcal{C}[T_1] \text{ in } \ldots \text{ let } t_n = T_c^\mathcal{C}[T_n] \text{ in } \]
\[ \text{let id}_1 : T_c^\mathcal{E}[U_1] = T_c^\mathcal{C}\mathcal{E}E_1 : U_1 \text{ in } \ldots \]
\[ \text{let id}_k : T_c^\mathcal{E}[U_k] = T_c^\mathcal{C}\mathcal{E}E_k : U_k \text{ in } \]
\[ T_c^\mathcal{C}\mathcal{E}E : \text{Command} ; T_c^\mathcal{C}\mathcal{E}E : T \text{ end } \ldots \text{ end } \]
where \text{Block is } t_1 = T_1 ; \ldots ; t_n = T_n
\[ \text{const id}_j : U_1 = E_1 ; \ldots ; \text{id}_k : U_k = E_k \]
\[ \{ \text{S return } E \} \]
\[ T_c^\mathcal{C}\mathcal{E}E \triangleq m : T \triangleq (\text{proj}_1(T_c^\mathcal{C}\mathcal{E}E : T))(\text{proj}_1(T_c^\mathcal{C}\mathcal{E}E : T)).m \]
where \( T = \text{VisObjectType}(\text{IVR}, \text{ObjectType}) \{m_1 : T_1, \ldots, m_n : T_n\} \)
\[ T_c^\mathcal{C}\mathcal{E}E \triangleq m : T \triangleq \text{open } T_c^\mathcal{C}[C, \mathcal{E}E : T] \text{ as } (\text{IR}, \text{vo}) \]
\[ \text{in } (\text{proj}_1(\text{vo})(\text{proj}_1(\text{vo}))).m \]
where \( T = \text{ObjectType} \{m_1 : T_1, \ldots, m_n : T_n\} \)
\[ T_c^\mathcal{C}\mathcal{E}E \triangleq E.1 : T \triangleq \text{proj}_1(T_c^\mathcal{C}[C, \mathcal{E}E : \text{VisObjectType}(\text{IVR}, M)]).J \]
where \( \text{IVR} = \{l_1 : T_1, \ldots, l_n : T_n\} \) and \( 1 \leq k \leq n \).

Figure 11.2 Translation of selected expressions of Sool to expressions in \( \Lambda^\mathcal{C}_\mathcal{E} \).
$T_C[C, E \vdash \text{Block: } T] \triangleq (\lambda t_1 \ldots \lambda t_n.(\lambda(id_1: T_C[U_1]) \ldots ; \lambda(id_k: T_C[U_k])).$

$T_C[C, E \vdash S : \text{Command}]; T_C[C, E \vdash E : T]

[T_C[T_1]] \ldots [T_C[T_n]](T_C[C, E \vdash E_1; U_1]) \ldots (T_C[C, E \vdash E_k; U_k])$

where Block = type $t_1 = T_1; \ldots ; t_n = T_n$

const $id_1; U_1 := E_1; \ldots ; id_k; U_k := E_k$

{ $S; \text{return } E$ }

Using the rule $TypeLet$ in Section 9.1, it can be shown that handling the type definitions by the use of type parameters is equivalent to adding expressions of the form $t_i = T_C[T_i]$ to $C$.

Translation of message sends depends on the type of the receiver. Suppose first that the receiver has a type of the form $\text{VisObjectType}(IV, M)$ (e.g., it is self). Recall from the previous section that the translation of this type is of the form $IV \times (IV \rightarrow M)$. To send a message to an object with this type, simply apply the second component (the method suite) of the object to the first component (the record of instance variables), and then extract the appropriate method.

Suppose, on the other hand, that the receiver has a type of the form $\text{ObjectType} M$. The translation of this type is of the form $\exists Y. Y \times (Y \rightarrow M)$. Therefore, to send a message to an object of this type, the object must first be opened before performing the same operations as the earlier case (i.e., applying the second component to the first component and then extracting the appropriate method).

Extraction of instance variables is obtained simply by extracting the appropriate field of the first component of the object (recall that instance variables are only available in values with type of the form $\text{VisObjectType}(IVR, M)$).

Because the translation of expressions is by induction on the type derivations, we must also include a translation corresponding to the application of the subsumption rule to promote the type of an expression from a subtype to a supertype. This is completely trivial as it simply gives the expression with supertype exactly the same type as the same expression with the subtype.

The translation of statements of $SCOL$ is given in Figure 11.3. The translation of if-then-else, assignment, and sequencing expressions are trivial. Translations of programs are obtained as the translations of their blocks. The

1. We could have used slightly different syntax for these two types of definitions in order to avoid having the translation depend on the type of the receiver. For example, it would be easy to have the type-checker generate different notations for the internal representations of expressions. However, we decided the distinction was not worth the effort here.
11.3 Modeling SOOCL expressions in $\Lambda^p$.

\[ T_c[C,E \vdash if \ E \ then \ \{ S_1 \} \ else \ \{ S_2 \} : Command] \triangleq \]
\[ if T_c[C,E \vdash E : Boolean] then \{ T_c[C,E \vdash S_1 : Command] \} \]
\[ else \{ T_c[C,E \vdash S_2 : Command] \} \]

\[ T_c[C,E \vdash E : T] \triangleq T_c[C,E \vdash \{ S \} : E < : T] \]
where $C \vdash S < : T$.

\[ T_c[C,E \vdash \text{id} := E : Command] \triangleq \text{id} = T_c[C,E \vdash E : T] \]
where $C, E \vdash \text{id} : \text{Ref} T$.

\[ T_c[C,E \vdash \text{while} \ E \ do \ \{ S \} : Command] \triangleq \text{fix[Command]}[\lambda(w : \text{Command}). \]
\[ if T_c[C,E \vdash E : Command] then \{ T_c[C,E \vdash S : Command]; w \} \]
\[ else \{ \text{nop} \} \]

\[ T_c[C,E \vdash S_1 ; S_2 : Command] \triangleq T_c[C,E \vdash S_1 : Command] ; \]
\[ T_c[C,E \vdash S_2 : Command] \]

\[ T_c[C,E \vdash \text{Program id}; \ Block : : \ Command] \triangleq \]
\[ T_c[C,E \vdash \text{Block : : \ Command}] \]

Figure 11.3 Translation of more expressions and statements of SOOCL to expressions in $\Lambda^p$.

use of subtyping in deriving the type of an expression has no effect on its translation.

While statements are interpreted recursively. Using $w$ as an abbreviation for $\text{while} \ E \ do \ \{ S \}$, it is easy to see that $w$ is equivalent to $\text{if} \ E \ then \ \{ S ; w \} \ else \ \{ \text{nop} \}$. We saw earlier that recursive definitions can be interpreted as fixed points. Thus the meaning of the while loop in Figure 11.3 is defined as the fixed point of the function taking parameter $w$ and returning the meaning defined in terms of $\text{if-then-else}$. Because the fixed point will be a statement (and hence an expression of type $\text{Command}$), we apply $\text{fix[Command]}$ to interpret the recursion.

At this point we could show that the translations of types and expressions given so far preserve typing. However, we will postpone the proof of this until after we have completed the translations of classes and subclasses.

However to provide some assurance that our translations so far are reasonable, we look at an example using class $\text{CellClass}$ example. Examination of the translations shows that if expression $o$ has type $\text{ObjectType CellM}$, then the translation $T_c[C,E \vdash o \leftarrow \text{set}(? : \text{Command})]$ will have type $\text{Command}$.
mand and $\mathcal{T}_c [c, e \vdash o \leftarrow \text{get}(): \text{Integer}]$ will have type Integer. These are exactly the translations of the types of the original expressions in SIOOC.

### 11.4 Modeling classes — first try

While we now know how to represent objects, we have not yet talked about how to represent classes or how classes can be used to create objects. An important part of modeling classes is to figure out how to use the initial values of instance variables and methods provided in the class in order to create the “visible” objects, which can then be packed up to hide the instance variables. Using the initial values of instance variables is pretty straightforward, but getting the methods in proper form will be a bit more difficult. In this section we present a first approximation to translating classes. Later we will see that this translation will need to be refined in order to handle subclasses properly.

The bodies of methods in classes may refer to the keyword self, which represents the receiver of the message. This keyword may be understood as an extra parameter of these messages. We expect that when we instantiate new objects using the class, self will be resolved as a fixed point representing the object being defined. As we have seen in our CellClass example, self provides access both to the instance variables and methods of the object. Thus we type check method bodies assuming that the type of self is \( \text{VisObjectType}(\text{IV}_{\text{ref}}, \text{M}) \).

We translate the class as a pair:

$$
\mathcal{T}_c [c, e \vdash \text{class}(\text{inst, meth}): \text{ClassType}(\text{IV}, \text{M})] \triangleq \langle \mathcal{T}_c [c, e \vdash \text{inst}: \text{IV}], \text{methfun} \rangle,
$$

---

2 Later on, we will see that a somewhat more general type for self is better, but this will be fine for now.
where

\[ \text{methfun} = \lambda (\text{self}: \text{VisObj}(IVR, M)) . \]

\[ \text{let close: VisObj}(IVR, M) \rightarrow (\exists Y. \text{VisObj}(Y, M)) \]

\[ = \text{closeobj}[IVR][M] \]

\[ \text{in } \mathcal{T}_C[\mathcal{C}', \mathcal{E}' \vdash \text{meth: } M], \]

\[ \mathcal{C}' = \mathcal{C} \cup \{ \text{SelfType} = \text{VisObject_Type}(IV^\text{ref}, M) \} \]

\[ \mathcal{E}' = \mathcal{E} \cup \{ \text{self: SelfType, close: SelfType } \rightarrow \text{ObjectType } M \} \]

\[ IVR = \mathcal{T}_C[IV^\text{ref}], \text{ and} \]

\[ M = \mathcal{T}_C[\mathcal{M}]. \]

In this translation, \textit{self} is translated as a parameter to the method suite, while \textit{close} is translated as the function that takes elements of type \( \mathcal{T}_C[\text{VisObject_Type}(IV^\text{ref}, M)] = \text{VisObj}(IVR, M) \) and transforms them to elements of type \( \mathcal{T}_C[\text{ObjectType } M] = \exists Y. \text{VisObj}(Y, M) \) by packing them up as existentials.

The translation of the type of a class is given by:

\[ \mathcal{T}_C[\text{ClassType}(IV, M)] = \]

\[ \mathcal{T}_C[IV] \times (\mathcal{T}_C[\text{VisObject_Type}(IV^\text{ref}, M)] \rightarrow \mathcal{T}_C[\mathcal{M}]). \]

The initial values of the instance variables have type \( \mathcal{T}_C[IV] \), while the translation of the visual object type involves type \( \mathcal{T}_C[IV^\text{ref}] \) because it includes variables rather than just values.

In order to translate a \textit{new} expression, which creates an object from the class, it appears that all we need to do is to create a record of references to the initial values of instance variables given in the class, and create a pair of this record of instance variables and of the method suite from the class. However, we have one problem. The method suite for objects takes only the record of instance variables (of type \( \mathcal{T}_C[IV^\text{ref}] \)) as a parameter, while the method suite for classes takes an entire object of type \( \mathcal{T}_C[\text{VisObject_Type}(IV^\text{ref}, M)] \) as a parameter. That is, the method suite for objects has type \( \mathcal{T}_C[IV^\text{ref}] \rightarrow \mathcal{T}_C[\mathcal{M}] \), while the method suite for classes has type \( \mathcal{T}_C[\text{VisObject_Type}(IV^\text{ref}, M)] \rightarrow \mathcal{T}_C[\mathcal{M}] \).

This difference is important because subclasses might override the definition of a method. As a result, we would not like to have the meanings of

\[ \text{3. Actually, this is only for “visible objects”, those with type } \text{VisObject_Type}(IV^\text{ref}, M), \text{ as the type of instance variables for object types has been hidden with an existential quantifier. However, we will need to go through the stage of representing the object as a “visible object”, before “packing” it up with the existential.} \]
methods frozen to be their definitions in the class. We need to leave open
the possibility that the meaning of \texttt{self ⇐ m} will be different in a subclass
than in the class. Because methods are parameterized by \texttt{self} in classes, this
happens naturally when \texttt{self} changes. On the other hand, methods cannot
be overridden in objects, so it makes sense to freeze the meaning of methods
when creating objects. Thus the methods of objects take only the record of
instance variables as a parameter rather than all of \texttt{self}.

We must resolve this (deliberate) inconsistency in representations by con-
verting methods in classes to a different form in objects so that the meanings
of the methods are fixed. (Recall we could have also fixed the location of
instance variables in methods of the objects, but we chose to share methods
between all objects of the same class. As a result instance variables must be
supplied to methods when sending messages.)

We will start with \texttt{methfun}, the record of methods parameterized by \texttt{self},
and then transform this into a function \texttt{methfun'} that is parameterized by only
the record, \texttt{inst}, of instance variables. We will accomplish this by replacing
all occurrences of \texttt{self} in the method bodies by the pair \texttt{inst:methfun}'.
Because we will be defining \texttt{methfun'} in terms of itself, \texttt{methfun'} will be
defined recursively using fixed points.

Let \texttt{C} be an expression with type \texttt{ClassType(IV,M)}. Let

\[
\texttt{methfun} = \texttt{proj}_{\texttt{ClassType(IV,M)}}(\texttt{proj}_{\texttt{C,E}}(\texttt{C} + \texttt{C}))
\]

be the translation of the method suite from \texttt{C}. By the translation of class types,
the type of \texttt{methfun} will be \texttt{ClassType(IV,M)} \rightarrow \texttt{M}. We now
create \texttt{methfun'} with the desired type \texttt{ClassType(IV,M)} \rightarrow \texttt{M} as follows:

\[
\texttt{methfun'}(\texttt{inst'}: \texttt{ClassType(IV,M)}) = \texttt{methfun}(\texttt{inst'},\texttt{methfun'})
\]
or equivalently,

\[
\texttt{methfun'} = \lambda(\texttt{inst'}: \texttt{ClassType(IV,M)}). \texttt{methfun}(\texttt{inst'},\texttt{methfun'})
\]

Notice that the pair \texttt{inst',methfun'} thus replaces all occurrences of \texttt{self} in
the body of the methods. We can write this definition more formally with a
fixed point to capture the recursive definition, but it is even less readable!
The formal definition is:

\[
\texttt{methfun'} = \texttt{fix}(\texttt{proj}_{\texttt{ClassType(IV,M)}} \rightarrow \texttt{M}) \lambda(\texttt{fm}: \texttt{ClassType(IV,M)} \rightarrow \texttt{M}).
\]

\[
\lambda(\texttt{inst'}: \texttt{ClassType(IV,M)}). \texttt{methfun}(\texttt{inst'},\texttt{fm})
\]
Creating the record of instance variables is straightforward. If

\[
\textit{inst} \triangleq \text{proj}_1(\mathcal{T}_c[\mathcal{C}, \mathcal{E} \vdash \text{C: ClassType}(\text{IV}, \mathcal{M})]) : \| l_1 : T_1 ; \ldots ; l_n : T_n \|
\]

then

\[
\textit{inst}^{\text{ref}} = \| l_1 : \text{Ref} T_1 = \text{ref} \text{inst}.l_1 ; \ldots ; l_n : \text{Ref} T_n = \text{ref} \text{inst}.l_n \|
\]

Now we have all the pieces necessary to create a new object from a class:

\[
\mathcal{T}_c[\mathcal{C}, \mathcal{E} \vdash \text{new : ObjectType} \mathcal{M}] \triangleright \text{closeobj}[\text{IVR}][\mathcal{M}](\text{inst}^{\text{ref}}, \text{methfun}')
\]

where

\[
\begin{align*}
c & \triangleq \mathcal{T}_c[\mathcal{C}, \mathcal{E} \vdash \text{C: ClassType}(\text{IV}, \mathcal{M})] \\
\text{IVR} & \triangleq \mathcal{T}_c[\text{IV}^{\text{ref}}] \\
\mathcal{M} & \triangleq \mathcal{T}_c[\mathcal{M}] \\
\text{inst} & \triangleq \text{proj}_1(c), \\
\text{methfun} & \triangleq \text{proj}_2(c), \\
\text{inst}^{\text{ref}} & \text{is formed by adding refs to all fields of inst as above, and} \\
\text{methfun'} & \text{fix}[\text{IVR} \rightarrow \mathcal{M}] \lambda fm : \text{IVR} \rightarrow \mathcal{M}.
\end{align*}
\]

\[
\lambda (\text{inst}' : \text{IVR}) . \text{methfun}(\text{inst}', fm).
\]

To make this construction more concrete we provide the translation for \text{CellClass}. From the definition, we see that

\[
\text{inst} = \| \ x : \text{Integer} = 0 \ | \ : \text{CellIV}
\]

and

\[
\begin{align*}
\text{meth} & \triangleq \| \ \text{get} : \text{Void} \rightarrow \text{Integer} = \\
& \quad \text{function}(v : \text{Void}) : \text{Integer} \text{ is} \\
& \quad \quad \text{return} \ \text{val} \ \text{self}.x \\
\text{set} : \text{Integer} \rightarrow \text{Command} = \\
& \quad \text{function}(\text{nuVal} : \text{Integer}) : \text{Command} \text{ is} \\
& \quad \quad \text{self}.x \ := \ \text{nuVal}; \\
& \quad \quad \text{return} \ \text{nop} \\
\text{bump} : \text{Void} \rightarrow \text{Command} = \\
& \quad \text{function}(v : \text{Void}) : \text{Command} \text{ is} \\
& \quad \quad \text{self} \leftarrow \text{set}(\text{self} \leftarrow \text{get}()+1); \\
& \quad \quad \text{return} \ \text{nop}
\end{align*}
\]

\[
\| : \text{CellM}
\]
The class \texttt{CellClass} will be represented by \texttt{class \{inst, meth\}} for \texttt{inst} and \texttt{meth} defined above.

Recall from Section 11.2 the definitions of \texttt{CellIV}, \texttt{RfCellIV}, and \texttt{CellM}. The translations of the instance variables, \texttt{inst} $\xrightarrow{T_c} \mathcal{C}, \mathcal{E} \vdash \text{inst: CellIV}$, and the method suite, \texttt{methfun} $\xrightarrow{T_c} \lambda(self: \text{VisObj}(IVR, M)). T_c[\mathcal{C}, \mathcal{E} \vdash \text{meth: CellM}]$, are straightforward:

\begin{align*}
\texttt{inst} &= \{ | x: \text{Integer} = 0 | \} : \text{CellIV}, \\
\texttt{methfun} &= \lambda(self: \text{VisObj}(IVR, M)).
\begin{align*}
&\{ | \texttt{get} = \lambda(v: \text{Void}. \text{val}\_self.x); \\
&\quad \texttt{set} = \lambda(\text{nuVal: Integer}. \texttt{self.x} = \texttt{nuVal}; \\
&\quad \texttt{bump} = \lambda(v: \text{Void}. \texttt{self} \leftarrow \texttt{set}(\texttt{self} \leftarrow \texttt{get}() + 1) \\
&\quad | : \text{CellM}.}
\end{align*}
\end{align*}

To improve readability, we wrote expressions of the form \texttt{self} $\leftarrow m$ in the definition of \texttt{bump} as abbreviations for \texttt{proj}_2((\texttt{proj}_1(\texttt{self})).m. Technically the body of \texttt{bump} should be written as

\begin{align*}
\texttt{proj}_2((\texttt{proj}_1(\texttt{self})).\texttt{set}(\texttt{proj}_2((\texttt{proj}_1(\texttt{self}))).\texttt{get}() + 1),
\end{align*}

but that is virtually unreadable! As a result, we will continue to use the abbreviation with “$\leftarrow$” to make the expressions more readable.

Using these expressions, we can construct the interpretation of \texttt{CellClass}:

\begin{align*}
T_c[\emptyset, \emptyset] \vdash \text{CellClass: ClassType}(\text{CellIV}, \text{CellM}) = (\text{inst, methfun})
\end{align*}

Let’s now examine what happens when we create a new object from class \texttt{CellClass}. We begin by creating a record of locations holding the initial values obtained from the class.

\begin{align*}
\text{inst}^{\text{ref}} &\xrightarrow{T_c} \{ | x: \text{Ref Integer} = \text{ref} \_\emptyset | \} : \text{RfCellIV}.
\end{align*}

Next we rewrite the method suite, \texttt{methfun}, by fixing the method suite of \texttt{self} to be exactly the methods in \texttt{methfun}. This is accomplished by replacing all occurrences of \texttt{self} by \texttt{(ir, methfun')}, where \texttt{methfun'} is the new method suite being defined, and \texttt{ir} is a parameter of \texttt{methfun'} representing the record
of instance variables of the associated object.

\[ \text{methfun}' \triangleq \lambda (ir: \text{RfCellIV}). \]
\[ \begin{align*}
&\quad \mid \text{get} = \lambda (v: \text{Void}). \text{val proj}.(\langle ir, \text{methfun}' \rangle).x; \\
&\quad \text{set} = \lambda (\text{nuVal: Integer}). \text{proj}.(\langle ir, \text{methfun}' \rangle).x = \text{nuVal}; \\
&\quad \text{bump} = \lambda (v: \text{Void}). \langle ir, \text{methfun}' \rangle \Leftarrow \text{set}(\langle ir, \text{methfun}' \rangle \Leftarrow \text{get}) + 1 \\
\end{align*} \]

where we have used “\( \Leftarrow \)” as an abbreviation as above. Then \( \text{methfun}' \) has type \( \text{RfCellIV} \to \text{CellM} \).

The new object generated by class \texttt{CellClass} can be obtained by evaluating

\[ \text{closeobj} [\text{RfCellIV}] [\text{CellM}] [\text{inst}_{\text{df}}, \text{methfun}']. \]

To get a better sense of the meaning of the method suite, \( \text{methfun}' \), of this new object, we simplify the method bodies a bit further. Using the definitions and simplifying, we get

\[ \text{methfun}' = \lambda (ir: \text{RfCellIV}). \]
\[ \begin{align*}
&\quad \mid \text{get} = \lambda (v: \text{Void}). \text{val ir}.x; \\
&\quad \text{set} = \lambda (\text{nuVal: Integer}). \text{ir}.x = \text{nuVal}; \\
&\quad \text{bump} = \lambda (v: \text{Void}). \text{methfun}'(\text{ir}).\text{set}(\text{methfun}'(\text{ir}).\text{get}) + 1 \\
\end{align*} \]

Of course, we could go further in simplifying the code for \( \text{bump} \), replacing expressions of the form \( \text{methfun}'(\text{ir}).m \) by their definitions. If we do this then the body of \( \text{bump} \) becomes:

\[ \text{bump} = \lambda (v: \text{Void}). \text{ir}.x = \text{val ir}.x + 1 \]

just as expected.

**Exercise 11.4.1** Translate and then evaluate \( \circ \Leftarrow \text{bump}() \), where \( \circ \) is an object of type \texttt{ObjectType CellM} that was created through a call of \texttt{new CellClass} and whose \texttt{x} instance variable currently has value 47. Recall that you must open the existential before sending the message (and that opening a packed element gives you access to the details of its innards - see the computation rule at the end of Section 9.3).

The translations discussed in this section are collected in Figure 11.4.

In the next section we investigate problems with using this representation for defining subclasses.
\[ T_C[C, E] \vdash \text{class}(\text{inst}, \text{meth}); \text{ClassType}(IV, M) \overset{\Delta}{=} \\
\langle T_C[C, E] \vdash \text{inst}; IV], \text{methfun}, \rangle \\
\text{where} \\
\text{methfun} = \lambda (\text{self}; \text{VisObj}(IVR, M)). \\
\text{let close: VisObj}(IVR, M) \rightarrow (\exists Y. \text{VisObj}(Y, M)) \\
= \text{closeobj}[IVR][M] \\
in T_C[C', E'] \vdash \text{meth}; M, \\
C' = C \cup \{\text{SelfType} = \text{VisObject}(IV'^{\text{ref}}, M)\} \\
E' = \emptyset \cup \{\text{self: SelfType, close: SelfType} \rightarrow \text{ObjectType} M\} \\
IVR = T_C[IV'^{\text{ref}}], \text{and} \\
M = T_C[M]. \\
T_C[C, E] \vdash \text{new} C; \text{ObjectType} M \overset{\Delta}{=} \text{closeobj}[IR][M](\text{inst}^{\text{ref}}, \text{methfun}'). \\
\text{where} \\
c \overset{\Delta}{=} T_C[C, E] \vdash C; \text{ClassType}(IV, M)] \\
IR = T_C[IV'^{\text{ref}}] \\
M = T_C[M] \\
\text{inst} = \text{proj}(_{1}(c)), \\
\text{methfun} = \text{proj}(_{2}(c)), \\
\text{inst}^{\text{ref}} \text{is formed by adding refs to all fields of inst, and} \\
\text{methfun}' = \text{fix}[IR \rightarrow M](\lambda (f): IR \rightarrow M).\lambda (\text{inst}'; IR).\text{methfun}(\langle \text{inst}'', \text{fm} \rangle). \\
\]

Figure 11.4 Translation of class and new expressions of \text{SOOL} to expressions in \Lambda_C^P.

11.5 Problems with modeling subclasses

Now that we have presented the encoding of classes, we investigate how to encode subclasses. Subclasses can add new instance variables and methods as well as override methods from the superclass. Recall from Chapter 6 that in defining subclasses, no changes are allowed to the types of instance variables from the superclass, while types of methods of the superclass may be replaced by subtypes in the subclass definition.

We begin by using the simple encoding of the last section and then seeing where we run into trouble. Recall that

\[
\text{class} \text{ inherits } C \text{ modifies } m_{j_1}, \ldots, m_{j_u}(\text{inst}, \text{meth})
\]
represents a subclass formed by inheriting the instance variables and methods from $C$ and then adding the instance variables and methods from \( \text{inst} \) and \( \text{meth} \). The methods overridden are named in the list \( m_{j_1}, \ldots, m_{j_s} \).

If a method in $C$ is redefined in $\text{meth}$, then the type-checking rules require that the redefined method have a type that is a subtype of the original. As an example, suppose $C = \text{class} (\text{inst}_{\text{sup}}, \text{meth}_{\text{sup}})$, and thus has encoding

\[
\begin{align*}
T_C [C, E] &\vdash C : \text{ClassType} (\text{IV}_{\text{sup}}, \text{M}_{\text{sup}})] = \\
&\langle T_C [C, E] \vdash \text{inst}_{\text{sup}} : \text{IV}_{\text{sup}}, \text{methfun}_{\text{sup}} \rangle,
\end{align*}
\]

where

\[
\text{methfun}_{\text{sup}} = \lambda (\text{self} : \text{VisObj} (\text{IV}_{\text{sup}}, \text{M}_{\text{sup}})).
\]

\[
\begin{align*}
&\text{let close} : \text{VisObj} (\text{IV}_{\text{sup}}, \text{M}_{\text{sup}}) \rightarrow (\exists Y. \text{VisObj} (Y, M_{\text{sup}})) \\
&\quad = \text{closeobj} [\text{IV}_{\text{sup}}] [M_{\text{sup}}] \\
&\quad \text{in } T_C [C', E' \vdash \text{meth}_{\text{sup}} : \text{M}_{\text{sup}}],
\end{align*}
\]

\[
C' = C \cup \{ \text{SelfType} = \text{VisObjType} (\text{IV}_{\text{sup}}, \text{M}_{\text{sup}}) \}
\]

\[
E' = E \cup \{ \text{self} : \text{SelfType},
\]

\[
\text{close} : \text{SelfType} \rightarrow \text{ObjectType} \text{M}_{\text{sup}}
\]

\[
\text{IV}_{\text{sup}} = T_C [\text{IV}_{\text{sup}}], \text{and}
\]

\[
M_{\text{sup}} = T_C [M_{\text{sup}}].
\]

Let $\text{inst}_{\text{all}} = \text{inst}_{\text{sup}} \oplus \text{inst}$ have type IV, where $\oplus$ is right dominant record combination, as usual. Similarly, let $\text{meth}_{\text{all}} = \text{meth}_{\text{sup}} \oplus \text{meth}$ have type M. Mimicking the encoding of classes from the previous section, one would anticipate that we could encode the subclass as follows:

\[
\begin{align*}
T_C [\text{class inherits } C \text{ modifies } m_{j_1}, \ldots, m_{j_s} (\text{inst}, \text{meth})] &\vdash \\
&\langle T_C [C', E'] \vdash \text{inst}_{\text{all}} : \text{IV}, \text{methfun}_{\text{all}} \rangle,
\end{align*}
\]

where

\[
\text{methfun}_{\text{all}} = \lambda (\text{self} : \text{VisObj} (\text{IV}_{\text{sup}}, M_{\text{sup}})).
\]

\[
\begin{align*}
&\text{let close} : \text{VisObj} (\text{IV}_{\text{sup}}, M_{\text{sup}}) \rightarrow (\exists Y. \text{VisObj} (Y, M_{\text{sup}})) \\
&\quad = \text{closeobj} [\text{IV}_{\text{sup}}] [M_{\text{sup}}] \\
&\quad \text{in } T_C [C', E' \vdash \text{meth}_{\text{all}} : M_{\text{sup}}],
\end{align*}
\]

\[
C' = C \cup \{ \text{SelfType} = \text{VisObjType} (\text{IV}_{\text{sup}}, M_{\text{sup}}) \}
\]

\[
E' = E \cup \{ \text{self} : \text{SelfType},
\]

\[
\text{close} : \text{SelfType} \rightarrow \text{ObjectType} M_{\text{sup}}
\]

\[
\text{IV}_{\text{ref}} = T_C [\text{IV}_{\text{ref}}], \text{and}
\]

\[
M_{\text{ref}} = T_C [M_{\text{ref}}].
\]

4. We leave off the typing information in expressions to be translated for the rest of this chapter to reduce clutter.
Unfortunately, this simple encoding will not work for subclasses because of typing problems with inherited methods. The problem is that when we type-checked the methods of `meth`, we assumed that the type of `self` was `VisObjectType(IV_{sup}^{ref}, M_{sup})`, whereas now we are using the inherited methods in a context where the type of `self` is `VisObjectType(IV_{ref}^{ref}, M)`. This would be fine if `self` having type `VisObjectType(IV_{ref}^{ref}, M)` implied that it had type `VisObjectType(IV_{sup}^{ref}, M_{sup})`, i.e., if

\[ \mathcal{T}_c[VisObjectType(IV_{ref}^{ref}, M)] <: \mathcal{T}_c[VisObjectType(IV_{sup}^{ref}, M_{sup})], \]

but that is NOT the case.

Recall that

\[ \mathcal{T}_c[VisGreaterThanType(IV_{ref}^{ref}, M)] = \mathcal{T}_c[IV_{ref}^{ref}] \times (\mathcal{T}_c[IV_{ref}^{ref}] \rightarrow \mathcal{T}_c[M]) \]

Thus for

\[ \mathcal{T}_c[VisGreaterThanType(IV_{ref}^{ref}, M)] <: \mathcal{T}_c[VisGreaterThanType(IV_{sup}^{ref}, M_{sup})], \]

we will need

\[ \mathcal{T}_c[IV_{ref}^{ref}] <: \mathcal{T}_c[IV_{sup}^{ref}], \]

and

\[ \mathcal{T}_c[IV_{ref}^{ref}] \rightarrow \mathcal{T}_c[M] <: \mathcal{T}_c[IV_{sup}^{ref}] \rightarrow \mathcal{T}_c[M_{sup}]. \]

The first is no problem since `IV_{ref}^{ref}` is simply an extension of `IV_{sup}^{ref}`, and hence `\mathcal{T}_c[IV_{ref}^{ref}] ::: \mathcal{T}_c[IV_{sup}^{ref}]`. However, recall that subtyping for function types is covariant in the return type and contravariant in the domain type. The return types are in the subtype relation, as `M` is an extension of `M_{sup}`, but the domains change in a covariant way, rather than contravariant. That is, we need `\mathcal{T}_c[IV_{sup}^{ref}] ::: \mathcal{T}_c[IV_{ref}^{ref}]` for the function types to be subtypes, but we have the reverse. The result is that the methods in `meth` may not be type safe when used in a context in which `self` has type `VisObjectType(IV_{ref}^{ref}, M)`. Let’s look at an example showing what can go wrong when you use a parameter, `self`, of type `VisObjectType(IV_{ref}^{ref}, M)` in a context that is type-correct when `self` has type `VisObjectType(IV_{sup}^{ref}, M_{sup})`.

Let `oself` be a record of type `IV_{sup}^{ref} = \{ j : Ref Integer \}`, and define class `C` to be:
11.5 Problems with modeling subclasses

class (|
    j: Integer = 0
|), |
    m: Void → Integer = function(v: Void): Integer is
        { return ⟨oself, snd(oself)⟩ ≐ n() ⟩;
    n: Void → Integer = function(v: Void): Integer is
        { return val self.j ⟩
|)

Of course you normally can’t treat self as a pair in object-oriented languages or use a projection function, snd to grab the second component of a pair, but we will be loose with the syntax in order to express the problem in a simple way.

Let oself ≡ \( T_C \langle oself \rangle \). Then the following is the translation of \( C \):

\[
\begin{align*}
\langle | ] : \text{Integer} = 0 | ]
\lambda \text{self}: T_C [\text{VisObjectType}(IV_{\text{sup}_{M_{\text{sup}}}})].
\langle | m : \text{Void} \rightarrow \text{Integer} = \lambda (v : \text{Void}). \langle oself, proj_2 (self) \rangle \approx n() ⟩;
\langle | n : \text{Void} \rightarrow \text{Integer} = \lambda (v : \text{Void}). \langle \text{val self.j} \rangle
\end{align*}
\]

The method \( m \) of \( T_C[C] \) builds a new object, \( \langle oself, proj_2 (self) \rangle \), with type \( T_C [\text{VisObjectType}(IV_{\text{sup}_{M_{\text{sup}}}})] \), by replacing the first component of \( self \) by \( oself \), and then sends it the message \( n \). In other words, when method \( n \) is executed, it is with \( self \) interpreted by this new object whose instance variables are now represented by \( oself \). All the method \( n \) of \( C \) does is to return the value of the instance variable \( j \). This is no problem at all when \( n \) is called from \( m \), as \( oself \) has a field \( j \) of type \( \text{Ref Integer} \).

Now define subclass \( SC \) of \( C \) as follows:

class inherits C modifies n (|
    k: Integer = 0
|), |
    n: Void → Integer = function(v: Void): Integer is
        { return val self.k ⟩
|)

5. We omit the let clause introducing close because it does not occur in the translation of the methods. We will also omit it in the translation of \( SC \).
The translation of $SC$ is

$$
\langle \langle j : \text{Integer} = 0; k : \text{Integer} = 0 \rangle, \\
\lambda (self : T_C[[\text{VisObjectType}(IV_{vis}^{ref}, M_{sup})]]).
\langle \langle m : \text{Void} \rightarrow \text{Integer} = \lambda (v : \text{Void}). \{ (\text{oself}, \text{proj}(self)) \leftarrow n(\rangle) \}; \\
n : \text{Void} \rightarrow \text{Integer} = \lambda (v : \text{Void}). \{ \text{val self.k} \\
\rangle
\rangle
$$

But now a problem arises when method $m$ is sent to an object generated from class $SC$. Since $m$ is not overridden, the inherited code is run, resulting in sending a message $n$ to an object whose method suite is the same (i.e., the full method suite of $self$), but whose instance variables have been replaced by $oself$ (which has no $k$ field). Thus when the evaluation of method $n$ results in an attempt to access $self.k$, an error will result.

**Exercise 11.5.1** Write out the translation of classes $C$ and $SC$ in detail, evaluate the expression $\text{new SC}$ to get an object, and then send that object the message $m$ in order to show that a type error results.

What does this imply about the type-safety of languages like Java and C++? Luckily, not very much, as no object-oriented language would actually allow one to write a method like $m$ above. That is fortunate, for if it were legal, virtually all existing statically typed languages would allow type errors. The stumbling block to writing such a method is that object-oriented languages do not allow you to take apart $self$ and then build a new object with a different collection of instance variables.

In fact we can only generate type errors with our encoding if we allow the methods to refer to the record of instance variables as a whole. If we only are allowed to access individual instance variables and methods of $self$, no errors result. In particular, the typing rules for classes and subclasses in $SOOL$ given in Section 10.2.3 are correct in that any $SOOL$ program that passes the type checker is guaranteed not to have type errors at run time. The problem that has arisen here is that our encoding of classes and subclasses in $\Lambda_C$ is not type-safe, because extra operations are available in $\Lambda_C$ that do not correspond to operations in $SOOL$.

As a result, we could try to create a restricted version of the typed lambda calculus used here that disallows the updates that caused the problems, and then try to prove the correctness of our translation. However, it is quite difficult to restrict manipulations of $self$ in the translation without also restricting other useful constructs in our underlying lambda calculus. Thus, rather than
restricting the underlying calculus, in the next chapter we will instead modify the type-checking rules and translation of classes so that it will be type safe no matter what manipulations of self are made in the method bodies.

11.6 Summary

In this chapter we presented a translation of SOOC into $\Lambda^p_\prec$. The translation of non-object-oriented features was straightforward, and our translation of classes resulted in reasonable, though somewhat complex, definitions of the translations of new $C$ and $E \leftarrow m$. Unfortunately, we found that the translation of classes does not extend properly to subclasses. As a result, in the next chapter we introduce a more careful type-checking rule and translation of classes that will support the correct definition of subclasses.
Improved Semantics for Classes

We need to rethink the translation of classes from first principles in order to overcome the difficulties encountered in trying to translate subclasses. A problem whose solution will lead us to the correct definition of translation is that we type checked the methods under the assumption that the type of self was the same as the (visible) type of the objects generated by the class. Yet, these methods may be inherited in subclasses that generate objects having more instance variables and methods (let alone the fact that existing methods may be overridden). It’s no wonder that we got into trouble!

12.1 (Re-)Defining classes

If the methods in classes are to be inherited in subclasses without any danger of encountering new errors, then it seems obvious that they should be type checked under the assumption that they may be inherited in any subclass of the class being defined. Of course, at the time we are defining a class, we have no idea what subclasses may later be defined.

In Chapter 6 we investigated what changes are allowed in the types of instance variables and methods in order to preserve type safety in subclasses. These restrictions were reflected in the \(SOOL\) type-checking rule, Subclass. That rule implies that if class \(C\) has type \(\text{ClassType}(IV, M)\) and \(C'\) is a subclass with type \(\text{ClassType}(IV', M')\), then it is necessary to have \(IV' \text{ ref} \subseteq: IV \text{ ref}\) and \(M' \subseteq: M\) to ensure that inherited methods will be type correct in the subclass.

We can use this to write type-checking rules for subclasses in such a way that inherited methods will be guaranteed to be type-safe in subclasses. As before, let \(\text{inst}\) of type \(IV\) be the record of initial values of instance variables for a class, and let \(\text{meth}\) of type \(M\) be the record of method bodies (which
Improved Semantics for Classes

may include the keyword `self`). The new type-checking rules for classes and subclasses can be found in Figure 12.1.

The new type-checking rules use a generalization of the type constraint system $\mathcal{C}$ that supports adding type identifiers with their bounds. While $SOO\mathcal{C}$ does not support bounded polymorphism\(^1\), the type constraint systems are required for type checking. They keep track of the requirements that ensure that the type of `self` can be the type of objects generated in any subclass of the class being type checked. In particular the definition of $\mathcal{C}'$ in the new rule `Class` adds to $\mathcal{C}$ two subtyping constraints that correspond to the restrictions on the types of the records of instance variables and methods in subclasses.

**Definition 12.1.1** Relations of the form $\varepsilon = T$ and $\varepsilon <: T$, where $\varepsilon$ is a type identifier and $T$ is a type expression are said to be type constraints. A type constraint system is defined as follows:

1. The empty set, $\emptyset$, is a type constraint system.
2. If $\mathcal{C}$ is a type constraint system and $\varepsilon$ is a type identifier that does not appear in $\mathcal{C}$ or $T$, then $\mathcal{C} \cup \{\varepsilon = T\}$ is a type constraint system.
3. If $\mathcal{C}$ is a type constraint system and $\varepsilon$ is a type identifier that does not appear in $\mathcal{C}$, then $\mathcal{C} \cup \{\varepsilon <: T\}$ is a type constraint system.

We define the function $\mathcal{C}(T)$ as in Definition 10.2.5 to be the result of replacing all type identifiers by their definitions in $\mathcal{C}$. If $\varepsilon$ is included in $\mathcal{C}$ with a subtyping bound rather than a definition, then $\mathcal{C}(\varepsilon) = \varepsilon$.

With this new definition of type constraint system we must add one new rule to our subtyping rules to cover the case of type identifiers introduced with a subtyping bound:

$$\text{Identifier} <: \quad \mathcal{C} \vdash \varepsilon <: \mathcal{C}(T), \text{if } \{\varepsilon <: T\} \in \mathcal{C}$$

This rule simply states that if $\mathcal{C}$ contains a type identifier, $\varepsilon$, with a bound, then we may deduce that $\varepsilon$ has that bound (where we expand all abbreviations introduced in $\mathcal{C}$).

We now examine the new type-checking rules for classes and subclasses. As before, when type checking the methods of a class, `self` will have type

\(^1\) Later, we will introduce an extension of $SOO\mathcal{C}, P \mathcal{C}$, that will support bounded polymorphism.
12.1 (Re-)Defining classes

Class

\[ \begin{align*}
\mathcal{C}, \mathcal{E} & \vdash \text{inst: IV, } \mathcal{C}', \mathcal{E}' \vdash \text{meth: M} \\
\mathcal{C}, \mathcal{E} & \vdash \text{class (inst, meth): ClassType(IV, M)}
\end{align*} \]

where

- \( \mathcal{C}' = \mathcal{C} \cup \{ M' <: M, \text{IVR'} <: \text{IV}^{ref} \}, \)  
  \( \text{SelfType} = \text{VisObjectType(IVR', M')} \),

- \( \mathcal{E}' = \mathcal{E} \cup \{ \text{self: SelfType, close: SelfType } \rightarrow \text{ObjectType } M' \} \),  
  and

- \( \text{SelfType does not occur in IV or M} \).

\[ \begin{align*}
\mathcal{C}, \mathcal{E} & \vdash \mathcal{E}: \text{ClassType(IV}_{\text{sup}}, M_{\text{sup}}) \\
\mathcal{C}, \mathcal{E} & \vdash \text{inst: IV}_{\text{sub}}, \quad \mathcal{C}', \mathcal{E}' \vdash \text{meth: M}_{\text{sub}}, \\
\mathcal{C} & \vdash T_{i_1} <: T_i \text{ for } 1 \leq j \leq m
\end{align*} \]

Subclass

\[ \begin{align*}
\mathcal{C}, \mathcal{E} & \vdash \text{class inherits } E \text{ modifies } l_{i_1}, \ldots, l_{i_m} \\
& \quad (\text{inst, meth): ClassType(IV, M)}
\end{align*} \]

where

- \( IV = IV_{sup} \oplus IV_{sub} \),

- \( M = M_{sup} \oplus M_{sub} \),

- there is no overlap in the labels occurring in \( IV_{sup} \) and \( IV_{sub} \), and

- the overlapping labels in \( M_{sup} \) and \( M_{sub} \) are exactly \( l_{i_1}, \ldots, l_{i_m} \).

- The type of \( l_{i_j} \) in \( M_{sub} \) is \( T'_{i_j} \), while the corresponding type in \( M_{sup} \) is \( T_{i_j} \).

- \( \mathcal{C}' = \mathcal{C} \cup \{ M' <: M, \text{IVR'} <: \text{IV}^{ref} \}, \)  
  \( \text{SelfType} = \text{VisObjectType(IVR', M')} \), and

- \( \mathcal{E}' = \mathcal{E} \cup \{ \text{self: SelfType, close: SelfType } \rightarrow \text{ObjectType } M' \} \)

- \( \text{SelfType does not occur in IV or M} \).

Figure 12.1 Revised type-checking rules for classes and subclasses.
SelfType. However, we may not assume that \( \text{SelfType} = \text{VisObject-Type}(\mathit{IVref}, M) \). Instead we may only assume that it has type \( \text{VisObject-Type}(\mathit{IVR}', M') \), where all we know about \( \mathit{IVR}' \) and \( \mathit{M}' \) is that they are subtypes of the corresponding types \( \mathit{IVref} \) and \( M \) from the class. Notice that this does allow us to infer that \text{self} has all the instance variables and methods of the class, but it may have more of each, and the types of the methods may be subtypes of those in \( M \).

These are exactly the conditions that hold for subclasses. Thus if we type check the methods of classes under these assumptions, then we are guaranteed that they will type check for the types corresponding to any subclass. In particular, if the initial values of the instance variables and the methods were found to be type correct in the superclass, they should remain correct in any subclass.\(^2\)

The same argument shows that type checking subclasses under the weaker conditions on \text{SelfType} in rule \text{Subclass} will ensure that methods inherited in further subclasses will remain type correct.

These more general type-checking rules for classes and subclasses will require more general rules for message sending and instance variable accesses. The problem is that when we try to type check \text{self} and \text{self} \( \leftarrow m \) inside methods of a class, we only know the type of \text{self} is \( \text{SelfType} = \text{VisObject-Type}(\mathit{IVR}', \mathit{M}') \), and, rather than having specific information about the fields and types of \( \mathit{IVR}' \) and \( \mathit{M}' \), we only know that they are subtypes of \( \mathit{IVref} \) and \( M \). Thus we need type-checking rules that will allow us to type message sends and instance variable accesses when we only know information about the supertype of the records of instance variables and methods.

The new rules for message sending and instance variable access can be found in Figure 12.2. We take advantage of subtyping to simplify the statement of the rules by having the supertype in each case contain only a single field. Of course if

\[
C \vdash M <: \text{ObjectType} \left\{ \| m_1: T_1; \ldots; m_n T_n \| \right\},
\]

then we can also deduce

\[
C \vdash \text{ObjectType} \left\{ \| m_1: T_1; \ldots; m_n T_n \| \right\} <: \text{ObjectType} \left\{ \| m_i: T_i \| \right\}
\]

for whichever method \( m_i \) is of interest, and hence deduce

\[
C \vdash M <: \text{ObjectType} \left\{ \| m_i: T_i \| \right\},
\]

\(^2\) We give a formal proof of this in Section 13.1.
and apply the type-checking rule.

At this point these more general rules are mainly necessary for objects whose types are of the form \( \text{VisObject}(IVR,M) \). However we run into a similar problem in type checking if \( \text{close}(self) \), which has type \( \text{ObjectType} M' \), is the receiver of a message. While we could just use subtyping to promote the type to \( \text{ObjectType} M \), we include the more general type-checking rule for elements of regular object types as well.

We now use the new type-checking rules to guide the definition of the translation of classes. The new translation of classes and their types, provided in Figure 12.3, explicitly takes into consideration that the methods should work for any subclass. It does this by parameterizing the translation of the class by types extending the method and instance variable types.

Just as we type checked instance variables using \( C \), and methods using \( C' \), we translate the instance variables of the class using \( C \) and the methods using \( C' \). The record of initial values of instance variables still has type \( IV \), but the parameter \( self \) to the method suite now has type \( \text{VisObj}(IVR', M') \). When \( IVR' = T_C[IV]\text{ref} \) and \( M' = T_C[M] \), the methods will be typed appropriately for objects generated from a class \( C \). But we can also instantiate the type identifiers with types corresponding to any subclass of \( C \).

Of course, we must modify the translation of the type of a class similarly. The new translation for class types as a polymorphic type is also given in Figure 12.3.

The translation of expressions that create new objects from classes, shown in Figure 12.4, now just takes one more step from the procedure given for our previous encoding. First apply the translation of the class to the translation of the type of its record of methods, \( M \), and to the translation, \( T_C[IV]\text{ref} \), of
\[ T_C [\mathcal{C}, \mathcal{E} ] \vdash \text{class}(\text{inst}, \text{meth}) : \text{ClassType}(\text{IV}, M) \]  
\[ \triangleq \Lambda (M' : T_C [M]). (IVR' \triangleright T_C [IV' ref]).(T_C [\mathcal{C}, \mathcal{E} ] \vdash \text{inst} : \text{IV}), \text{methfun}, \]

where

\[ \text{methfun} = \lambda (\text{self}) : \text{VisObj}(IVR', M'). \]

\[ \text{let close} : \text{VisObj}(IVR', M') \rightarrow (\exists Y. \text{VisObj}(Y, M')) \]

\[ = \text{closeobj}[IVR'][M] \]

\[ \text{in } T_C [\mathcal{C}', \mathcal{E} ] \vdash \text{meth} : M', \]

\[ \mathcal{C}' = \mathcal{C} \cup \{ M' : M, IVR' \triangleright IV' ref, \}

\[ \text{SelfType} = \text{VisObjectType}(IVR', M'), \]

\[ \mathcal{E}' = \mathcal{E} \cup \{ \text{self} : \text{SelfType}, \text{close} : \text{SelfType} \rightarrow \text{ObjectType} M \} \]

\[ T_C [\text{ClassType}(\text{IV}, M)] \triangleq \forall (M' : T_C [M]). \forall (IVR' \triangleright T_C [IV' ref]). \]

\[ T_C [\text{IV}] \times (\text{VisObj}(IVR', M') \rightarrow T_C [M]). \]

**Figure 12.3** New translation of classes and class types.

the type \( IV' ref \) of its record of instance variables, and then proceed exactly as before.

With the new translation for classes, the translation of the problem code from the last chapter that broke subtyping will no longer type check properly. While it might be more compelling to show that the untranslated code doesn’t type check, that is impossible because we have written that code using pairs and projection, neither of which is legal in \( S O OC \). As a result we will be satisfied by showing that the translation no longer type checks.

Here is the troublesome example again:

```plaintext
class ( ||
    j: Integer = 0
  ||, ||
  m: Void \rightarrow Integer = \text{function}(v: Void): Integer is
  { return \langle oself, snd(self) \rangle \leftarrow n() };
  n: Void \rightarrow Integer = \text{function}(v: Void): Integer is
  { return \text{val self}.j }
\)

where oself has type \( IV' ref \) = \{ j: \text{Ref Integer} \}
```

Team-Fly®
12.1 (Re-)Defining classes

\[ T_C[C, E] \vdash \text{new } C : \text{ObjectType} \overset{\Delta}{=} \text{closeobj } [IVR][M](\text{inst}^{ref}, \text{methfun}') \]

where
\[
c \overset{\Delta}{=} T_C[C, E] \vdash C : \text{ClassType}(IV, M)[M][IVR]
\]
\[ IVR = T_C[IV^{ref}] \]
\[ M = T_C[M] \]
\[ \text{inst} = \text{proj}_1(c), \]
\[ \text{methfun} = \text{proj}_2(c), \]
\[ \text{inst}^{ref} \] is formed by adding refs to all fields of inst, and
\[ \text{methfun}' = \text{fix} [IVR \rightarrow M] \lambda (fm : IVR \rightarrow M). \]
\[
\lambda (\text{inst}' : IVR'). \text{methfun}(\langle \text{inst}', fm \rangle).
\]

Figure 12.4 New translation of new expressions.

The translation is
\[
\forall M' <: T_C[M]. \forall IVR' <: T_C[IV^{ref}].
\]
\[
\langle j : Integer = 0 \rangle, \lambda (self : \text{VisObj}(IVR', M')).
\]
\[
\langle m : \text{Void} \rightarrow Integer = \lambda (v : \text{Void}). \langle (\text{oself}, \text{proj}_2(self)) \rangle \leftarrow n(); \rangle;
\]
\[
n : \text{Void} \rightarrow Integer = \lambda (v : \text{Void}). \langle \text{val self.j} \rangle
\]

where \( oself \overset{\Delta}{=} T_C[C, E] \vdash \text{oself} : IV^{ref} \).

If we eliminate the use of the abbreviation \( \leftarrow \) in the body of method \( m \), it becomes:
\[
\lambda (v : \text{Void}). \langle (\text{proj}_2(self)(oself)).n() \rangle
\]

Because we may only assume that the type of self is \( \text{VisObj}(IVR', M') = IVR' \times (IVR' \rightarrow M') \), the application of \( \text{proj}_2(self) \) to \( oself \) is not type correct. The function has type \( IVR' \rightarrow M' \) while the argument has type \( IV^{ref} \). The problem is that we only know \( IVR' <: T_C[IV^{ref}] \). For the application of \( \text{proj}_2(self) \) to \( oself \) to be correct, we need the type of \( oself \) to be a subtype of the domain of the function, i.e., for \( T_C[IV^{ref}] <: IVR' \). Because we only know the reverse, we cannot send the message \( n \) to this pair. The translation of class \( C \) will not type check, and hence is illegal.

The fact that (the translation of) this counterexample has been ruled out should give us some confidence that we are on the right track with our translation. We should also gain confidence because the type-checking rules ex-
explicitly take into consideration the possibility of being inherited into subclasses. The translation for classes and subclasses also follows the type-checking rules closely. In the next chapter we prove that this confidence is indeed warranted.

12.2 A correct subclass encoding

With more confidence in our refined type-checking rules and translation for classes, let us proceed to redefine the translation of subclasses and provide some assurance that the translated expression is type safe. A formal proof that the translation preserves types will be given in Section 13.1.

Suppose the class $C$ has type $\text{ClassType}(IV_{sup}, M_{sup})$, and let $C_{sub}$ be a subclass with type $\text{ClassType}(IV, M)$. As before, we assume that new and old instance variables have no overlap, and if a method in $C$ is redefined in $\text{meth}$, then the redefined method must have a type that is a subtype of the original.

We will use notation similar to that of the type-checking rule for subclasses. Let $IV_{sub}$ be the type of $\text{inst}$ and $M_{sub}$ be the type of $\text{meth}$. Let $IV = IV_{sup} \sqcup IV_{sub}$ and $M = M_{sup} \sqcup M_{sub}$.

The translation of subclasses is given in Figure 12.5. It generalizes the previous translation by adding type parameters $M'$ and $IV'$, just as we did for classes.

The definition of $\text{methfun}_sup$ is the only questionable part of the definition because it is where the old methods are combined with the new. We argue informally that the application of $\text{methfun}_sup$ to $\text{self}$ in the definition of $\text{methfun}_all$ is type correct.

We expect $\mathcal{T}_C[C, \mathcal{E}] : \text{ClassType}(IV_{sup}, M_{sup})$ to have type:

$$\forall (M' : \mathcal{T}_C[IV_{sup}]), \forall (IV' : \mathcal{T}_C[IV_{sup}]).$$

$$\forall ((IVR', M') : (\text{VisObj}(IV', M') \rightarrow \mathcal{T}_C[IV_{sup}])).$$

From this we see that $\text{methfun}_sup$ has type $\text{VisObj}(IV', M') \rightarrow \mathcal{T}_C[IV_{sup}]$ under the assumptions that $IV' : \mathcal{T}_C[IV_{sup}]$ and $M' : \mathcal{T}_C[IV_{sup}]$.

In the subclass translation, these inherited methods are applied to $\text{self}$, which has type $\text{VisObj}(IV', M')$, where $IV' : \mathcal{T}_C[IV_{sup}]$ and $M' : \mathcal{T}_C[M]$. However, because we are defining a subclass of $C$, $\mathcal{T}_C[IV_{sup}] : \mathcal{T}_C[IV_{sup}]$ and $\mathcal{T}_C[M] : \mathcal{T}_C[M]$. By transitivity, it follows that $IV' : \mathcal{T}_C[IV_{sup}]$ and

3. We prove this in Section 13.1.
4. Again, these will follow from the results in Section 13.1.
12.3 Summary and a look ahead

The final translations of all types and expressions of SOOC are summarized in Figures 12.6, 12.7, 12.8, and 12.9. The major difference between the translation of classes and subclasses presented here versus those given in the previous chapter is that the classes are polymorphic, parameterized by the types of methods and instance variables that might arise in subclasses. As a result, methods take a self parameter whose type is not that of objects generated from the class, but instead from some possible subclass (which one depends on the instantiations of the type parameters).

The constraints on the type parameters, IVR' and M', in the subclass type-checking rules and translation are stronger than those in the superclass, so any method that type checks under the original, weaker assumptions, is guaranteed to type check under the stricter assumptions in the subclass. As a result, we have now succeeded in modeling objects, classes, and subclasses
in such a way that inherited methods will remain type correct if they were type correct in the superclass.

In the next chapter, we prove the soundness of our translation by showing that if an expression of SOOL is well-typed, then its translation in $\Lambda_\mathcal{C}$ is also well-typed, with a type which is the translation of the type of the original SOOL expression. The proof is long and quite technical, but provides confidence that our translation makes sense.

Before proceeding to this proof, we pause to look ahead to topics to be discussed in succeeding chapters. First, there are some details that we left out of our typing rules and translations that we have yet to address. These include how to handle calls to superclasses, the typing and translation of nil, and handling of errors. They will be addressed in Chapter 14.

It may be surprising that the translation of classes required moving to a language which included bounded polymorphism. However, this reflects the complexity of subclasses and inheritance. But as a result, we will find in Chapter 15 that adding F-bounded polymorphism to SOOL requires no new technical tools. The added complexity necessary to model classes is sufficient to handle the translation of F-bounded polymorphism.

Also in Chapter 14, we discuss some subtle points which arise in the typing of self. While these are easily handled with the use of close and the Subsumption rule, they end up giving a less accurate typing of self than might be desired. This weakness in the typing system suggests the addition
where
\[\mathcal{T}_C[C, \mathcal{E} \vdash \text{class}(\text{inst}, \text{meth}): \text{ClassType}(IV, M) \overset{\Delta}{=} \lambda(M' \leftarrow \mathcal{T}_C[M]). \lambda(IV' \leftarrow \mathcal{T}_C[IV'^{\sigma}]). \langle \mathcal{T}_C[C, \mathcal{E} \vdash \text{IV}, \text{methfun} \rangle, \]

where

\[\text{methfun} = \lambda(\text{self}: \text{VisObj}(IV', M')).\]

let close: VisObj(IV', M') \rightarrow (\exists Y. \text{VisObj}(Y, M'))

= closeobj[IV'][M']

in \mathcal{T}_C[C', \mathcal{E}' \vdash \text{meth}: M],

\[C' = C \cup \{M' \leftarrow M, IV' <: IV'^{\sigma}, \}
\]

\[\text{SelfType} = \text{VisObjectType}(IV', M'),\]

\[\mathcal{E}' = \mathcal{E} \cup \{\text{self: SelfType, close: SelfType} \rightarrow \text{ObjectType} M\}\]

\[\mathcal{T}_C[C, \mathcal{E} \vdash \text{class inherits} C \text{ modifies} l_1, \ldots, l_m (\text{inst}, \text{meth}): \text{ClassType}(IV, M) \overset{\Delta}{=} \lambda(M' \leftarrow \mathcal{T}_C[M]). \lambda(IV' <: \mathcal{T}_C[IV'^{\sigma}]). \langle \text{inst}_{\text{sup}}, \text{methfun}_{\text{sup}} \rangle, \]

where

\[c = \mathcal{T}_C[C, \mathcal{E} \vdash C: \text{ClassType}(IV_{\text{sup}}, M_{\text{sup}})] [M'][IV']\]

\[\text{inst}_{\text{sup}} = \text{proj}_1(c)\]

\[\text{methfun}_{\text{sup}} = \text{proj}_2(c)\]

\[\text{inst}_{\text{all}} = \text{inst}_{\text{sup}} \oplus \mathcal{T}_C[C, \mathcal{E} \vdash \text{inst: IV}_{\text{sub}}]\]

\[\text{methfun}_{\text{all}} = \lambda(\text{self}: \text{VisObj}(IV', M')).\]

let close: VisObj(IV', M') \rightarrow (\exists Y. \text{VisObj}(Y, M'))

= closeobj[IV'][M']

in \text{methfun}_{\text{sup}}(\text{self}) \oplus \mathcal{T}_C[C', \mathcal{E}' \vdash \text{meth: M}_{\text{sub}}],

\[C' = C \cup \{M' \leftarrow M, IV' <: IV'^{\sigma}, \}
\]

\[\text{SelfType} = \text{VisObjectType}(IV', M'),\]

\[\mathcal{E}' = \mathcal{E} \cup \{\text{self: SelfType, close: SelfType} \rightarrow \text{ObjectType} M\}\]

\[\mathcal{T}_C[C, \mathcal{E} \vdash \text{new} C: \text{ObjectType} M \overset{\Delta}{=} \text{closeobj}[IVR][M]^{\sigma}(\text{inst}^{\sigma}, \text{methfun'})\]

where

\[c \overset{\Delta}{=} \mathcal{T}_C[C, \mathcal{E} \vdash C: \text{ClassType}(IV, M)] [M][IVR]\]

\[IVR = \mathcal{T}_C[IV'^{\sigma}], M = \mathcal{T}_C[M]\]

\[\text{inst} = \text{proj}_1(c), \text{methfun} = \text{proj}_2(c),\]

\[\text{inst}^{\sigma} \text{ is formed by adding refs to all fields of inst, and}\]

\[\text{methfun'} = \text{fix} [IVR \rightarrow M] \lambda(\text{fm}: IVR \rightarrow M) . \lambda(\text{inst'}: IVR). \text{methfun}(\text{inst'}, \text{fm})].\]

**Figure 12.8** Final translation of expressions of SOOL to \(\Lambda^\rho_C\). Part 2.
of a type safe MyType construct similar to that of Eiffel’s unsafe like Current expression, which was discussed in Section 4.2. The reason for this addition is to provide more accurate typing of self, especially when it is used in methods inherited in subclasses.

Perhaps surprisingly, we only need to make minor changes to the encoding introduced to this point in order to model MyType. Moreover, the encoding suggests safe type-checking rules for programs using this construct. However, it will turn out that MyType interferes with the possibility of subtypes. Instead a different, though similar, relation, called matching, will work more uniformly than subtyping in languages with MyType.

We will come back and discuss these extensions to $\mathcal{S}_{OOL}$ in future chapters, but in the next chapter we dig deep into the technical details in proving the soundness of our translation.
This page intentionally left blank
13 SOOL’s Type System Is Safe (and Sound)

Why bother to have a static type system for a programming language unless it guarantees some safety properties in the run-time behavior of a language? After all, one of the main reasons for having a static type system is to uncover errors as early as possible. In this chapter we investigate whether type-checking rules can guarantee the absence of certain run-time errors.

In this chapter we show that SOOL’s static type-checking rules are safe in the sense that they are sufficient to rule out run-time type errors. We do this by providing careful mathematical proofs of the soundness of our translation.

For example, we will show that if types $S$ and $T$ of SOOL are in the subtype relation, then so are their translations. Similarly, if expression $M$ of SOOL has type $S$, then the translation of $M$ has a type that is the translation of $S$. In Section 13.3 we discuss type errors and type safety in more detail and use the fact that $\Lambda_{\leq}$ has a statically safe type system in order to deduce the extremely important result that SOOL’s type system is also safe. We finish the chapter with a brief discussion of general run-time errors.

The reader who is less interested in the mathematical proofs may find it easier to skip the detailed proofs in the first section of this chapter on a first reading, returning to them later. However, we strongly recommend that the rest of the chapter not be skipped. The discussion of errors and type safety highlights the goals of the formal analysis in this book.

13.1 The translation of SOOL to $\Lambda_{\leq}$ is sound

We say that a translation of SOOL to $\Lambda_{\leq}$ is sound if the translation of well-typed expressions of SOOL are well-typed, and the types of the translated expressions are the translations of the types of the original SOOL expres-
sions. That is, the translation is sound if the translation function $T_{C'}[:]$ preserves typing. We will first show that if two types of SOOL are subtypes, then their translations are subtypes in $\Lambda_P$. We will then show that if $C, E \vdash E: T$ in SOOL then its translation, $T_{C'}[C, E \vdash E: T]$, will have type $T_{C'}[T]$ via a translation of the typing derivation in SOOL to a typing derivation in $\Lambda_P$.

Before getting started on the proofs of soundness we first need some definitions and technical lemmas about the translations with respect to different type constraint systems. These results will be used in the proof of the soundness of the translation.

Because typing judgements depend on a collection of assumptions on type variables, we must translate the assumptions as well.

**Definition 13.1.1** If $C$ is a type constraint system and $E$ is a static type environment for SOOL, then let $T_{C'}[E'] = \{ x: T_{C'}[T] \mid (x: T) \in E \}$. Similarly, if $C'$ is a type constraint system for SOOL, let $T_{C'}[C'] = \{ v <: T_{C'}[T] \mid (v <: T) \in C' \}$.

The translation of type constraints drops constraints of the form $t = T$ because they have already been taken into consideration in computing $T_{C'}[t]$ for type $U$. Of course, $\Lambda_P$ does not even include this sort of type constraint, so if we are going to use $T_{C'}[C']$ in typing expressions of $\Lambda_P$, it is also necessary to have eliminated these type definitions.

The following lemma shows that adding a new bounded type identifier to $C$ makes no difference in the translation of type expressions.

**Lemma 13.1.2** Suppose $C$ and $C' = C \cup \{ t <: T \}$ are both type constraint systems. Then for all type expressions, $S$, $T_{C}[S] = T_{C'}[S]$.

**Proof.** The proof is by a trivial induction on the complexity of type expressions using the fact that $T_{C'}[t] = T_{C'}[t]$ by the remark after Definition 12.1.1.

The result of adding a new equational type constraint to $C$ in translating expressions is described in the following lemmas.

**Lemma 13.1.3** Let $C$ and $C'$ be type constraint systems such that $C' = C \cup C''$. Let $S$ be a type expression that contains no type identifiers introduced in $C''$. Then $T_{C'}[S] = T_{C'}[S]$.

The Lemma above follows immediately from the following result.

**Lemma 13.1.4** Let $C_0$ be a type constraint system containing only bounded constraints (i.e., no equations). For all $1 \leq i \leq n$, let $C_i = C_{i-1} \cup \{ t_i = T_i \}$ be a type constraint system, where $C'_i$ contains only bounded constraints. If $S$ is a type
expression that contains no type variables \( \tau_j \), for \( j > i \), then for all \( k \geq i \), \( \mathcal{T}_c \cdot [\alpha] = \mathcal{T}_c \cdot [\alpha] \).

**Proof.** By induction on \( i \). If \( i = 0 \), then \( \alpha \) contains no type variables involved in a type definition in any of the \( \mathcal{C}_j \). In this case it is easy to use Lemma 13.1.2 to prove by induction on the complexity of \( \alpha \) that \( \mathcal{T}_c \cdot [\alpha] = \mathcal{T}_c \cdot [\alpha] \) for all \( k \).

Suppose the lemma is true for all \( m < i \). We show it for \( i \). The proof is by induction on the complexity of \( \alpha \), where \( \alpha \) contains no type variables \( \tau_j \) for \( j > i \).

Most cases are trivial. The only interesting case is for type variables. If \( \alpha \) is a type variable, then it must be \( \tau_m \) for some \( m \leq i \).

If \( \alpha = \tau_m \) for \( m < i \), then by induction, \( \mathcal{T}_c \cdot [\tau_m] = \mathcal{T}_c \cdot [\tau_m] = \mathcal{T}_c \cdot [\tau_m] \) for all \( k \geq i \), because \( \mathcal{T}_c \cdot [\tau_m] \) contains no \( \tau_j \) with \( j \geq m \).

If \( \alpha = \tau_i \), then \( \mathcal{T}_c \cdot [\tau_i] = \mathcal{T}_c \cdot [\tau_i] \). But \( \mathcal{T}_c \cdot [\tau_i] \) does not contain \( \tau_j \) for \( j > i - 1 \). It follows by induction that \( \mathcal{T}_c \cdot [\tau_i] = \mathcal{T}_c \cdot [\tau_i] = \mathcal{T}_c \cdot [\tau_i] = \mathcal{T}_c \cdot [\tau_i] \) for all \( k \geq i \).

Finally if \( \alpha = u \) for some type variable different from all \( \tau_i \), then \( \mathcal{T}_c \cdot [u] = u = \mathcal{T}_c \cdot [u] \) for all \( k \).

Using the above lemma we show the impact of adding a new type definition to a type constraint used in a translation only adds a simple substitution to the previous translation.

**Lemma 13.1.5** Let \( \mathcal{C} \) be a type constraint system. Let \( t \) be a type variable not occurring in \( \mathcal{C} \) such that \( \mathcal{C}' = \mathcal{C} \cup \{ \tau = T \} \) is a type constraint system. Then

1. For all type expressions, \( \alpha, \mathcal{T}_c \cdot [\alpha] = [\mathcal{T}_c [\alpha]] / t \mathcal{T}_c \cdot [\alpha] \).

2. For all expressions, \( \alpha, \mathcal{T}_c \cdot [\alpha] = [\mathcal{T}_c [\alpha]] / t \mathcal{T}_c \cdot [\alpha] \).

**Proof.** The proof of the first part is a simple proof by induction on the construction of type expressions. The only non-trivial part is the case for type identifiers. If \( \tau' \neq \tau \), then \( \mathcal{T}_c \cdot [\tau'] = \mathcal{T}_c \cdot [\tau'] \), while \( \mathcal{T}_c \cdot [\tau] = \mathcal{T}_c \cdot [\tau] \).

However, \( T \) contains no occurrences of \( \tau \). Thus by Lemma 13.1.3, \( \mathcal{T}_c \cdot [\tau] = \mathcal{T}_c \cdot [\tau] \). Also, \( \mathcal{T}_c \cdot [\tau] = \tau \) because \( \tau \) does not occur in \( \mathcal{C} \). Thus \( \mathcal{T}_c \cdot [\tau] = \mathcal{T}_c \cdot [\tau] = [\mathcal{T}_c [\tau]] / t \mathcal{T}_c \cdot [\tau] \). The other parts of the proof follow easily.

The proof of the second part is a simple induction on the construction of expressions.

Now we show that iterating this process gives similar results.

**Lemma 13.1.6** Let \( \mathcal{C}_0 = \mathcal{C} \) and for all \( 1 \leq k \leq n \), let \( \mathcal{C}_k = \mathcal{C}_{k-1} \cup \{ t_k = T_k \} \) be type constraint systems. Then
1. For all type expressions, $S$, $\mathcal{T}_c \llbracket S \rrbracket = [\mathcal{T}_c \llbracket T_1 \rrbracket / t_1] \cdots [\mathcal{T}_c \llbracket T_k \rrbracket / t_k] \mathcal{T}_c \llbracket S \rrbracket$.

2. For all regular expressions, $E$, $\mathcal{T}_c \llbracket E \rrbracket = [\mathcal{T}_c \llbracket T_1 \rrbracket / t_1] \cdots [\mathcal{T}_c \llbracket T_k \rrbracket / t_k] \mathcal{T}_c \llbracket E \rrbracket$.

**Proof.** The proof is by induction on $k$ for $0 \leq k \leq n$. Both parts of the lemma are clearly true for $k = 0$ as no substitutions are made. Suppose both are true for $k$, for some $k \geq 0$. We show both are true for $k+1$.

We begin with type expressions, $S$. By Lemma 13.1.5,

$$\mathcal{T}_{c+1} \llbracket S \rrbracket = [\mathcal{T}_c \llbracket T_{k+1} \rrbracket / t_{k+1}] \mathcal{T}_c \llbracket S \rrbracket.$$ 

But by induction,

$$\mathcal{T}_c \llbracket T_{k+1} \rrbracket = [\mathcal{T}_c \llbracket T_1 \rrbracket / t_1] \cdots [\mathcal{T}_c \llbracket T_k \rrbracket / t_k] \mathcal{T}_c \llbracket T_{k+1} \rrbracket,$$

and

$$\mathcal{T}_c \llbracket S \rrbracket = [\mathcal{T}_c \llbracket T_1 \rrbracket / t_1] \cdots [\mathcal{T}_c \llbracket T_k \rrbracket / t_k] \mathcal{T}_c \llbracket S \rrbracket.$$

Thus,

$$\mathcal{T}_{c+1} \llbracket S \rrbracket = [\mathcal{T}_c \llbracket T_1 \rrbracket / t_1] \cdots [\mathcal{T}_c \llbracket T_k \rrbracket / t_k] \mathcal{T}_c \llbracket T_{k+1} \rrbracket / t_{k+1} = [\mathcal{T}_c \llbracket T_1 \rrbracket / t_1] \cdots [\mathcal{T}_c \llbracket T_k \rrbracket / t_k] \mathcal{T}_c \llbracket S \rrbracket / t_{k+1} \mathcal{T}_c \llbracket S \rrbracket.$$

The proof of the induction case is similar for the second part of the lemma.

When we process declarations of SOOC, the type constraint system is augmented with new type definitions. We show that addition of these definitions has no impact on the translation of static type environments and type constraints.

**Lemma 13.1.7** Let $C$ be a type constraint system and $E$ be a static type environment. Let $\tau$ be a type variable not occurring in $C$ or $E$ such that $C' = C \cup \{ \tau = T \}$ is a type constraint system. Then $\mathcal{T}_c \llbracket C' \rrbracket = \mathcal{T}_c \llbracket C \rrbracket$ and $\mathcal{T}_c \llbracket E \rrbracket = \mathcal{T}_c \llbracket E \rrbracket$.

**Proof.** $\mathcal{T}_c \llbracket C' \rrbracket = \mathcal{T}_c \llbracket C \rrbracket$ because equations are erased in the translation of $C'$. The fact that $\mathcal{T}_c \llbracket C \rrbracket = \mathcal{T}_c \llbracket C \rrbracket$ follows easily from Lemma 13.1.5 because $\tau$ does not occur in $C$. It again follows from the Lemma and the fact that $\tau$ does not occur in $E$ that $\mathcal{T}_c \llbracket E \rrbracket = \mathcal{T}_c \llbracket E \rrbracket$.

Finally we show that substitutions make no difference in translating types.

**Lemma 13.1.8** Let $C$ be a constraint system and $S$ a type expression. Then $\mathcal{T}_c \llbracket S \rrbracket = \mathcal{T}_c \llbracket \mathcal{C}(S) \rrbracket$. 
13.1 The translation of \( SOOL \) to \( \Lambda^p_c \) is sound

\[ \mathcal{T}_C \llbracket \tau \rrbracket = \mathcal{T}_C \llbracket \mathcal{C} (\tau) \rrbracket \text{ for } \tau \text{ a type variable.} \]

The other cases follow easily by induction.

Now that we have finished these technical preliminaries, we are ready to begin showing our translation is sound. We begin by showing that subtyping is preserved in the translation from \( SOOL \) to \( \Lambda^p_c \). This refinement of Theorem 11.2.1 is necessary because revisions of the type-checking rules changed the definition of \( \mathcal{C} \) to include type constraints.

**Theorem 13.1.9** Let \( S \) and \( T \) be types of \( SOOL \) such that \( C \vdash S <: T \). Then

\[ \mathcal{T}_C \llbracket C \rrbracket \vdash \mathcal{T}_C \llbracket S \rrbracket <: \mathcal{T}_C \llbracket T \rrbracket. \]

**Proof.** We give a proof by induction on the number of steps in the proof that \( C \vdash S <: T \). That is, we assume that for all proofs of length less than \( n \), if \( C \vdash S <: T \), then \( \mathcal{T}_C \llbracket C \rrbracket \vdash \mathcal{T}_C \llbracket S \rrbracket <: \mathcal{T}_C \llbracket T \rrbracket \). We now must show that the theorem holds for all subtype relations provable in exactly \( n \) steps.

Suppose the last step in the proof is \( TypeDef \llbracket S \rrbracket \). Then the last step shows that \( C \vdash S <: T \) based on a shorter proof that \( C \vdash C(S) <: C(T) \). By the induction hypothesis, \( \mathcal{T}_C \llbracket C \rrbracket \vdash \mathcal{T}_C \llbracket C(S) \rrbracket <: \mathcal{T}_C \llbracket C(T) \rrbracket \). However, by Lemma 13.1.8, \( \mathcal{T}_C \llbracket S \rrbracket = \mathcal{T}_C \llbracket C(S) \rrbracket \) and similarly for \( T \). Hence \( \mathcal{T}_C \llbracket C \rrbracket \vdash \mathcal{T}_C \llbracket S \rrbracket <: \mathcal{T}_C \llbracket T \rrbracket \).

If the last step in the proof is reflexivity for \( SOOL \), then \( S = T \), and hence by the reflexivity rule for \( \Lambda^p_c \), \( \mathcal{T}_C \llbracket C \rrbracket \vdash \mathcal{T}_C \llbracket S \rrbracket <: \mathcal{T}_C \llbracket S \rrbracket \).

Transitivity is straightforward, while the other structural rules are all similar. We do the rule \( Object \llbracket S \rrbracket \) as an example.

Suppose that \( C \vdash ObjectType M' <: ObjectType M \) follows from \( C \vdash M' <: M \). By induction, \( \mathcal{T}_C \llbracket C \rrbracket \vdash \mathcal{T}_C \llbracket M' \rrbracket <: \mathcal{T}_C \llbracket M \rrbracket \). Let \( Y \) be a type variable not occurring in \( \mathcal{T}_C \llbracket C \rrbracket \). By the function and product type subtyping rules of \( \Lambda^p_c \), it follows that

\[ \mathcal{T}_C \llbracket C \rrbracket \vdash Y \times (Y \rightarrow \mathcal{T}_C \llbracket M' \rrbracket) <: Y \times (Y \rightarrow \mathcal{T}_C \llbracket M \rrbracket). \]

Finally, by rule \( Exist \llbracket S \rrbracket \),

\[ \mathcal{T}_C \llbracket C \rrbracket \vdash \exists Y. Y \times (Y \rightarrow \mathcal{T}_C \llbracket M' \rrbracket) <: \exists Y. Y \times (Y \rightarrow \mathcal{T}_C \llbracket M \rrbracket). \]

But this simply shows that

\[ \mathcal{T}_C \llbracket C \rrbracket \vdash \mathcal{T}_C \llbracket ObjectType M' \rrbracket <: \mathcal{T}_C \llbracket ObjectType M \rrbracket. \]

Now we wish to show that if a type can be assigned to an expression of \( SOOL \), then the translation of that type can be assigned to the translated expression.
Theorem 13.1.10 Let $E$ be an expression of S{"{O}}OC that does not involve $\mathbf{null}$, and suppose $C, E \vdash E : S$. Then
\[ T_C[C], T_C[E] \vdash T_C[C, E : E : S] : T_C[S]. \]

Proof. Following the definition of the translation of expressions, the proof of the theorem is given by induction on the length of the deduction of the typing (not the structure of the expression!). We may assume without loss of generality that the typing of $E$ does not include any application of rule Type Abbrev since $T_C[S] = T_C[C(S)]$.

Because the proof is relatively long and notationally dense already, we omit the typing information inside of the translation function. That is, we write $T_C[E]$ rather than $T_C[C, E \vdash E : S]$ for the translation of expressions in most of the proof. The only exception will be the subsumption case because the added notation clarifies the explanation of the proof. Simplified notation should not result in any ambiguity in the other cases as we will always be discussing the translation in combination with the typing derivation for the expression.

The cases where the last step uses either of the type-checking rules Constant or Void are trivial. The cases for rules Value, Application, Record, and Reference follow easily by induction. The case for the Function rule follows from the derived rule for $n$-ary functions in Section 8.2. The cases for statements are similarly straightforward.

Subsumption Suppose the typing deduction ends with an application of the subsumption rule. In particular, suppose that $C, E \vdash E : T$ follows by subsumption from $C, E \vdash E : S$ because $C \vdash S <\vdash T$. By induction,
\[ T_C[C], T_C[E] \vdash T_C[C, E : E : S] : T_C[S]. \]

But by Theorem 13.1.9, $T_C[C] \vdash T_C[S] <\vdash T_C[S]$, and hence by subsumption in $\lambda^{<\vdash}$,
\[ T_C[C], T_C[E] \vdash T_C[C, E : E : S] : T_C[T]. \]

But $T_C[C, E \vdash E : T] = T_C[C, E \vdash E : S]$ by the definition of the translation, so
\[ T_C[C], T_C[E] \vdash T_C[C, E : E : T] : T_C[T]. \]

Identifier The case for identifiers is also straightforward. Suppose $C, E \vdash \text{id: } T$ because $(\text{id: } T) \in E$. Then by definition, $(\text{id: } T) \in T_C[E]$, and hence $T_C[C], T_C[E] \vdash \text{id: } T_C[T]$. 

13.1 The translation of SOOL to $\Lambda^S_{\leq}$ is sound

**Block** Suppose Block is

\[
\begin{align*}
t_1 = T_1; \ldots; t_n = T_n; \\
\text{const id}_1 : U_1 := E_1; \ldots; \text{id}_k : U_k := E_k; \\
\{ S \text{ return } E_i \}
\end{align*}
\]

Let $\text{TDefs}$ abbreviate the type declarations, and let $\text{CDefs}$ abbreviate the constant declarations in Block.

For $1 \leq i \leq n$, let

\[C_i = C \cup \{ t_1 = T_1, \ldots, t_i = T_i \},\]

and for $1 \leq j \leq k$, let

\[E_j = E \cup \{ \text{id}_1 : U_1, \ldots, \text{id}_j : U_j \}\]

Let $C_0 = C, C' = C_n, E_0 = E, \text{ and } E' = E_k$.

Suppose $C, E \vdash \text{Block}: T$ where $T$ does not contain any of $t_1, \ldots, t_n$, because

\[C, E \vdash \text{TDefs} \circ C', E, \quad C', E \vdash \text{CDefs} \circ C', E', \quad C', E' \vdash S: \text{Command}\]

and

\[C', E' \vdash E : T.\]

By induction,

\[T_C[C'], T_C[E'] \vdash T_C[S]: \text{Command},\]

and

\[T_C[C'], T_C[E'] \vdash T_C[E] : T_C[T].\]

Hence by the Sequencing rule for $\Lambda^S_{\leq}$,

\[T_C[C'], T_C[E'] \vdash T_C[S]; T_C[E] : T_C[T].\]

Suppose, moreover, that $C', E \vdash \text{CDefs} \circ C', E'$ holds because for all $1 \leq j \leq k, C', E_{j-1} \vdash E_j : U_j$. It follows by induction that, for all $1 \leq j \leq k$,

\[T_C[C'], T_C[E_{j-1}] \vdash T_C[E_j] : T_C[U_j].\]

From the last two displayed formulas and the (derived) type rule Let presented in Section 8.2 it follows that

\[T_C[C'], T_C[E'] \vdash \text{let } \text{id}_1 : T_C[U_1] := T_C[E_1] \text{ in } \ldots \]

\[\text{let } \text{id}_k : T_C[U_k] := T_C[E_k] \text{ in } T_C[S]; T_C[E] \text{ end } \ldots \text{ end: } T_C[T] \]
Let $\text{Block}_c$ be

\[
\text{const id}_1 : U_1 := E_1 \ldots \text{id}_k : U_k := E_k;
\{
\text{S return } E_i
\}
\]

Then the translation of $\text{Block}_c$ is the let expression above, and hence

\[
\mathcal{T}_C[\mathcal{C}], \mathcal{T}_C[\mathcal{E}] \vdash \mathcal{T}_C[\text{Block}_c] : \mathcal{T}_C[\mathcal{T}]
\]

By repeatedly applying Lemma 13.1.7, $\mathcal{T}_C[\mathcal{C}] = \mathcal{T}_C[\mathcal{C}]$ and $\mathcal{T}_C[\mathcal{E}] = \mathcal{T}_C[\mathcal{E}]$.

But by Lemma 13.1.6,

\[
\mathcal{T}_C[\text{Block}_c] = \left[\mathcal{T}_C[\mathcal{T}]/t_1\right] \ldots \left[\mathcal{T}_C[\mathcal{T}]/t_k\right] \mathcal{T}_C[\text{Block}_c]
\]

and

\[
\mathcal{T}_C[\mathcal{T}] = \left[\mathcal{T}_C[\mathcal{T}]/t_1\right] \ldots \left[\mathcal{T}_C[\mathcal{T}]/t_k\right] \mathcal{T}_C[\mathcal{T}]
\]

But $\left[\mathcal{T}_C[\mathcal{T}]/t_1\right] \ldots \left[\mathcal{T}_C[\mathcal{T}]/t_k\right] \mathcal{T}_C[\mathcal{T}] = \mathcal{T}_C[\mathcal{T}]$ because $\mathcal{T}$ does not contain any of the $t_i$.

Thus,

\[
\mathcal{T}_C[\mathcal{C}], \mathcal{T}_C[\mathcal{E}] \vdash \left[\mathcal{T}_C[\mathcal{T}]/t_1\right] \ldots \left[\mathcal{T}_C[\mathcal{T}]/t_k\right] \mathcal{T}_C[\text{Block}_c] : \mathcal{T}_C[\mathcal{T}].
\]

By the definition of the translation,

\[
\mathcal{T}_C[\text{Block}_c] = \text{let } t_1 = \mathcal{T}_C[\mathcal{T}_1] \text{ in let } t_2 = \mathcal{T}_C[\mathcal{T}_2] \text{ in } \ldots \text{ let } t_n = \mathcal{T}_C[\mathcal{T}_n] \text{ in }
\]

\[
\mathcal{T}_C[\text{Block}_c] \text{ end } \ldots \text{ end}
\]

But it then follows from the previous typing judgement and the derived TypeLet rule for type let expressions in Section 9.1 that

\[
\mathcal{T}_C[\mathcal{C}], \mathcal{T}_C[\mathcal{E}] \vdash \mathcal{T}_C[\text{Block}_c] : \mathcal{T}_C[\mathcal{T}],
\]

as desired.

**Class** Suppose that

\[
\mathcal{C}, \mathcal{E} \vdash \text{class(} \text{inst, meth}\text{)} : \text{ClassType}(\mathcal{IV}, \mathcal{M})
\]

because

\[
\mathcal{C}, \mathcal{E} \vdash \text{inst: } \mathcal{IV} \text{ and } \mathcal{C}', \mathcal{E}' \vdash \text{meth: } \mathcal{M}
\]

for
\[ C' = C \cup \{ \text{M'} \prec C, IVR' \prec IV^{\text{ref}}, \text{SelfType} = \text{VisObject}(IVR', M') \}, \]
\[ E' = E \cup \{ \text{self: SelfType}, \text{close: SelfType} \to \text{ObjectType} M' \}, \]
and
\[ \text{SelfType} \text{ does not occur in IV or M.} \]

Now
\[ T_C[C, E] \vdash \text{class(inst, meth): ClassType(IV, M)} \triangleq \lambda (M' \prec T_C[M]). \lambda (IVR' \prec T_C[IV^{\text{ref}}]). (T_C[C, E] \vdash \text{inst: IV}, \text{methfun}), \]
where
\[ \text{methfun} \triangleq \lambda (\text{self: VisObj}(IVR', M')). \]
\[ \text{let close: VisObj}(IVR', M') \to (\exists Y. \text{VisObj}(Y, M')) \]
\[ \triangleq \text{closeobj}[IVR'[M']] \]
in \[ T_C[C', E'] \vdash \text{meth: M} \]

and
\[ T_C[\text{ClassType}(IV, M)] = \forall (M' \prec T_C[M]). \forall (IVR' \prec T_C[IV^{\text{ref}}]). (T_C[IV] \times (\text{VisObj}(IVR', M') \to T_C[M]) \]

By induction,
\[ T_C[C'], T_C[E] \vdash T_C[\text{inst}]: T_C[IV] \]
and
\[ T_C[C'], T_C[E'] \vdash T_C[\text{meth}]: T_C[M]. \]

It follows from Lemma 13.1.5, our translation rules, and the definition of \( \text{VisObj}(IVR', M') \), that
\[ T_C[\text{SelfType}] = T_C[\text{VisObject}(IVR', M')] = \text{VisObj}(IVR', M'). \]

Let
\[ \text{meth} \triangleq \lambda (\text{let close: VisObj}(IVR', M') \to (\exists Y. \text{VisObj}(Y, M'))) \]
\[ \triangleq \text{closeobj}[IVR'[M']] \]
in \[ T_C[C', E'] \vdash \text{meth: M}. \]

and hence \( \text{methfun} = \lambda (\text{self: VisObj}(IVR', M')). \text{meth} \).

Let \( E'' \triangleq E \cup \{ \text{self: SelfType} \} \). Then by the derived \textit{Let} rule for \( \Lambda_C^{\text{ref}} \),
\[ T_C[C''], T_C[E''] \vdash \text{meth: } T_C[M]. \]
By Lemmas 13.1.2 and 13.1.5, \( T_C[Z] = \{\text{VisObj}(IVR',M')/\text{SelfType}\} T_C[Z] \) for all \( Z \). Therefore,

\[
T_C[C'], T_C[E'] \vdash \text{meth}: \{\text{VisObj}(IVR',M')/\text{SelfType}\} T_C[M].
\]

But \( M \) does not contain \text{SelfType}. Thus,

\[
T_C[C'], T_C[E'] \vdash \text{meth}: T_C[M].
\]

By the fact that \( T_C[E'] = T_C[E] \cup \{\text{self: VisObj}(IVR',M')\} \) and the \text{Function} rule for \( \Lambda^P_C \),

\[
T_C[C'], T_C[E] \vdash \langle T_C[\text{inst}], \text{methfun}: \text{VisObj}(IVR',M') \to T_C[M] \rangle
\]

It follows from this and the type-checking rules \text{Function} and \text{Pair} of \( \Lambda^P_C \) that

\[
T_C[C'], T_C[E] \vdash \langle T_C[\text{inst}], \text{methfun}: \text{VisObj}(IVR',M') \to T_C[M] \rangle
\]

Now

\[
T_C[C'] = T_C[C] \cup \{M' \vdash: T_C[M], IVR' \vdash: T_C[IV']\} = T_C[C] \cup \{M' \vdash: T_C[M], IVR' \vdash: T_C[IV']\}
\]

because none of \( C, M, \) or \( IV' \) contain \text{SelfType}. Similarly \( T_C[E] = T_C[E] \)

because \( E \) does not contain \text{SelfType}.

Finally, by the expansion of \( T_C[C'] \) above and the type-checking rule \text{Bd-PolyFunc} in Figure 9.9,

\[
T_C[C], T_C[E] \vdash \Lambda(M' \vdash: T_C[M]). \Lambda(IVR' \vdash: T_C[IV'])
\]

\[
\langle T_C[\text{inst}], \text{methfun}: \forall(M' \vdash: T_C[M]), \forall(IVR' \vdash: T_C[IV']): T_C[IV'] \times (\text{VisObj}(IVR',M') \to T_C[M]) \rangle
\]

Thus

\[
T_C[C], T_C[E] \vdash T_C[\text{class}(\text{inst}, \text{meth})]: T_C[\text{ClassType}(IV,M)].
\]

**Subclass** The case for subclasses is not much more difficult than that for classes, and we take advantage of some of the work done in that proof.

Suppose that

\[
C, E \vdash \text{class inherits } C \text{ modifies } l_1, \ldots, l_m (\text{inst}, \text{meth}): \text{ClassType}(IV,M)
\]
because
\[
\mathcal{C}, \mathcal{E} \vdash \mathcal{C} : \text{ClassType}([\text{IV}_{\sup}, M_{\sup}],
\mathcal{C}, \mathcal{E} \vdash \text{inst} : \text{IV}_{\sub},
\mathcal{C}', \mathcal{E}' \vdash \text{meth} : M_{\sub}, \text{and}
\mathcal{C} \vdash T'_{ij} : T_{ij} \text{ for } 1 \leq j \leq m.
\]
where
- \( \text{IV} = \text{IV}_{\sup} \oplus \text{IV}_{\sub} \) (and hence, \( \text{IV}^{\text{ref}} < : \text{IV}_{\sup}^{\text{ref}} \)),
- \( M = M_{\sup} \oplus M_{\sub} \) (and hence, \( M < : M_{\sup} \)),
- there is no overlap in the labels occurring in \( \text{IV}_{\sup} \) and \( \text{IV}_{\sub} \), and
- the overlapping labels in \( M_{\sup} \) and \( M_{\sub} \) are exactly \( l_{i_1}, \ldots, l_{i_m} \).
- The type of \( l_{i_j} \) in \( M_{\sub} \) is \( T'_{ij} \), while the corresponding type in \( M_{\sup} \) is \( T_{ij} \).
- \( \mathcal{C}' = \mathcal{C} \cup \{ M' < : M, \text{IVR}' < : \text{IV}^{\text{ref}}, \text{SelfType} = \text{VisObjectType}([\text{IVR}', M']) \} \),
- \( \mathcal{E}' = \mathcal{E} \cup \{ \text{self}: \text{SelfType}, \text{close}: \text{SelfType} \to \text{ObjectType} M' \} \), and
- \( \text{SelfType} \) does not occur in \( \text{IV} \) or \( M \).

By induction,
\[
\begin{align*}
\mathcal{T}_C[\mathcal{C}], \mathcal{T}_C[\mathcal{E}] & \vdash \mathcal{T}_C[\mathcal{C} : \text{ClassType}([\text{IV}_{\sup}, M_{\sup}]),
\mathcal{T}_C[\mathcal{C}], \mathcal{T}_C[\mathcal{E}] & \vdash \mathcal{T}_C[\text{inst} : \text{IV}_{\sub}],
\mathcal{T}_C[\mathcal{C}'], \mathcal{T}_C[\mathcal{E}'] & \vdash \mathcal{T}_C[\text{meth} : M_{\sub}],
\end{align*}
\]
and
\[
\mathcal{T}_C[\mathcal{C}] \vdash \mathcal{T}_C[T'_{ij}] < : \mathcal{T}_C[T_{ij}] \text{ for } 1 \leq j \leq m.
\]
It is easy to see that
\[
\mathcal{T}_C[\text{IV}] = \mathcal{T}_C[\text{IV}_{\sup}] \oplus \mathcal{T}_C[\text{IV}_{\sub}],
\]
and
\[
\mathcal{T}_C[M] = \mathcal{T}_C[M_{\sup}] \oplus \mathcal{T}_C[M_{\sub}].
\]
By definition,
\[
\begin{align*}
\mathcal{T}_C[\mathcal{C}, \mathcal{E}] & \vdash \text{class inherits } \mathcal{C} \text{ modifies } l_{i_1}, \ldots, l_{i_m} \text{ (inst, meth):} \\
\text{ClassType}([\text{IV}, M] & \triangleq \\
\Lambda(M' < : \mathcal{T}_C[M]). \Lambda(\text{IVR}' < : \mathcal{T}_C[\text{IV}^{\text{ref}}])(\text{inst}_{\text{all}}, \text{methfun}_{\text{all}}),
\end{align*}
\]
where

\[ c = \mathcal{T}_C[c',\mathcal{E} \vdash c : \text{ClassType}(IV_{sup}, M_{sup})][M'][IV'R] \]
\[ \text{inst}_{sup} = \text{proj}_1(c) \]
\[ \text{methfun}_{sup} = \text{proj}_2(c) \]
\[ \text{inst}_{all} = \text{inst}_{sup} \uplus \mathcal{T}_C[c',\mathcal{E} \vdash \text{inst: IV}_{sub}] \]
\[ \text{methfun}_{all} = \lambda (\text{self: VisObj}(IVR', M')). \]
\[ \text{let close: VisObj}(IVR', M') \rightarrow (\exists Y. \text{VisObj}(Y, M')) \]
\[ \quad = \text{closeobj}(IVR')[M'] \]
\[ \text{in methfun}_{sup}(\text{self}) \uplus \mathcal{T}_C[c',\mathcal{E}' \vdash \text{meth: M}_{sub}]. \]

Also,

\[ \mathcal{T}_C[\text{ClassType}(IV_{sup}, M_{sup})] = \forall (M' : \mathcal{T}_C[M_{sup}]). (IVR' : \mathcal{T}_C[IV'_{sup}]). \]
\[ \mathcal{T}_C[IV_{sup}] \times (\text{VisObj}(IVR', M') \rightarrow \mathcal{T}_C[M_{sup}]) \]

By Theorem 13.1.9,

\[ \mathcal{T}_C[c] \vdash \mathcal{T}_C[IV'_{sup}] \ll \mathcal{T}_C[IV'_{sup}] \text{ and } \mathcal{T}_C[c] \vdash \mathcal{M} \ll \mathcal{T}_C[M_{sup}]. \]

By the definition of \text{inst}_{sup} and \text{inst}_{all}, the induction hypothesis, and the Projection rule for \( \Lambda^{P'}, \)

\[ \mathcal{T}_C[c], \mathcal{T}_C[\mathcal{E}] \vdash \text{inst}_{sub}: \mathcal{T}_C[IV_{sub}], \]
\[ \mathcal{T}_C[c], \mathcal{T}_C[\mathcal{E}] \vdash \text{inst}_{all}: \mathcal{T}_C[IV]. \]

By the same reasoning as for classes,

\[ \mathcal{T}_C[c'], \mathcal{T}_C[\mathcal{E}'] \vdash \text{methfun}_{all}: \text{VisObj}(IVR', M') \rightarrow \mathcal{T}_C[M_{sup}]. \]

Thus

\[ \mathcal{T}_C[c'], \mathcal{T}_C[\mathcal{E}'] \vdash \text{methfun}_{sup}(\text{self}) \uplus \mathcal{T}_C[\text{meth: M}] \]

By the Function and Pair type-checking rules, it follows that

\[ \mathcal{T}_C[c'], \mathcal{T}_C[\mathcal{E}] \vdash (\text{inst}_{all}, \text{methfun}_{all}): \mathcal{T}_C[IV] \times \text{VisObj}(IVR', M') \rightarrow \mathcal{T}_C[M]. \]

As in the case for classes,

\[ \mathcal{T}_C[c'] = \mathcal{T}_C[c] \cup \{ M' \ll \mathcal{T}_C[M], IVR' \ll \mathcal{T}_C[IV'] \} \]
13.1 The translation of SOOL to $\Lambda_\alpha^L$ is sound

and $\mathcal{T}_c[\mathcal{E}] = \mathcal{T}_c[\mathcal{E}]$. Thus by the BdPolyFunc rule,

$$\mathcal{T}_c[\mathcal{C}], \mathcal{T}_c[\mathcal{E}] \vdash \Lambda(M' : \mathcal{T}_c[\mathcal{M}]); \Lambda(IVR' : \mathcal{T}_c[IV^{\text{ref}}]); (\text{inst}_{\text{all}}, \text{methfun}_{\text{all}});$$

$$\forall (M' : \mathcal{T}_c[\mathcal{M}]); \forall (IVR' : \mathcal{T}_c[IV^{\text{ref}}]). \mathcal{T}_c[IV] \times (\text{VisObj}(IVR', M') \rightarrow \mathcal{T}_c[\mathcal{M}])$$

and hence,

$$\mathcal{T}_c[\mathcal{C}], \mathcal{T}_c[\mathcal{E}] \vdash \mathcal{T}_c[\text{class inherits C modifies } l_{i_1}, \ldots, l_{i_m} (\text{inst, meth})];$$

$$\mathcal{T}_c[\text{ClassType}(IV, M)]$$

**New** Suppose that

$\mathcal{C}, \mathcal{E} \vdash \text{new C: ObjectType M}$

because

$\mathcal{C}, \mathcal{E} \vdash \text{C: ClassType}(IV, M)$

By induction,

$$\mathcal{T}_c[\mathcal{C}], \mathcal{T}_c[\mathcal{E}] \vdash \mathcal{T}_c[\mathcal{C}]; \mathcal{T}_c[\text{ClassType}(IV, M)]$$

The revised translation of new C is

$$\mathcal{T}_c[\mathcal{C}, \mathcal{E}] \vdash \text{new C: ObjectType M}] \triangleq \text{closeobj}[IVR][M](\text{inst}^{\text{ref}}, \text{methfun}^{'})$$

where

$\mathcal{c} \triangleq \mathcal{T}_c[\mathcal{C}, \mathcal{E}] \vdash \text{C: ClassType}(IV, M); [M][IVR]$

$IVR = \mathcal{T}_c[IV^{\text{ref}}]$,

$M = \mathcal{T}_c[\mathcal{M}]$,

$\text{inst} = \text{proj}_1(c)$,

$\text{methfun} = \text{proj}_2(c)$,

$\text{inst}^{\text{ref}}$ is formed by adding refs to all fields of inst, and

$\text{methfun}' = \text{fix}[IVR \rightarrow M] \lambda (fm: IVR \rightarrow M). \lambda (\text{inst}' : IVR). \text{methfun}(\langle \text{inst}', fm \rangle)$. 

Because

$$\mathcal{T}_c[\text{ClassType}(IV, M)] =$$

$$\forall M' : \mathcal{T}_c[\mathcal{M}]. \forall IVR' : \mathcal{T}_c[IV^{\text{ref}}]. \mathcal{T}_c[IV] \times (\text{VisObj}(IVR', M') \rightarrow \mathcal{T}_c[\mathcal{M}])$$
it follows that

\[ \mathcal{T}_c[C], \mathcal{T}_c[E] \vdash c : \mathcal{T}_c[\mathcal{M}] \times \text{VisObj}(\mathcal{T}_c[\mathcal{L}], \mathcal{T}_c[M]) \rightarrow \mathcal{T}_c[M] \]

\[ \mathcal{T}_c[C], \mathcal{T}_c[E] \vdash \text{inst} : \mathcal{T}_c[\mathcal{M}] \]

\[ \mathcal{T}_c[C], \mathcal{T}_c[E] \vdash \text{methfun} : \text{VisObj}(\mathcal{T}_c[\mathcal{L}], \mathcal{T}_c[M]) \rightarrow \mathcal{T}_c[M] \]

\[ \mathcal{T}_c[C], \mathcal{T}_c[E] \vdash \text{instref} \mathcal{T}_c[\mathcal{L}], \text{and} \]

\[ \mathcal{T}_c[C], \mathcal{T}_c[E] \vdash \text{methfun'} : \mathcal{T}_c[\mathcal{L}] \rightarrow \mathcal{T}_c[M]. \]

Hence

\[ \mathcal{T}_c[C], \mathcal{T}_c[E] \vdash \langle \text{instref}, \text{methfun'} \rangle : \mathcal{T}_c[\mathcal{L}] \times (\mathcal{T}_c[\mathcal{L}] \rightarrow \mathcal{T}_c[M]). \]

Recall that

\[ \emptyset, \emptyset \vdash \text{closeobj} : \forall \mathbb{A}. \forall \mathbb{M}. (\mathbb{A} \times (\mathbb{A} \rightarrow \mathbb{M})) \rightarrow (\exists \mathbb{Y} \times (\mathbb{Y} \rightarrow \mathbb{M})). \]

As a result

\[ \mathcal{T}_c[C], \mathcal{T}_c[E] \vdash \text{closeobj} \mathcal{T}_c[\mathcal{L}] (\mathcal{T}_c[\mathcal{L}]) (\mathcal{T}_c[M]) \langle \text{instref}, \text{methfun'} \rangle : (\exists \mathbb{Y} \times (\mathbb{Y} \rightarrow \mathcal{T}_c[M])). \]

Because \( \mathcal{T}_c[\text{ObjectType M}] = \exists \mathbb{Y} \times (\mathbb{Y} \rightarrow \mathcal{T}_c[M]) \), it follows that

\[ \mathcal{T}_c[C], \mathcal{T}_c[E] \vdash \mathcal{T}_c[\text{new C : ObjectType M}], \]

as desired.

**Message** Suppose

\[ C, E \vdash E \leftarrow m : T \]

because \( C, E \vdash E : M \) and \( C \vdash M \leftarrow \text{ObjectType} \{ m : T \} \).

By induction,

\[ \mathcal{T}_c[C], \mathcal{T}_c[E] \vdash \mathcal{T}_c[C, E \vdash E : M] : \mathcal{T}_c[M] \]

and

\[ \mathcal{T}_c[C] \vdash \mathcal{T}_c[M] \leftarrow \text{ObjectType} \{ m : T \} \]

By subsumption in \( \lambda^P_{<} \),

\[ \mathcal{T}_c[C], \mathcal{T}_c[E] \vdash \mathcal{T}_c[C, E] : \mathcal{T}_c[\text{ObjectType} \{ m : T \}]. \]
By definition,
\[ T_c[\text{ObjectType}] \vdash m : T \Downarrow = \exists Y. Y \times (Y \to \|m: T_c[T]\|) \]
and
\[ T_c[E \leftarrow m] = \text{open } T_c[E] \text{ as } \langle \text{IVR}, vo \rangle \text{ in } (\text{proj}_1(vo)(\text{proj}_1(vo))).m \]

It follows from the projection typing rules and the Selection rule of \( \Lambda^P \) that
\[ T_c[C], T_c[E] \cup \{ vo : \text{IVR} \times (\text{IVR} \to T_c[M]) \vdash (\text{proj}_1(vo)(\text{proj}_1(vo))).m : T_c[T] \]

It follows from 13.1 and the Unpack rule that
\[ T_c[C], T_c[E] \vdash \text{open } T_c[E] \text{ as } \langle \text{IVR}, vo \rangle \text{ in } (\text{proj}_1(vo)(\text{proj}_1(vo))).m : T_c[T] \]

Hence
\[ T_c[C], T_c[E] \vdash T_c[E \leftarrow m] : T_c[T] \]

**Inst Vble** Suppose
\[ C, E \vdash E.1 : T \]
because \( C, E \vdash E : \text{VisObjectType(IVR,M)} \) and \( C \vdash IVR < : \| 1 : T \| \).

By induction,
\[ T_c[C], T_c[E] \vdash T_c[E] : T_c[\text{VisObjectType(IVR,M)}] \]
and
\[ T_c[C] \vdash T_c[IVR] < : \| 1 : T_c[T] \| . \]

By definition,
\[ T_c[\text{VisObjectType(IVR,M)}] = T_c[IVR] \times (T_c[IVR] \to T_c[M]) \]
and
\[ T_c[E.1] = \text{proj}_1(T_c[E]).l \]

By the Proj type-checking rule,
\[ T_c[C], T_c[E] \vdash \text{proj}_1(T_c[E]) : T_c[IVR] \]
and by subsumption in \( \Lambda^P \).
\[ T_c[C], T_c[E] \vdash \text{proj}_1(T_c[E]) : \| 1 : T_c[T] \| . \]

By the Selection rule,
\[ T_c[C], T_c[E] \vdash \text{proj}_1(T_c[E]).k : T_c[T] \]

Hence,
\[ T_c[C], T_c[E] \vdash T_c[E.1] : T_c[T] . \]
**Self Message**  This case is similar to that for sending messages to regular objects except that it is not necessary to unpack the existential. However, because there is no subtyping between visible object types we will need to do things in a slightly different order.

Suppose

\[ C, E \vdash E \Leftarrow m : T \]

because \( C, E : \text{VisObjectType}(\text{IVR}, M) \) and \( C : \{ m : T \} \).

By induction,

\[ T_C[C], T_C[E] \vdash T_C[C, E : \text{VisObjectType}(\text{IVR}, M)] : T_C[\text{VisObjectType}(\text{IVR}, M)] \]

and

\[ T_C[C] : T_C[M] : \{ m : T \} \]

By definition,

\[ T_C[\text{VisObjectType}(\text{IVR}, M)] = \text{IVR} \times (\text{IVR} \rightarrow T_C[M]) \]

and

\[ T_C[E : \Leftarrow m] = (\text{proj}_2(T_C[E]) \circ (\text{proj}_1(T_C[E]))) \cdot m. \]

Thus,

\[ T_C[C], T_C[E] \vdash T_C[E] : \text{IVR} \times (\text{IVR} \rightarrow T_C[M]) \]

It follows from the projection typing rules and the \textit{FuncApp} rule of \( \Lambda_C^P \) that

\[ T_C[C], T_C[E] \vdash \text{proj}_2(T_C[E]) \circ (\text{proj}_1(T_C[E])) : T_C[M] \]

By subsumption in \( \Lambda_C^P \),

\[ T_C[C], T_C[E] \vdash \text{proj}_2(T_C[E]) \circ (\text{proj}_1(T_C[E])) : \{ m : T \} \]

and hence by the \textit{Selection} rule,

\[ T_C[C], T_C[E] \vdash (\text{proj}_2(T_C[E]) \circ (\text{proj}_1(T_C[E]))) \cdot m : T_C[T] \]

Hence

\[ T_C[C], T_C[E] \vdash T_C[E \Leftarrow m] : T_C[T] \]
13.2 The translation is well defined

The proof of Theorem 13.1.10 in the previous section essentially showed how to translate the derivation of \( \mathcal{C}, \mathcal{E} \vdash E : T \) in SOOL to a corresponding derivation of \( T_\mathcal{C}[\mathcal{C}], T_\mathcal{E}[\mathcal{E}] \vdash T_\mathcal{E}[\mathcal{E}] : T_\mathcal{C}[\mathcal{C}] \) in \( \Lambda^P \). However, that is not enough to ensure that the definition of \( T_\mathcal{C}[\mathcal{C}, \mathcal{E} \vdash E : T] \) is well-defined.

The problem arises because of the possibility of two distinct derivations of \( \mathcal{C}, \mathcal{E} \vdash E : T \). Because the definition of \( T_\mathcal{C}[\mathcal{C}, \mathcal{E} \vdash E : T] \) is given by induction on the derivation of the typing, two distinct derivations might give rise to distinct expressions of \( \Lambda^P \), even though both would end up having the same type.

The CellExample program in Figure 13.1 illustrates the problem. The figure contains two different class type definitions, CellClassType1 and CellClassType2. The class Cell in the figure will type-check correctly no matter which of these types is assigned to Cell. However, the two different typings result in two different translations of Cell. If translated with type CellClassType1 the method suite has both methods, while if translated with CellClassType2, the method suite has only the increase method, even though the actual code for CellClass includes both. As a result, the translation of new Cell will have either two or one methods depending on the type given Cell. Both of these can be assigned to cell, which has type CellType2, though the version in which Cell has type CellClassType1 needs the use of subsumption in typing the assignment statement.

This example shows that if one of the subexpressions of a SOOL program segment is given different types, the translations may not be the same.

In this particular example we were forced to declare a type for Cell when we wrote the program, but what if we substituted the definition of Cell into the new Cell expression in place of the identifier Cell? Now we have a context in which we could easily choose either typing derivation for the typing of new Cell, and hence get two different translations of the expression.

The two different translations of new Cell are not such a big problem as the types of the translations are subtypes and it is not hard to convince oneself that any context using cell and returning an element of a base type will get the same results no matter which translation is chosen. In particular, the bump method is not accessible from cell because it has type CellType2.

On the other hand, the two different translations of Cell are more of a problem. Not only do they have different class types, but those class types are not subtypes – a class type can only be a subtype of itself.

We can solve this problem by requiring that all classes be declared in the
Program CellExample;

  type
      CellType1 = ObjectType
        increase: Integer -> Command;
        bump: Void -> Command
    ;
      CellType2 = ObjectType
        increase: Integer -> Command
    ;
      CellClassType1 = ClassType(
        x: Integer
      , 
        increase: Integer -> Command;
        bump: Void -> Command
    );
      CellClassType2 = ClassType(
        x: Integer
      , 
        increase: Integer -> Command;
    );
  const
      Cell: CellClassType? = class(
        x: Integer = 0
      , 
        increase: Integer -> Command =
          function(dx: Integer): Command is {
            self.x := (val self.x) + dx;
            return nop
        });
      bump: Void -> Command =
        function(): Command is {
            self.x := (val self.x) + 1;
            return nop
        });
      cell: Ref CellType2 = ref nil;

      { 
        cell := new Cell;
      }

  // Figure 13.1  CellExample program.
constant section of the program rather than allowing them to be used in a more complex context. Because they are declared along with their types, there is no problem in figuring out with which type they should be translated. This does have the negative consequence that the type of a class expression must now be partially determined from the context of its definition, rather than based on its own internal structure.

A similar though perhaps superior solution in practice for the version of the language with abbreviations (in which programmers normally do not specify class types for class definitions) is to always type classes with the full collection of instance variables and methods in their definitions. The types of class features are always determined by their declarations, either directly for instance variables, or indirectly from the function declarations for methods. The type obtained in this way is essentially the most expressive type for the class and is the one that would naturally be expected by programmers. If the source language that programmers write in allowed identifiers to represent records then we might have added difficulties, but records are not directly supported in $SOOL$, they are only a convenience for type-checking. As a result, the feature declarations are always directly visible in class declarations.

What about the problems arising with the possibility of two different type derivations for the same expression and type? Aside from classes and subclasses, only the translation of object types depends on the type of the expression being translated. However, the translation of $\text{new } C$ depends only on the type of $C$. Thus if $C$ has a unique type (as it will with the conventions above), then the translation of $\text{new } C$ will depend only on that unique type. In other words, with this new restriction on the admissible expressions of class type, the translation no longer depends on type derivations. Instead it can be given inductively on the structure of the expressions. Of course the proof of soundness will still be given by induction on the length of the derivation of types as it is essentially a transformation from type derivations of $SOOL$ to type derivations in $\Lambda^P_{\subseteq}$.

Alternatively one could continue to use the definition of translation that depends on typing derivations. In this case, one could first show that every typing derivation can be converted to a canonical derivations (for example, by pushing uses of subsumption to uses on as small expressions as possible). (See Section 10.4 of [Mit96], for a similar example.) One could then define the translation as being based on canonical transformations or show that the steps in converting to a canonical derivation all preserve the translation (after all, recall that uses of subtyping have no effect on the translation).

Because the simple modification of $SOOL$ resulting in unique types for
classes is sufficient to ensure that the translation is well-defined, we do not include further discussion of this issue here.

13.3 SOOl’s Type System Is Safe (and Sound)

We would like the type system in an object-oriented language to guarantee that if a message is sent to an object, then that object has a method that can handle the message. The object-oriented programming language Smalltalk [GR83] is not statically typed. As a result, at run time it is possible to get an error message stating “message not understood”. This indicates that the receiving object does not have a method with matching name and signature to that of the message.

In Section 13.1, we showed that the translation of SOOl into $\Lambda_P$ is type-preserving. In this section we would like to go further and claim that SOOl is statically type safe. That is, well-typed SOOl programs are guaranteed not to have certain kinds of errors, called type errors, at run time. We begin by providing some background on the notion of safety in programming languages.

Cardelli [Car97] distinguishes between what he calls trapped and untrapped errors. Trapped errors cause a computation to stop immediately, while untrapped errors may allow the computation to continue (at least for a while). An example of an untrapped error might be accessing data beyond the end of an array in a language like C. Cardelli argues that a safe language is one that does not allow any untrapped errors to occur at run time.

In a given language, a subset of the errors may be designated as forbidden errors. The forbidden errors should include all untrapped errors as well as a subset of trapped errors. A language where none of the legal programs can generate forbidden errors is said to be strongly checked.

For SOOl, as with most typed programming languages, the forbidden errors will include all run-time type errors. A type error is an error that can be characterized as being an error based just on the knowledge of the types of the components. For example, if $f$ is a function of type $\text{Integer} \rightarrow \text{Integer}$ and $\text{arg}$ has type $\text{Boolean}$, then $f(\text{arg})$ results in a type error.

Similarly it is erroneous to attempt to apply an object (as though it were a function) to an integer. The type error we are most concerned with in object-oriented programming languages is to send a message to an object that does not have a method corresponding to the message. This is a type error, because the type of the object specifies the list of messages that it supports.
13.3 \textit{SoOCL} is type safe

A \textit{type-safe language} is one in which all type errors are trapped errors. A \textit{static type-safe language} is one in which all type errors are caught statically, \textit{e.g.}, at compile time if a compiler is used.

Smalltalk, LISP, and Scheme are examples of type-safe languages that are not statically type safe. Java, ML, and Haskell are statically type safe.

Examples of non-type errors include dividing an integer by zero (since both operands are integers, the division makes sense based on the types). Similarly sending a message to a nil object need not be a type error as long as the static type of the nil object includes the message name. Accessing data beyond the end of an array is also not a type error (though it hopefully is a forbidden error in the language). Finally, a non-terminating program is generally not the result of a type error.

Statically type-safe languages ensure that programs that pass the type-checker will not generate type errors at run time (though other forbidden errors may be detected and handled at run time). Programs that do not pass the type-checker are not considered legal programs. Semantics are generally only provided for legal programs of the language.

In the last chapter we provided a translation from \textit{SoOCL} into $\lambda^P_{<:}$. We proved in Section 13.1 that well-typed expressions of \textit{SoOCL} were translated into well-typed expressions of $\lambda^P_{<:}$. The semantics of $\lambda^P_{<:}$ are well understood in that there exist both denotational and operational semantics. The translation from \textit{SoOCL} into $\lambda^P_{<:}$ provides a semantics for well-typed expressions of \textit{SoOCL} by first translating to the corresponding $\lambda^P_{<:}$ program and then determining the semantics of that program.

As defined above, static type safety is a property defined in terms of the run-time behavior of a language. Thus it is most naturally seen as a property of an operational semantics. It is possible to write down a straightforward operational semantics for $\lambda^P_{<:}$ (recall we only gave fragments of such a semantics) and show that $\lambda^P_{<:}$ is statically type safe.\textsuperscript{1} Using this we can show that \textit{SoOCL} is statically type safe.

\textbf{Theorem 13.3.1} \textit{SoOCL} is statically type safe.

\textbf{Proof.} Let $p$ be a program of \textit{SoOCL} that is well-typed (i.e., it typable according to the type-checking rules of \textit{SoOCL}). Recall that $p$ is executed by

\textsuperscript{1} There appears to be no published proof that the entire language $\lambda^P_{<:}$ is type safe. Instead separate proofs of the type safety of the different parts can be found in \cite{Pie02} and other sources. We expect that, aside from the expressions corresponding to the universal and existential types, the language is simple enough that the reader is willing to trust in the type safety of $\lambda^P_{<:}$. 
first using the translation function to obtain a program $P'$ of $\Lambda_c^P$, and then executing $P'$. By Theorem 13.1.10, $P'$ is also well-typed. However, by the fact that $\Lambda_c^P$ is statically type safe, the execution of $P'$ cannot result in a type error. Thus the execution of $P$ cannot result in a type error. 

It is possible to provide a direct operational semantics for $\mathcal{SOOL}$ and to show that $\mathcal{SOOL}$ is statically type safe with respect to that semantics. (See [BSvG95, BFSvG01] for such a proof for a similar language.) However, that would take us too far afield, so we will be content with this more indirect proof of the static type safety of $\mathcal{SOOL}$.

### 13.4 Errors

We have seen that a static type system like that used for $\mathcal{SOOL}$ can guarantee that there are no run-time type errors. In particular, if a message is sent to an object, then that object has a method that can handle the message. Ideally, a static type system could provide an iron-clad guarantee that no run-time errors would occur at all, but this is too much to hope for. For example, no type system could guarantee that program logic is correct. However, without severely restricting the expressiveness of the language, it is even impossible to guarantee that errors like dividing by zero will not occur at run time.

The difficulty is that certain run-time errors are not associated with type systems. For example, if $m$ and $n$ are expressions of type $\text{Integer}$, the expression $m/n$ is well-typed, but if the value of $n$ at run time is 0, an error will occur. Because both zero and non-zero integers are in the same type, there is no way of detecting this problem with a type system.

Of course, one could refine the type system to include a type of non-zero integers. The type system could then require $n$ to have type non-zero integer in order for $m/n$ to be well-typed. This is fine in theory, but in practice this does not work out well at all. Using results in the theory of computation (e.g., see Lewis and Papadimitriou[LP98]), it can be shown that the problem of determining whether an integer-valued expression ever takes the value of zero is undecidable. Thus no algorithm can reliably determine whether or not such an expression can ever evaluate to zero. Because we desire a decidable static type system, no static type system can determine, for an arbitrary integer-valued expression, whether that expression will always result in a non-zero value.

In practice, this means that the programmer will have to insert run-time checks in order to convert the results of an arbitrary arithmetic expression
into a type consisting only of non-zero integers. At this point it is not at all clear how much benefit this type system providing non-zero integers from integers is providing, as it will likely be more straight-forward just to insert run-time checks (perhaps using exceptions and their handlers) to ensure that divisions by zero do not occur.

A similar problem occurs in languages with references in dealing with null references. Evaluating val null results in a run-time error. This is an issue with the polymorphic lambda calculus, which includes a keyword, null, which is an element of all reference types. However, S00L does not support explicit references, so it is not an issue. A more relevant problem for object-oriented languages is sending messages to nil.

The expression nil is a member of all object types. We could create object types that also exclude nil (like our non-zero integer type above) but we would run into similar difficulties as many classes, such as nodes for linked lists, have instance variables (e.g., next) that need to hold object values that may include nil. Certainly it is possible to program around these difficulties, but we adopt the attitude of most popular languages, that expressions of object types may hold the value of nil.

We have specifically postponed dealing with the translation of nil, partially because of the complications that arise in introducing run-time tests before message sending. We will introduce several alternatives for dealing with this problem in Section 14.2 of the next chapter.

How do we deal with these errors that can generally not be detected statically? They should be detected at run time and result in either aborting a computation or they should raise exceptions that can be caught and handled by the program. We will not discuss such exception handling here. Instead we will require that errors not detected when the program is type checked should result in an error expression of the appropriate type.

Because we have not detailed many of the basic operations of S00L, we have not included integer division or arrays. However, we will include a check for non-nil recipient in the revised definition of message send in Figure 14.5 of Section 14.2 of the next chapter. If the message is sent to a value representing nil, then an error expression of the appropriate type is returned. To complete the treatment of such run-time errors we would need to ensure that computation rules are strict in error expressions. This means that an operation which involves an expression whose value is an error should evaluate to an error expression of the appropriate type.

We did not provide a direct operational semantics of S00L here. Instead we provided a translational semantics into \( \Lambda_{\text{C}} \), and we have assumed a safe
operational semantics of $\Lambda^p_<$ exists. In order to handle errors we would have to ensure that there is an error expression for all types of $\Lambda^p_<$, and that computations involving errors should result in error values.\(^2\) For simplicity in writing our semantics, we ignore the problem of propagating errors in this text.

13.5 Summary

In this chapter we showed that the translation of $\textit{SOOL}$ into $\Lambda^p_<$ preserves both subtyping and types of expressions. Given that the type-checking rules of $\Lambda^p_<$ are well-understood, the translation into $\Lambda^p_<$ provides an explanation of $\textit{SOOL}$’s type-checking rules as well as providing an actual proof of type safety. This is quite an achievement – we have actually proved (modulo our assumption of the type safety of $\Lambda^p_<$) that no run-time type errors (e.g., “message not understood” errors) can arise with $\textit{SOOL}$ programs. Few existing programming languages have such proofs, partially because most languages do not have formal specifications of type-checking rules and semantics (aside from the implicit specifications encoded in compilers for the languages), and partially because real languages are quite complex. Nevertheless, having accomplished this proof we can be confident that the core of object-oriented languages with similar type systems are type safe.

However, in order to simplify our presentation to this point, we have ignored some important constructs in object-oriented languages. In the next chapter we will deal with some of these complicating factors.

\(^2\) The one exception is that we should ensure that if-then-else statements are evaluated lazily so that if, for example, evaluating the else branch would result in an error, but the boolean condition results in true, then the else branch is ignored. A proper semantics of $\Lambda^p_<$ would ensure that this happens, regardless of whether the ignored branch evaluates to an error or not, so this is usually not a concern.
Completing $SOOL$: super, nil, Information Hiding, and Multiple Inheritance

In this chapter we complete the translation of $SOOL$ by handling some details that were ignored up to this point in order to simplify the presentation. In particular we discuss the handling of calls to methods in the superclass, the handling of nil, and a problem that arises with returning self from a method. The latter problem will suggest extensions to the language to be discussed in detail in Chapter 16.

In the last few sections of this chapter we describe the semantics of information hiding and multiple inheritance. We include a brief discussion of whether these features are valuable or necessary in an object-oriented language.

14.1 Using methods from superclasses

A common practice in object-oriented languages is to call the version of a method from the superclass when redefining that method. So far we have not explained how to do this, but only a simple extension of our language and translation is needed to handle calls to superclass methods. While it is more common to write a reference to the superclass method as super $m$, we will find it technically easier to treat super as a record of methods for the purposes of type-checking.

A revised version of the subclass type-checking rule extended to handle super is contained in Figure 14.1. The only change from the previous version in Figure 12.1 is that $E'$ now contains a declaration of the keyword super as a record with the type of the record of methods of the superclass.

Figure 14.2 contains the translation of subclasses with super. The only
Completing SOQL: super, nil, Information Hiding, and Multiple Inheritance

\[
\begin{array}{c}
C,\mathcal{E} \vdash C::\text{ClassType}(IV_{sup}, M_{sup}), \\
\text{Subclass} \quad C,\mathcal{E} \vdash \text{inst}::IV_{sub}, \quad C',\mathcal{E}' \vdash \text{meth}::M_{sub}, \\
C' \vdash T'_j <:: T_i \text{ for } 1 \leq j \leq m \\
C,\mathcal{E} \vdash \text{class inherits } C \text{ modifies } l_{i_1}, \ldots, l_{i_m} \\
\{\text{inst},\text{meth}::\text{ClassType}(IV, M)\}
\end{array}
\]

where

- \( IV = IV_{sup} \oplus IV_{sub} \),
- \( M = M_{sup} \oplus M_{sub} \),
- there is no overlap in the labels occurring in \( IV_{sup} \) and \( IV_{sub} \), and
- the overlapping labels in \( M_{sup} \) and \( M_{sub} \) are exactly \( l_{i_1}, \ldots, l_{i_m} \).
- The type of \( l_{i_j} \) in \( M_{sub} \) is \( T'_j \), while the corresponding type in \( M_{sup} \) is \( T_i \).
- \( C' = C \cup \{ M <:: IVR' <:: IV'' \}, \quad \text{SelfType} = \text{VisObjectType}(IVR', M') \}, \text{and} \)
- \( \mathcal{E}' = \mathcal{E} \cup \{ \text{self}::\text{SelfType}, \text{close}::\text{SelfType} \rightarrow \text{ObjectType} M', \text{super}::M_{sup} \} \)
- \( \text{SelfType does not occur in IV or M} \).

**Figure 14.1** Type-checking rules for subclasses with super.

difference from the previous translation is that we interpret the keyword super as methfun_{sup}(self). That is, super is interpreted as the record of methods from the superclass in which self is interpreted in the same way as the subclass.

We illustrate this in Figure 14.3 with an example. Suppose clrCell is an object generated from C1rCellClass, and we wish to evaluate clrCell \( \triangleright \text{bump}() \).

Evaluation of bump() in C1rCellClass involves a call to the method in the superclass, CellClass, because the method was not overridden in C1rCellClass. The evaluation of bump in the superclass results in sending the message set to self.

An important issue is whether the set code for the superclass CellClass
\[ \mathcal{T}_C[T, E] \vdash \text{class inherits } C \text{ modifies } l_1, \ldots, l_m \langle \text{inst}, \text{meth} \rangle : \]

\[ \text{ClassType}(IV, M) \overset{\Delta}{=} \lambda (M' <: \mathcal{T}_C[M]). \lambda (IVR' <: \mathcal{T}_C[IVR'])(\langle \text{inst}_all, \text{methfun}_all \rangle) , \]

where

\[ c = \mathcal{T}_C[T, E] \vdash C : \text{ClassType}(IV_{sup}, M_{sup}) \] \[ \mathcal{M}[M'] \] \[ \mathcal{M}[IVR'] \]

\[ \text{inst}_{sup} = \text{proj}_1(c) \]

\[ \text{methfun}_{sup} = \text{proj}_2(c) \]

\[ \text{inst}_all = \text{inst}_{sup} \uplus \mathcal{T}_C[T, E \vdash \text{inst}: IV_{sub}] \]

\[ \text{methfun}_{all} = \lambda (\text{self}: \text{VisObj}(IVR', M')). \]

\[ \text{let close: VisObj}(IVR', M') \rightarrow (\exists Y. \text{VisObj}(Y, M')) \]

\[ = \text{closeobj}[IVR'][M'], \]

\[ \text{super: } \mathcal{T}_C[M_{sup}] = \text{methfun}_{sup} \text{(self)} \]

\[ \text{in } \text{methfun}_{sup} \text{(self)} \uplus \mathcal{T}_C[T, E \vdash \text{meth}: M_{sub}] \]

**Figure 14.2** Translation of subclass with super from S\text{OO}L to L\text{L}_\text{E}.

will be executed or the set code for ClrCellClass. In object-oriented languages, self represents clrCell, the receiver of the message, so the selection of appropriate method body should be based on the class of clrCell, ClrCellClass. Thus the evaluation of \textbf{bump()} in ClrCellClass should result in increasing the value of x by 1 and changing the color to \textit{red} (the result of executing the code for \textbf{set} in the subclass).

The encoding of \textbf{super} supports that choice because it is interpreted by \textbf{methfun}_{sup}(\text{self}). That is, the interpretation of \textbf{self} inside the methods of \textbf{super} is the same as it is in the methods of the subclass.

**Exercise 14.1.1** To make the translation more concrete, write out carefully the encoding of

\[ \langle \text{new ClrCellClass} \rangle \Leftarrow \text{bump()} \]

in order to see that the result ends up increasing the value of x by 1 and setting color to \textit{red}.

**Exercise 14.1.2** What would be the translation of \textbf{super} \Leftarrow m rather than \textbf{super}.m? Is there a way of changing the translation of \textbf{super} so that the first statement gives the desired results?
class CellClass {
    x: Integer = 0;

    function get(): Integer is
        return x
    
    function set(nuVal: Integer): Void is
        self.x := nuVal
    
    function bump(): Void is
        self := set(get()+1)
    }

class ClrCellClass inherits CellClass modifies set {
    color: ColorType = blue;

    function getColor(): ColorType is
        return self.color
    
    function set(nuVal: int): Void is
        super.set(nuVal);
        self.color := red
    }

Figure 14.3 Illustration of the use of super in a subclass.

It is straightforward to extend the proof of soundness to include the language with super. Only the subclass case needs to be modified. It is sufficient (and easy) to make sure that the type of \( T_c[\text{super}] \) is \( T_c[M_{sup}] \).

14.2 Translating nil

So far we have avoided considering how to interpret the expression nil of Sool. As discussed in Chapter 10, nil's type will depend on its context. This leads us to consider whether we should
14.2 Translating nil

- interpret all nil’s as the same value, a value which is included in all object types, or
- interpret nil differently depending on its type, while ensuring that the values at different types are related.

Even once we have made a choice between these two alternatives, there are several ways of encoding nil. These include

1. Interpret all objects as references and interpret nil as a nil reference.
2. Interpret nil as the only element in a type Bottom which is a subtype of all object types.
3. Interpret all object types as sum types where one summand is a single element type (e.g., Void) and the other summand is our previous definition of object types.
4. Omit nil from SOOL.

The first two of these interpret all nil’s as the same value, the third interprets them all differently, and the last attempts to avoid the issue altogether.

For the first suggested solution above, reinterpret \( T_c[\text{ObjectType} \times M] \) as

\[ \text{Ref } \exists Y. Y \times (Y \rightarrow T_c[M]) \]

This translation is the closest to the implementation of objects in existing object-oriented languages. It has the advantage of providing a mechanism for explaining how subtyping works in a memory model like that in existing computers.

For example, suppose class PointClass has instance variables \( x \) and \( y \) which hold integers, and class ColorPointClass inherits from PointClass by adding a new instance variable holding a color as well as adding new methods.

The representation of an object in existing languages typically includes a record which contains both the current values of instance variables and information sufficient to locate the code for all methods of the class (often a pointer to a method table for the class). Thus the representation of an object generated from ColorPointClass takes more space than an object generated from PointClass because it needs extra space for a color instance variable. How then can a variable of type PointType hold both points and color points? The solution generally is that objects are held as references to
Completing SOCL: super, nil, Information Hiding, and Multiple Inheritance

a record or table containing the desired information. While the tables take up different amounts of space, the references can fit in the same amount of memory. In this case it is trivial to represent nil as the null pointer.

A detailed translation based on this alternative is presented in Figure 14.4. Unfortunately this alternative has a fatal flaw: it does not preserve subtypes! As noted earlier, reference types do not have subtypes. Thus Ref T <: Ref U only if T = U. As a result, if we wish our translation to preserve subtyping, the interpretations of object types can never be references. Thus we must (reluctantly) rule out this solution.

The difficulty arises here because of the possibility of assigning to a reference. No translation of a SOCL expression will result in making an assignment to such a reference, as objects are represented as implicit references as in Java. This problem does arise in C++, where one can update such a reference – hence the failure of subtyping in C++ when directly manipulating an object rather than its reference.

We could work around this problem by creating a form of reference that may not be assigned to, but we prefer a cleaner solution for our translational semantics. Another alternative [BTCGS89] is to translate to a language without subtyping, but instead insert explicit coercion functions to take care of subsumption.

In the second solution we create a new artificial type Bottom, which is a subtype of all object types and contains only the single element nil. By subtyping, nil is an element of all other object types. An advantage of this approach is that we would not have to change the translations of any of the existing types and expressions. However, the existence of a Bottom type that is a subtype of all other object types (and only object types) is not very natural. The interpretation of Bottom in \( \Lambda^C \) would have to be a subtype of the interpretations of all object types. Do the interpretations of all existential types in \( \Lambda^C \) have the interpretation of Bottom as a subtype or only some? Why only existential types? This solution seems to raise serious problems with a translational semantics to \( \Lambda^C \).

In the third solution, presented in Figure 14.5, all object types are interpreted as sum types. For example, we could define \( T_c[\text{ObjectType}] \) to be Void + \( \exists Y. Y \times (Y \to T_c[M]) \). The interpretation of nil is the element null obtained by injecting \( \langle \rangle \) into the first summand. To access an object we would first have to check to see if it is null. If not, it could be extracted from the second summand and used as before.

This is a more abstract representation than that represented by the first solution, giving the language implementor more choices in how to represent
14.2 Translating \textit{nil}

\[ T_c[\text{ObjectType } M] \triangleq \text{Ref } \exists Y. Y \times (Y \rightarrow T_c[M]). \]

\[ T_c[C, E \vdash E \leftarrow \text{m_\text{null}} \text{ open var } T_c[E] \text{ as } \langle IR, vo \rangle \text{ in } (\text{proj}_2(\text{vo})(\text{proj}_1(\text{vo}))).m \text{ if } C, E \vdash E: \{ \ldots, m: U, \ldots \}. \]

\[ T_c[C, E \vdash \text{new } C: \text{ObjectType } M \triangleq \text{ref closeobj } [IVR][M](\text{inst}_r, \text{methfun}') \]

where

\[ c \triangleq T_c[C, E \vdash C: \text{ClassType } (IV, M)] [M] [IVR] \]

\[ IVR = T_c[IV_r] \]

\[ M = T_c[M] \]

\[ \text{inst} = \text{proj}_1(c), \]

\[ \text{methfun} = \text{proj}_4(c), \]

\[ \text{inst}_r \text{ is formed by adding refs to all fields of } \text{inst}, \text{ and} \]

\[ \text{methfun}' = \text{fix } [IVR \rightarrow M] \lambda fm: IVR \rightarrow M. \lambda(\text{inst}'.IVR).\text{methfun}(\langle \text{inst}'.fm\rangle). \]

\[ T_c[C, E \vdash \text{nil}: \text{ObjectType } M] \triangleq \text{null}. \]

\textbf{Figure 14.4} Alternative 1 for supporting \textit{nil}: Object types as references.

objects in the computer. Moreover, it makes it clearer that when sending a message to an element of object type, it is necessary to check to see that the element is non-null before attempting to access the methods or instance variables. The problem remains, however, of deciding what to do when a message is sent to an object that is the interpretation of \textit{null}. In the figure, we interpret this by returning an \textit{error} value. We refer the reader back to the discussion in Section 13.4 on the necessity of introducing error expressions in every type and ensuring that errors are propagated during the evaluation of expressions.

The fourth alternative is tempting. If \textit{ObjectType } \textit{M} is an object type for which we would like to have an object like \textit{nil}, we can define a class implementing \textit{ObjectType } \textit{M} which has that behavior. For example, suppose we have an object type representing a node with a link to another node, and suppose the method \textit{getNext} returns the adjacent node. We can create a class \textit{NilNode} in which the \textit{getNext} method of an object always returns itself. We could attempt to test for the end of a list by checking to see if
$\mathcal{T}_C[\text{ObjectType } M] \triangleq \text{Void } + \exists Y. Y \times (Y \rightarrow \mathcal{T}_C[Y]).$

$\mathcal{T}_C[C, E \vdash E \leftarrow m : U] \triangleq \text{case } \mathcal{T}_C[C, E \vdash E : [\\ldots, m : U, \ldots]] \text{ of }$
$x_1 : \text{Void then error}_U ||$
$x_2 : \exists Y. Y \times (Y \rightarrow \mathcal{T}_C[Y]) \rightarrow \mathcal{T}_C[Y].$ then open $x_2$ as $\langle IR, vo \rangle$ in
$(\text{proj}_2(\text{vo})(\text{proj}_1(\text{vo}))).m$

where
$U = \mathcal{T}_C[U].$

$\mathcal{T}_C[C, E \vdash \text{new } C : \text{ObjectType } M] \triangleq$
$\text{in}_2^{\text{Void}, U}(\text{closeobj}(\mathcal{T}_C[\text{IV}^o])\mathcal{T}_C[M](\text{instref}, \text{methfun}').)$

where
$U = \mathcal{T}_C[\text{ObjectType } M]$
$\text{obj} = \mathcal{T}_C[C, E \vdash C : \text{ClassType}(\text{IV}, M)\mathcal{T}_C[M]\mathcal{T}_C[\text{IV}^o]]$
$\text{inst} = \text{proj}_1(\text{obj}),$
$\text{methfun} = \text{proj}_2(\text{obj}),$
$\text{inst}^o$ is formed by adding refs to all values of fields of $\text{inst}$, and
$\text{methfun}' = \text{fix}(\mathcal{T}_C[\text{IV}^o] \rightarrow \mathcal{T}_C[M]) \lambda fm: \mathcal{T}_C[\text{IV}^o] \rightarrow \mathcal{T}_C[M].$
$\lambda (\text{inst'}: \mathcal{T}_C[\text{IV}^o]).\text{methfun}(\text{inst'}, fm).$

$\mathcal{T}_C[C, E \vdash \text{nil} : \text{ObjectType } M] \triangleq \text{in}_1^{\text{Void}, U}(\langle \rangle)$

where $U = \text{ObjectType } M$ and $U = \mathcal{T}_C[U].$

Figure 14.5 Alternative 3 for supporting $\text{nil}$: Object types as sums.

node.getNext() = node, but a real node might need to be allowed to point to itself in some applications, e.g., circular lists. It makes more sense to add a method called $\text{isNil}$ to all node classes that returns true only if the object is generated by $\text{NilNode}$. Other methods can be defined to always return an error value or raise an exception when invoked. Many other variations of this technique are also possible, but they all tend to be awkward in practice.

If we wish to present a translational semantics into $\Lambda_C^{\infty}$, we find ourselves having to choose between the third (sum semantics) and fourth alternatives (eliminate $\text{nil}$ altogether). While the last alternative is tempting in that it allows us to ignore $\text{nil}$, existing object-oriented languages do provide for $\text{nil}$. As a result we find the representation of object types as sums as the most attractive.
14.3 A complication with \texttt{self}

There is one complication with \texttt{self} that we have only dealt with indirectly. Suppose a class has a method that returns \texttt{self}. In type-checking methods of a class with type \texttt{ClassType(IV, M)}, we assumed that the type of \texttt{self} was \texttt{SelfType = VisObjectType(IVR', M')}, where \texttt{IVR' <: IVrf} and \texttt{M' <: M}. Thus if a method returns \texttt{self}, then it returns an element of type \texttt{SelfType}. We introduced the \texttt{close} function, with type \texttt{SelfType \rightarrow ObjectType M'}, in order to solve that problem. We said earlier that rather than returning \texttt{self}, we will instead return \texttt{close(self)}, which has type \texttt{ObjectType M'}. However, it makes no sense to write \texttt{ObjectType M'} as the return type of the method, as this will mean nothing to the user. In particular, the class type \texttt{ClassType (IV, M)} may not mention \texttt{M'} because it is only introduced as a type variable in \texttt{C} in order to write down the type-checking rules. It never appears in the source language!

Luckily, this can be taken care of by using the fact that if \texttt{M' <: M}, then \texttt{ObjectType M' <: ObjectType M} by the subtyping rules of \texttt{SOOL}. Thus the type of \texttt{close(self)} can be promoted by the \texttt{Subsumption} rule to type \texttt{ObjectType M}. Hence the return type of a method returning \texttt{self} in a class with type \texttt{ClassType(IV, M)} can be given as \texttt{ObjectType M} (first by replacing \texttt{self} by \texttt{close(self)} and then by promoting the type using subsumption). Similarly, \texttt{close(self)} can be used as an actual parameter whenever the corresponding formal parameter has type \texttt{ObjectType M}.

This solves our problem, though at the cost of losing type information. Here is an example. Let \texttt{C} be a class with type \texttt{ClassType(IV, M)} that has a method \texttt{m} that returns \texttt{self}. As noted above, the return type of \texttt{m} will be \texttt{ObjectType M}, the type of objects generated by \texttt{C}. Now suppose that \texttt{C'} is a subclass of \texttt{C} with type \texttt{ClassType(IV_{all}, M_{all})} where \texttt{IV_{all}rf} <: \texttt{IVrf} and \texttt{M_{all} <: M}. Assume also that \texttt{C} does not override method \texttt{m}. Then the type checker will assume the return type of the inherited method \texttt{m} will also be \texttt{ObjectType M}, even though it were overridden with exactly the same method body, it would type check as having return type \texttt{ObjectType M_{all}}.

In actuality, of course, this inherited method would return an object with
run-time type \texttt{ObjectType}_\texttt{null}, but the \texttt{SOOL} type system is not expressive enough to recognize this.

This loss of information is suggestive of that found in Java's type system, where method \texttt{clone} is declared in \texttt{Object} with return type \texttt{Object}. As a result, in all subclasses, \texttt{clone} also must return \texttt{Object}. In our encoding, as in C++, we can actually do a bit better. If the user is willing to override \texttt{clone} in a class, we can change its return type to be the object type corresponding to the class. However, inherited methods never have their types automatically modified. We will return later to this problem and see how we can retain more exact type information by using a \texttt{MyType} construct.

### 14.4 Finer control over information hiding

The typing rules for \texttt{SOOL} provide only very simple support for information hiding. Instance variables are always hidden from outside the object, but are available to subclasses (similarly to "protected" in Java and C++), while methods are always available outside of the object as well as to all subclasses (corresponding to "public" in Java and C++).

It is easy to make selected instance variables public. Simply provide default "setter" and "getter" methods for the instance variable (e.g., \texttt{setX} and \texttt{getX}, if \texttt{x} is an instance variable), treat an access to an instance variable as syntactic sugar for the corresponding getter method, and treat updates to an instance variable as syntactic sugar for the corresponding setter method. Of course these can be optimized in the translation to provide direct access to the appropriate instance variables, but logically providing these methods is equivalent to making the variables themselves public.

Providing more restricted access to instance variables and methods takes more work (and more complicated rules – at least notationally), but is relatively straightforward. We sketch the changes required below. Rather than use the names \texttt{public}, \texttt{protected}, and \texttt{private} from Java and C++, we adopt the new names \texttt{visible}, \texttt{hidden}, and \texttt{secret} to have similar, although slightly different, meanings.

\textbf{Visible Methods}  
Methods that are declared as \texttt{visible} methods have the same visibility as methods treated previously in this text. They are visible from other methods in the class, from methods in any subclasses, and are visible from the outside of objects generated from the class. They are equivalent to the "public" methods of Java and C++.

\textbf{Hidden Methods}  
Methods that are declared as \texttt{hidden} methods are visible from other meth-
14.4 Finer control over information hiding

ods in the class and from methods in any subclasses. However they are not visible from the outside of objects generated from the class. That is, if \( m \) is a hidden method of class \( C \), and \( o \) is an object generated from \( C \), then one may not send the message \( m \) to \( o \). (Though one can send the message \( m \) to \( self \) from methods in \( C \) or any of its subclasses.)

This is similar to the use of protected methods in Java and C++.\(^1\) It differs most strikingly from the notion of protected in Java and C++ in the following situation. Let \( m(C\ other) \) be a protected method of \( C \) that takes a parameter from the same class. In Java and C++, the method body of \( m \) has access to both protected and private methods and instance variables of \( other \). In the model we are describing, classes may not be used as the types of parameters. Thus it is impossible to get access to hidden methods of parameters, even if their type is the same as the type of objects generated by the class. (This should make sense as, even though the parameter has the same type, it may have been generated by a different class.)

Finally, methods that are declared as secret methods are only visible from other methods in the same class. They are not visible in subclasses nor from the outside of objects generated from the class. This is similar to the private declaration in C++ and Java. (We do not discuss secret instance variables, but they can be handled similarly to secret methods.)

---

\(^1\) For simplicity, we ignore package visibility in Java. Instead we presume that each class is in a separate package so that package scope is not an issue.
We write class(inst, smeth, hmeth, vmeth) to represent a class with (hidden) instance variables inst, secret methods smeth, hidden methods hmeth, and visible methods vmeth. All of the methods are type checked under the same assumptions. (See Figure 14.6.) The class type includes the type of the record of hidden methods, HM, and the type of the record of visible methods, VM, though not that of the secret methods. The function close takes SelfType to ObjectType VM, reflecting the fact that the hidden methods are no longer visible in the type of objects generated from the class.

While we don’t bother to write the type-checking rule for subclasses, the modifications necessary from our earlier rule should be obvious from the changes made to the type-checking rule for classes.

The type-checking rule for new expressions given in Figure 14.6 makes it clear that when an object is created from a class, the resulting object’s type no longer includes the types of hidden methods. Because the type-checking rules for message sending stay the same, only messages corresponding to the visible methods can be sent to an object.

The semantics of classes change only in relatively insignificant ways to reflect hidden methods, while more work is involved in creating new objects from classes. The new translations are given in Figure 14.7. These translations use the following abbreviations:

\[
\begin{align*}
\text{VisObj} & \triangleq \text{IVR} \times (\text{IVR} \to \text{SM} \times \text{HM} \times \text{VM}) \\
\text{Obj} & \triangleq \exists\text{HM}. \exists\text{SM}. \exists\text{Y}. \text{VisObj}(\text{Y}, \text{SM}, \text{HM}, \text{VM}) \\
\text{closeobj} & \triangleq \forall\text{IVR}. \forall\text{SM}. \forall\text{HM}. \forall\text{VM}. \lambda (\text{vo} : \text{IVR} \to (\text{IVR} \to \text{SM} \times \text{HM} \times \text{VM})). \\
& \quad \text{pack} \langle \text{HM}, \text{ho} \rangle \text{ as } \text{Obj} (\text{VM}) \\
\text{where} \\
& \quad \text{ho} \triangleq \text{pack} \langle \text{SM}, \text{so} \rangle \text{ as } \exists\text{SM}. \exists\text{Y}. \text{VisObj}(\text{Y}, \text{SM}, \text{HM}, \text{VM}) \\
& \quad \text{so} \triangleq \text{pack} \langle \text{IVR}, \text{vo} \rangle \text{ as } \exists\text{Y}. \text{VisObj}(\text{Y}, \text{SM}, \text{HM}, \text{VM}) \\
\end{align*}
\]

The translation of visible objects is a straightforward generalization of the earlier definition. The method portion is represented by a function that takes the instance variables as a parameter and returns a triple consisting of the secret, hidden, and visible methods. The translation of objects differs from the earlier translations by hiding the types of secret and hidden methods with existential quantifiers. Thus we can see that an object type has hidden instance variables and secret and hidden methods, but we do not know any-
thing about their names or types because of the use of existential quantifiers to hide them.

The actual translations of class and new expressions shown in Figure 14.7 are straightforward generalizations of the earlier translations. The function closeobj takes care of the work of hiding the instance variables and hidden methods in the construction of new objects.

14.5 Multiple inheritance

A language supports multiple inheritance if a class may directly inherit from two or more superclasses. A common example is defining a class for teaching assistant as being a subclass of both a student class and a teacher class.

The opportunity to inherit from two or more superclasses is often seen as reflecting the fact that objects may be looked at as members of two or more categories, as in the example above of teaching assistant. From the point of view of types, this specification can be understood in a very straightforward way by defining a type TAType which is a subtype of both StudentType and TeacherType. The classification of TA’s as both student and teacher indicates that they can respond to messages sent to either students or teachers. This classification need not have anything to do with inherited instance variables or methods, though it may be convenient to implement in this way.

Perhaps surprisingly, it is not difficult to model multiple inheritance of classes in the translational semantics presented here, once one decides what is intended. Moreover, the type rules are relatively straightforward. The only difficulty lies in determining what the intended semantics are when there are conflicts that arise in inheriting from two or more classes.

A classic problem is illustrated with the class hierarchy drawn in Figure 14.8. Let C be a class with a method m. Let subclasses D1 and D2 be subclasses of C, each of which overrides method m. Now let E be a subclass of both D1 and D2. Which method m should E inherit? Similar and perhaps even more difficult issues arise when instance variables with the same name are inherited through multiple superclasses.

Different languages resolve this difficulty in different ways. The two main strategies are replicated inheritance and shared inheritance. With replicated inheritance, a subclass inherits all copies of instance variables and methods. Suppose that in the above example both D1 and D2 include an instance variable x. With replicated inheritance, subclass E would contain both instance variables, whether or not they had the same type. Methods inherited from D1
\[ T_c[\text{VisObjectType}(\text{IVR}, SM, HM, VM)] \triangleq \]
\[ \text{VisObj}(T_c[\text{IVR}], T_c[SM], T_c[HM], T_c[VM]) \]

\[ T_c[\text{ObjectType VM}] \triangleq \text{Obj}(T_c[VM]) \]

\[ T_c[\text{ClassType}(IV, HM, VM)] \triangleq \]
\[ \forall (VM' <: T_c[VM]). \forall (HM' <: T_c[HM]). \forall (IVR' <: \text{IVr}'). \exists SM. \]
\[ T_c[IV] \times (\text{VisObj}(IVR', SM, HM', VM') \to SM \times T_c[HM] \times T_c[VM]) \]

\[ T_c[C, E] \vdash \text{class}(\text{inst}, \text{smeth}, \text{hmeth}, \text{vmeth}): \text{ClassType}(IV, HM, VM)] \]
\[ \triangleq \lambda (VM' <: T_c[VM]). \lambda (HM' <: T_c[HM]). \lambda (IVR' <: \text{IVr}'). \]
\[ \text{pack}(\langle SM, \langle T_c[C, E] \vdash \text{inst}: IV \rangle, \text{methfun} \rangle) \ as \]
\[ \exists SM. T_c[IV] \times (VO \to SM \times T_c[HM] \times T_c[VM]) \]

where
\[ C' = C \cup \{ HM' <: HM, VM' <: VM, IVR' <: \text{IVr}' \}, \]
\[ \text{SelfType} = \text{VisObjectType}(\text{IVR}', SM, HM', VM'), \]
\[ SM = T_c[SM], \]
\[ VO = \text{VisObj}(IVR', SM, HM', VM'), \]
\[ \text{and} \]
\[ \text{methfun} = \lambda (\text{self}; VO). \text{let} \]
\[ \text{close}: VO \to \text{Obj}(VM') = \text{closeobj}[IVR'][SM][HM'][VM'] \]
\[ \text{in} \langle T_c[C', E'], \vdash \text{smeth}: SM \rangle, T_c[C', E', \vdash \text{hmeth}: HM], \]
\[ T_c[C', E', \vdash \text{vmeth}: VM] \rangle, \]

\[ T_c[C, E] \vdash \text{new E: ObjectType VM} \] let
\[ \text{protoobj} = \]
\[ T_c[C, E] \vdash C: \text{ClassType}(IV, HM, VM)] \]
\[ [T_c[VM]] \times (T_c[HM] \times T_c[IVr']) \]

open protoobj as (SM, visobj) in let
\[ \text{inst} = \text{proj}(visobj), \]
\[ \text{methfun} = \text{proj}(visobj), \]
\[ \text{instr} \text{is formed by adding refs to all values of fields of inst, and} \]
\[ \text{methfun'} = \text{fix}(T_c[IVr']). \to SM \times T_c[HM] \times T_c[VM] \]
\[ \lambda (\text{fm}: T_c[IVr']) \to SM \times T_c[HM] \times T_c[VM]. \]
\[ \lambda (\text{inst}: T_c[IVr']). \text{methfun}(\text{instr}, \text{fm}). \]
\[ \text{in obj} = \text{closeobj}[IVR'][SM][HM'][VM](\text{instr}, \text{methfun'}) \]

in obj

Figure 14.7 Translation semantics for classes and objects and their types with hidden and visible methods.
that reference $x$ would reference the copy inherited from $D_1$, while those inherited from $D_2$ would reference the second copy. Changes to one would not affect the value of the other. Of course some mechanism must be provided to indicate which instance variable is intended when writing new code in $E$ that accesses one of these.

With shared inheritance, only one copy of a method or instance variable is inherited into the subclass. If $x$ has the same type in $D_1$ and $D_2$, only one copy is inherited in $E$. Thus methods inherited from $D_1$ and $D_2$ that reference their respective instance variables named $x$ would end up referring to the same instance variable in $E$. (If the types of the instance variables from the superclasses are distinct then two copies would be inherited if the language supported overloading of instance variable names, and would generate a type error and be disallowed otherwise.)

The default in Eiffel is shared inheritance for circumstances like those in our example above, while replicated inheritance is the default in C++. In circumstances where all copies of the multiply inherited method did not arise from the same class, Eiffel requires the programmer to select which $m$ to inherit. Both Eiffel and C++ provide notational mechanisms for achieving the non-default mechanism. Other languages provide complex mechanisms for determining which version of a method is called when a method or instance variable inherited from multiple superclasses is used.

There seems to be a growing consensus that the complications of multiple inheritance are not worth the dangers of ambiguity. For example, it has been observed that most uses of multiple inheritance in C++ arise in contexts in which one of the superclasses is an abstract class. As a result, the designers of Java decided not to support multiple inheritance of classes, but instead allow classes to implement multiple interfaces. (Java interfaces can be used in ways similar to C++’s abstract classes.)
Moreover, Java interfaces can extend multiple interfaces. Because methods are specified only by their signatures in interfaces, this causes no difficulties as long as methods with the same name and parameter types share the same return types. The effect of this is similar to our uses of object types that can match multiple “super” object types.

We specify typing rules and semantics for multiple inheritance. For simplicity, we assume in the rules that there is no overlap in instance variable or method names in the superclasses. If overlap is to be allowed, the rules can be revised in order to reflect the priorities (e.g., first method declaration prevails, both method types must be the same, etc.).

A type-checking rule for subclasses with multiple inheritance is given in Figure 14.9. We write it here only for the case with exactly two superclasses. It is written so as to be a possible replacement for the corresponding rule of SIOCL.

The rule is a very straightforward generalization from our earlier subclass rule. We simply make sure both superclasses are type correct, concatenate corresponding records of instance variables and of method types to form the final type for the subclass, and type check the new instance variables and methods under appropriate assumptions on the type of objects generated from the subclass.

The semantics of multiple inheritance can be given as a minor variant of the rules for single inheritance. Figure 14.10 provides a semantic rule appropriate for extending SIOCL with multiple inheritance.

Because there is no overlap, the translations of the instance variables and methods (and their types) from superclasses can be concatenated with those of the subclass, with priority given to subclass methods over inherited ones from the superclasses. In this way, it is a rather straightforward extension of the earlier translations of subclasses.

We leave it as an exercise for the reader to modify these rules to correspond to the type-checking rules and semantics of any existing language’s rules for multiple inheritance.

The author’s opinion is that the rules necessary for resolving conflicts are more complex than the construct is worth. The default for the languages described in this book is not to support multiple inheritance, but to allow objects from classes to implement any number of types. This allows objects to take on different roles in different contexts, but avoids the complications of resolving ambiguities arising from having multiple features with the same name coming from different superclasses.
14.6 Summary

In this chapter we went back to address details of SCOC and its translation that we had skipped in order not to complicate the discussion of the semantics of classes and objects. We also addressed the translation of more refined rules for information hiding and of multiple inheritance.

The type checking and translation of calls to methods of a superclass were handled in a relatively straight-forward fashion. The translation of \texttt{nil} was a different matter. We presented four options, selecting the option that interpreted objects as elements of a sum type whose alternatives represent the \texttt{nil} element or the translation of objects discussed in earlier chapters.

In the next section we discussed a problem with typing \texttt{self} (or more
\[ T_C \| C, \mathcal{E} \vdash \]

\[
\text{class inherits } E^1, E^2 \text{ modifies } l_{i_1}, \ldots, l_{i_m} \text{(inst,meth)}: \\
\quad \text{ClassType}(IV,M) \triangleq \\
\quad \text{inst}_{all} \text{ modifies } \lambda(M' <: T_C[IVR'] \cdot \lambda(IV' <: T_C[IVR']), \langle \text{inst}_{all}, \text{methfun}_{all} \rangle) \\
\text{where} \\
\quad c^1 = T_C[C, \mathcal{E} \vdash E^1: \text{ClassType}(IV^1_{sup}, M^1_{sup})][M'][IVR'] \\
\quad c^2 = T_C[C, \mathcal{E} \vdash E^2: \text{ClassType}(IV^1_{sup}, M^1_{sup})][M'][IVR'] \\
\quad \text{inst}^1_{sup} = \text{proj}_1(c^1), \\
\quad \text{inst}^2_{sup} = \text{proj}_1(c^2), \\
\quad \text{methfun}^1_{sup} = \text{proj}_2(c^1), \\
\quad \text{methfun}^2_{sup} = \text{proj}_2(c^2), \\
\quad \text{inst}_{all} = \text{inst}^1_{sup} \sqcup \text{inst}^2_{sup} \sqcup T_C[C, \mathcal{E} \vdash \text{inst: IV}_{sub}], \\
\quad \text{methfun}_{all} = \lambda(\text{self: VisObj}(IVR', M')). \\
\quad \text{let close: VisObj}(IVR', M') \rightarrow (\exists Y. \text{VisObj}(Y, M')) \\
\quad = \text{closeobj}[IVR'][M'] \\
\quad \text{in methfun}^1_{sup}(\text{self}) \sqcup \text{methfun}^2_{sup}(\text{self}) \sqcup T_C[C', \mathcal{E}' \vdash \text{meth: M}_{sub}] \\
\quad C' = C \cup \{ M' <: M, IVR' <: IV_{ref}, \\
\quad \text{SelfType} = \text{VisObjectType}(IVR', M') \},
\]

Figure 14.10  Semantics of multiple inheritance in SOOL.

accurately close(self). The difficulty is that the type of close(self) obtained by using the type-checking rules is a type that is an artifact of the type-checking rules, rather than something that would be recognizable to the programmer. As a result, subsumption had to be used to obtain a type that makes sense. Unfortunately, this resulted in a loss of information. We will see later that we can obtain more accurate type information using the MyType construct.

We also discussed more refined information hiding strategies that are similar to those of languages like C++ and Java. While there is no difficulty in extending the encoding in this way, we note that from a language design point of view, there may be more advantages to supporting a module system in object-oriented languages to provide the information hiding. Many systems consist of groups of collections of classes in which those classes within a group may need privileged access to features of other classes in the group, while classes outside of the group should not have such access.
Java’s packages provide some hints on a possible direction for such modules, but only represent a first step. Module systems need also to provide differential access to types (interfaces) as well as class features. Moreover such modules may need to provide different views to different clients. In particular, they may need to provide more access to classes in the module for clients that wish to create subclasses than for clients who wish simply to use objects from the classes provided.

Finally we addressed typing rules and semantics for multiple inheritance. While multiple inheritance seems to be losing its attraction in object-oriented languages, we showed that there are no serious difficulties in presenting type-checking rules or a translational semantics for this construct. Rather, the difficulty is trying to figure out how to disambiguate the situation when multiple features with the same name are being inherited from different superclasses.

The language designs presented to this point have supported only single inheritance, but allow classes to generate objects whose types match those of many object types. In practice this seems to cover most of the desired uses of multiple inheritance. The designers of Java made a similar decision in allowing classes to only directly inherit from one superclass, but to allow them to implement many interfaces.

This completes our discussion of SOOL. In the next section of the book we examine several extensions to the type system of SOOL. Extensions include parametric polymorphism and support for a MyType construct similar to Eiffel’s like Current. Rather than including all of the subtleties discussed in this chapter, we will build from the core SOOL language presented earlier. However, each of these features can easily be added to the languages to be presented later using the same ideas discussed here.
This page intentionally left blank
Inspired by the work by Cardelli [Car88], several authors in the late 1980’s designed semantics of inheritance for untyped languages. These were presented in papers by Kamin [Kam88], Reddy [Red88], and Cook and Palsberg [Coo89a, CP94] (originally presented in 1989). The Reddy and Cook-Palsberg definitions are based on a denotation semantics in which the meaning of self is determined by a fixed point, while the Kamin definition was more operational and based on self-application. The paper, “Two semantic models of object-oriented languages” [KR94], by Kamin and Reddy compares these models and shows their equivalence.

In 1990 the Abel group [CHC90] and Mitchell [Mit90] independently presented semantics for typed object-oriented languages. The Abel group’s semantics relied on fixed points, while Mitchell’s model was operational and based on self-application. The Abel group’s semantics of classes was defined using bounded polymorphism similar to the definition presented here. (They actually used F-bounded polymorphism to handle a MyType construct like that which will be introduced in Chapter 16.) Interestingly the Abel group provided a semantics, but no actual programming language.

The semantics proposed by Mitchell were for a language that supports delegation rather than inheritance using classes. This definition employed higher-order bounded polymorphism, though with negative information – it specified which method names must not occur in a type. Unfortunately the translation of Mitchell’s language into a calculus with records and recursive type definitions did not preserve the subtype relation because each method took self as a parameter. Thus the parameter types of methods changed covariantly when overridden in subclasses, breaking subtyping. Bruce [Bru92]
showed the rough equivalence of these two approaches to the semantics of inheritance.

Many of these and other early papers are collected by Gunter and Mitchell in *Theoretical Aspects of Object-Oriented Languages* [GM94]. That collection includes many of the most important papers on the foundations of object-oriented languages.

Neither of these semantics for typed object-oriented languages handled instance variables. In 1993, Pierce and Turner [PT94] and Bruce [Bru94] separately provided semantics for typed object-oriented languages with instance variables, though technically each was for a functional language whose instance variables were updated by making a new copy of the object.

Pierce and Turner presented their semantics as a translation from a source language to a higher-order polymorphic lambda calculus. Their semantics used existential types for object definitions, essentially as given in this section. The only (inessential) difference is that in their semantics each individual method takes the record of instance variables as a parameter rather than having it be a parameter to the entire record of methods. Because the type of this parameter was hidden with an existential, subtypes were preserved in the semantics.

Because they wished to avoid the use of recursive types, Pierce and Turner considered the record of instance variables (called \texttt{state}) and record of methods (called \texttt{self}) separately, rather than considering \texttt{self} as a whole. Because there is no \texttt{close} function in the source language, it is not easy to define a method that returns an object of the same type as the receiver. Instead, the method returns the record of instance variables of the object, and the code sending the message is responsible for packaging this result into an object using the appropriate method suite. (Hofmann and Pierce [HP95] provided a more abstract approach to modeling these languages.)

Bruce presented his semantics as a denotational semantics of the language TOOPLE in a model of the F-bounded polymorphic lambda calculus. This semantics can be understood as a generalization of the Abel group’s semantics to include the encoding of instance variables using existential types. The semantics of objects involved both existential types and fixed points. As with the Abel group’s semantics, fixed points were required to handle a \texttt{MyType} construct as described in Chapter 16. Similarly, F-bounded polymorphism was required to interpret classes in order to handle \texttt{MyType}.

The semantics of classes and objects provided in this section can be seen to be a hybrid of the Pierce-Turner and Bruce semantics. The translational semantics using only existentials (no fixed points) is clearly quite similar to
Pierce-Turner. The semantics can also be seen as a variant of the semantics of Bruce, modified to be a translational semantics and simplified by the omission of $\text{MyType}$. In particular, the $\text{close}$ construct provided the opportunity to write methods that returned an object of the same type as the receiver rather than just its representation. The concerns expressed in Section 14.3 with the typing of $\text{self}$ will lead to the extension to $\text{MyType}$ in Chapter 16.

Abadi and Cardelli designed an object calculus because of difficulties in finding a translational semantics for their object-based languages in which the translation preserved subtyping. Such a semantics is more difficult because of the possibility of method override in objects (rather than classes). Later, however, in work with Viswanathan [ACV96], they were able to find a translational semantics that preserved subtyping. This semantics is more complex than either the Pierce-Turner or Bruce semantics, with the meaning of object types expressed in terms of both fixed points and bounded existentials.

The paper “Comparing object encodings” [BCP99] by Bruce, Cardelli, and Pierce provides a careful description of these three encodings along with a simple recursive record translation. (Bruce’s semantics is first transformed to be a translation to the polymorphic lambda calculus in this paper.) It also catalogs the advantages and disadvantages of each of the encodings.

Eifrig et al. [ESTZ94b, ESTZ94a] provided one of the first semantics for imperative object-oriented languages, as well as making progress on the difficult issue of type inference [EST95b, EST95a] for object-oriented languages. The languages TOIL [vG93] and PolyTOIL [BSvG95, BFSvG01] are examples of imperative languages provided with semantics like that given here.

Abadi and Cardelli also designed imperative object calculi [AC95a] as a foundation for object-based languages. While their text [AC96] focuses primarily on object-based languages, they also show how to model classes in their object calculus, providing the semantics for quite rich class-based object-oriented languages. There has also been a great deal of work by others on extending their object calculi. We omit it here because our focus is on class-based languages.

Other interesting semantics of typed object-oriented languages were provided by Abadi [Aba94] and Palsberg and Schwartzbach [PS94]. Fisher and Mitchell [FM96] provide an interesting survey of research on type systems for object-oriented languages through the mid-90’s.

The modeling of information hiding given in Section 14.4 does not correspond exactly to the hiding provided by C++ and Java because we do not allow classes to be used as types. In C++ and Java if a method $m$ of class $C$
takes a parameter \( c' \) of class \( C \), then in the body of \( m \) it is possible to access private and protected variables of \( c' \). Fisher and Mitchell [FM98] designed a two phase modeling of classes that provides exactly that access. Their solution involves the use of nested existential types so that subclasses can obtain access to the instance variables of superclasses. The definition involves two kinds of types. Those for the protected interface are modeled by types that support method extension and override, while those for public interfaces only support method invocation. Like Mitchell's earlier work, method invocation is modeled by self-application.

C++ includes support for "friend functions", which allow certain classes to have privileged access to methods and instance variables that are normally hidden. Papers by Pierce and Turner [PT93] and Katiyar et al. [KLM94] propose the use of bounded existential types to model privileged access by keeping the privileged classes behind an abstraction barrier, and only providing general access to methods listed in the bound of the existential.

Modula-3 [CDG+88, CDJ+89] uses a module in order to group items which are intended to be "friends". Partial revelations [Fre95] restrict access to full type information from outside of the module. The partial revelations indicate that a type identifier is an extension of a fixed type, but do not reveal what other information values of that type may contain that is not described by the bound. Bruce and Petersen's paper, "Modules in \( L O O M \): Classes are not enough" [BPV98, BFP97], presents an extension of their language \( L O O M \) with a module structure that supports a generalization of partial revelations along with separate access privileges for subclasses of classes in the module.

Fisher and Reppy's language Moby [FR99] supports both classes and modules, and is provided with a firm theoretical foundation. Their later paper, "Extending Moby with inheritance-based subtyping" points out flaws with modeling friends with partial abstraction and suggests that "inheritance-based subtyping" provides better support for friends, binary methods, and object cloning. The underlying semantics is based on an object calculus similar to that used in [FM98].

In Section 13.2, we noted the difficulties with the coherence of a translation based on derivations of typing judgements. The problem is that two different typing derivations of the same judgement may result in different translations. Curien and Ghelli, in their paper "Coherence of subsumption, minimum typing and type-checking in \( F_{\leq} \)" [CG92] provided a careful analysis of coherence issues in a version of the bounded polymorphic lambda calculus. They provided a rewriting system for typing derivations that provide the basis of a proof that a semantics is well-defined. Mitchell's text
[Mit96] provides a compact discussion of the issues and techniques to show coherence.

A style of writing semantics for object-oriented languages that we have not discussed here is the use of row variables. Row variables were introduced by Wand [Wan87, Wan89] in connection with a type inference scheme for simple objects, and were used in Mitchell’s semantics of typed object-oriented languages [Mit90]. They can be used in specifying partial information about the method suite of an object, but allow further extension. (The version used by Mitchell also allows the specification of negative information.) Row types are especially useful with type inference.

The language Objective ML is the theoretical basis of the language Objective CAML, an extension of ML with classes and objects. The semantics of Objective ML is expressed in terms of row variables. The language supports type inference, but requires explicit coercions to treat an object as having the type of one of its superclasses. The paper, “Objective ML: An effective object-oriented extension to ML” [RV98], describes the language and its semantics, as well as providing a host of references on row variables and type inference in object-oriented languages. Another fairly minimal extension to ML supporting features of object-oriented languages is “Simple Objects for Standard ML” [RR96], by Reppy and Riecke.

There has been a great deal of interest in the semantics of Java and the safety of its type system. Representative papers are by Drossopoulou et al. [DE98, DEK99] and Börger and Schulte [BS98]. The design of Featherweight Java by Igarashi et al. [IPW99] is an important development for language theoreticians wanting to test their language ideas with a simplified version of Java that is amenable to theoretical analysis. The full Java language is extremely complicated, and many of these complications get in the way of careful study of particular language features. Because Featherweight Java has a simple semantics, it is easy to extend it to encompass new features and yet still prove type soundness.
This page intentionally left blank
Part IV

Extending Simple Object-Oriented Languages
15 Adding Bounded Polymorphism to \( \text{SOOL} \)

Adding Bounded Polymorphism of the sort discussed in Section 4.1 to \( \text{SOOL} \) is relatively straightforward, as it is not that different from the addition of parametric polymorphism to the lambda calculus. In this chapter we provide the details of an extension, \( \text{PSOOL} \), of \( \text{SOOL} \). The reader is invited to go back and review Section 4.1 for the motivation to add F-bounded polymorphism.

15.1 Introducing \( \text{PSOOL} \)

We begin by introducing the syntax of type and value expressions in \( \text{PSOOL} \) and then providing type-checking rules. We follow this with a few simple examples of the use of F-bounded polymorphism.

We first add a kind system like that for the polymorphic lambda calculus in Chapter 9. However, for simplicity we restrict the kinds to those that take types as arguments rather than elements of higher kinds. We represent the kind of all types by \( \ast \).

\[
\kappa \in \text{Kind} ::= \ast \mid \ast \Rightarrow \kappa
\]

Type constructors are either types or functions that take a type as a parameter and return a constructor. We write \( U : : \kappa \) to indicate that \( U \) is a type constructor with kind \( \kappa \). Let \( \mathcal{T} \) be a set of type constants containing the type constant \( \text{TopObject} \). Let \( \mathcal{L} \) be a set of record labels, and \( \mathcal{T}_I \) a set of type identifiers. Constructor expressions of \( \text{PSOOL}, \text{CONSR}_{\text{SOOL}}(\mathcal{T} \mathcal{C}, \mathcal{L}, \mathcal{T}_I) \), (with their associated kinds) are defined as follows:

1. If \( c \in \mathcal{T} \mathcal{C} \), then \( c :: \ast \).

2. If \( t \in \mathcal{T}_I \), then \( t :: \ast \).
3. If $T::\tau$ and $U::\kappa$, then $\text{TpFunc}(t).U::\tau \Rightarrow \kappa$.

4. If $T::\tau$ and $F::\tau \Rightarrow \kappa$, then $F(T)::\kappa$.

5. If $T::\tau$ and $U::\tau$, then $\text{ForAll}(t <: T).U::\tau$.

6. ...

The elided elements in the last case consist of all of the type definitions from $SOOL$, all given with kind $\tau$. We omit them here because they would overwhelm the new constructor expressions.

If $T$ has kind $\tau$, then we say that $T$ is a type expression of $\mathcal{P}SOOL$ (written $T \in \mathcal{TYP}_{\mathcal{P}SOOL}(\mathcal{T}, \mathcal{L}, \mathcal{T})$). Notice that the constructor identifiers and constants are only of kind $\tau$ (and hence will be referred to as type identifiers and constants).

Constructor expressions of the form $\text{TpFunc}(t).U$ represent functions taking a type $T$ to type $[T/t]U$. The expression $F(T)$ represents the application of type function $F$ to argument $T$.

If $T$ and $U$ are types, then $\text{ForAll}(t <: T).U$ is the type of polymorphic functions taking subtypes of $T$ to values of type $U$.

While constructors may only take types as arguments, it is possible to obtain type functions taking several type parameters by writing them in a curried fashion, that is, defining a type function that returns another type function. An example is $\text{TpFunc}(t).\text{TpFunc}(u).t \Rightarrow u$.

As in $\lambda^P$, we include in Figure 15.1 a congruence rule, $\text{FuncAppCong}$, for simplifying constructor applications. In combination with the corresponding subtyping and type-checking rules, $\text{Cong}_{\ll}$ and $\text{Congruence}$, this will allow us to simplify constructor applications appearing in expressions to be type-checked. (While we don’t bother to write the corresponding rules, the congruence relation is reflexive, symmetric, and transitive.)

It is important not to confuse polymorphic types – those types with the form $\text{ForAll}(t <: T).U$, and functions from types to types – those functions with kind $\tau \Rightarrow \tau$ and typically written in the form $\text{TpFunc}(t).U$. Functions from types to types can be applied to a type and return a type, while a polymorphic type is just a type – the type of an expression that takes a type as a parameter and returns an element of the return type.

We now introduce the expressions of $\mathcal{P}SOOL$. New expressions include bounded polymorphic functions and their applications.

**Definition 15.1.1** The set of pre-expressions, $\mathcal{P}EXP_{\mathcal{P}SOOL}(\mathcal{E}, \mathcal{L}, \mathcal{EI}, \mathcal{TT})$ of $\mathcal{P}SOOL$ over a set $\mathcal{E}$ of expression constants, a set $\mathcal{L}$ of labels, a set $\mathcal{EI}$ of expres-
15.1 Introducing PSoOL

### Figure 15.1 Congruence rules for PSoOL.

<table>
<thead>
<tr>
<th>Rule</th>
<th>Premise</th>
<th>Conclusion</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>FuncAppCong</strong></td>
<td>$T : *$</td>
<td>$C \vdash (\text{TpFunc}(t) \cdot U)(T') \cong [T'/t]U$</td>
</tr>
<tr>
<td><strong>Congruence</strong></td>
<td>$C, E \vdash M : T$</td>
<td>$C \vdash T \cong T'$</td>
</tr>
<tr>
<td><strong>Cong &lt;:</strong></td>
<td>$S \cong S'$, $T' \cong T$, $C \vdash S &lt;: T$</td>
<td>$C \vdash S' &lt;: T'$</td>
</tr>
</tbody>
</table>

As above, the elided expressions represent the usual expressions of SoOL, which are omitted here.

An expression of the form $\text{polyFunc}(t <: T) : U$ is Block represents a polymorphic function that takes a type parameter and returns the element of type $U$ determined by evaluating Block. The second new expression represents the application of a polymorphic function to a type parameter. Notice that $F(T)$ represents the application of a type function to a type, while $E[T]$ represents the application of a polymorphic function to a type.

We present a detailed example of these constructs later in this chapter. However, simple examples of a type function and a polymorphic function follow.

$\text{PFcn} = \text{TpFunc}(t) \cdot t \rightarrow \text{Integer};$

$\text{polyx} = \text{polyFunc}(t <: \text{Point}) : t \rightarrow \text{Integer}$

$\text{ PFcn is a type function that send a type } T \text{ to the type } T \rightarrow \text{Integer. On the other hand, } polyx \text{ is a polymorphic function that takes a subtype, } T, \text{ of Point, and a value } p \text{ of type } T, \text{ and returns the integer obtained by sending a } \text{getx message to } p. \text{ The type of polyx is ForAll}(t <: \text{Point}). \text{PFcn}(t).$

The subtyping rules are the same as those for SoOL except for the addition of rules for constructor functions, constructor applications, and bounded
Adding Bounded Polymorphism to $\mathcal{SOOL}$

**Figure 15.2** New subtyping rules for $\mathcal{SOOL}$.

1. **Constructor**: $C \vdash U' \lll U$
   \[ C \vdash \text{TpFunc}(t).U' \lll \text{TpFunc}(t).U \]

2. **ConstructorApp**: $C \vdash F : \ast \Rightarrow \kappa$, $T : \ast$, $C \vdash F' : F$
   \[ C \vdash F'(T) : F(T) \]

3. **BdPoly**: $C \cup \{ t : T \} \vdash U' \lll U$
   \[ C \vdash \text{ForAll}(t : T).U' \lll \text{ForAll}(t : T).U \]

**Figure 15.3** Typing rules for new expressions of $\mathcal{SOOL}$.

- **PolyFcn**: $C, \mathcal{E} \vdash \text{Block}: U$
  \[ C \cup \{ t : T \}, \mathcal{E} \vdash \text{Block}: U \]

where $T$ may involve $t$.

- **PolyFcnApp**: $C, \mathcal{E} \vdash E : \text{ForAll}(t : T).U$
  \[ C, \mathcal{E} \vdash T' : [T'/t].T \]

- **PolyFcnApp**: $C, \mathcal{E} \vdash E[T'] : [T'/t].U$

Polymorphic types. The rules, which are similar to the corresponding rules for $\lambda^\sim_\alpha$, are given in Figure 15.2.

The definition of static type environments, $\mathcal{E}$, does not need to be modified from Definition 10.2.3, nor does the definition of type constraint system, $C$, from Definition 12.1.1, which already included bounded types.

Type-checking rules for the new expressions are given in Figure 15.3. Because the definition of type constraints allows the constraint $T$ on a type identifier $t$ to involve that identifier, we may type check instances of $F$-bounded polymorphism.

Recall the following example of GJ code from Section 4.1.

```java
interface OrderableF<T> {
    public boolean equal(T other);
    public boolean greaterThan(T other);
    public boolean lessThan(T other);
}
```
In \textit{PSOOL} we model \texttt{OrderableF} as a function from types to types:

\[
\text{OrderableF} = \text{TpFunc}(t).\text{ObjectType} \{
\text{equal: } t \rightarrow \text{Boolean};
\text{greaterThan: } t \rightarrow \text{Boolean};
\text{lessThan: } t \rightarrow \text{Boolean}
\}
\]

Thus \texttt{OrderableF} takes an object type parameter, \( T \prec \text{TopObject}, \) and returns an object type with \texttt{equal}, \texttt{greaterThan}, and \texttt{lessThan} methods that take parameters of type \( T \).

\texttt{OrderableF} can then be used in a class definition as follows:

\[
\text{BPOrderedList: ForAll} (\text{Elt} \prec \text{OrderableF}(\text{Elt})).
\]

\[
\text{BPClassTp}(\text{Elt}) = \text{polyFunc}(\text{Elt} \prec \text{OrderableF}(\text{Elt})):\text{BPClassTp}(\text{Elt}) \text{ is } \{
\text{return class } (\{ \ldots \}, \{ \ldots \})
\}
\]

if \( \text{class } (\{ \ldots \}, \{ \ldots \}) \) has type \( \text{BPClassTp}(\text{Elt}) \).

As with \textit{S\OOL}, we write programs in a language supporting convenient abbreviations. Parametric functions returning classes will be written as parameterized classes. Thus the above examples can be written in the following more readable style:

\[
\text{OrderableF}(t \prec \text{TopObject}) = \text{ObjectType} \{
\text{equal: } t \rightarrow \text{Boolean};
\text{greaterThan: } t \rightarrow \text{Boolean};
\text{lessThan: } t \rightarrow \text{Boolean}
\}
\]

\[
\text{class BPOrderedList}(\text{Elt} \prec \text{OrderableF}(\text{Elt})) \{ \text{return class } \{ \ldots \}; \}
\]

If we also have

\[
\text{SomeOrdType} = \text{ObjectType} \{
\text{equal: } \text{SomeOrdType} \rightarrow \text{Boolean};
\text{greaterThan: } \text{SomeOrdType} \rightarrow \text{Boolean};
\text{lessThan: } \text{SomeOrdType} \rightarrow \text{Boolean};
\ldots
\}
\]
then new BPorderedList(SomeOrdType) will create a new ordered list with elements of type SomeOrdType if we can show that SomeOrdType <: BPorderedList(SomeOrdType).

Interestingly, we have no way of showing directly that

\[ C \vdash \text{SomeOrdType} <: \text{OrderableF}(\text{SomeOrdType}). \]

However, OrderableF(SomeOrdType) is congruent to

\[
\text{SmallOrdType = ObjectType}\{ \\
\text{equal: SomeOrdType \rightarrow Boolean;} \\
\text{greaterThan: SomeOrdType \rightarrow Boolean;} \\
\text{lessThan: SomeOrdType \rightarrow Boolean}
\}
\]

and \( \emptyset \vdash \text{SomeOrdType} <: \text{SmallOrdType} \). Thus by Cong \( \vdash \) it is type safe to apply BPorderedList to SomeOrdType.

## 15.2 Translational semantics of \( \mathcal{PSOOL} \)

The translation of \( \mathcal{PSOOL} \) to \( \mathcal{LP} \) is straightforward, as \( \mathcal{LP} \) has constructs corresponding to all new constructs added in \( \mathcal{PSOOL} \). In order to simplify the notation we assume that restrictions have been made as suggested in Section 13.2 to ensure that class types have unique types. Thus we can define the translation by induction on the expressions rather than on their typings. As a result we will write the translation in the more notationally compact form of \( T_c[\mathcal{E}] \) rather than \( T_c[\mathcal{C}, \mathcal{E} \vdash \mathcal{E}: \mathcal{T}] \).

The translation of types and type functions is given in Figure 15.4, while the translation of expressions is given in Figure 15.5. We do not bother to repeat the translation of expressions of \( \mathcal{SOOL} \) as they are unchanged.

We interpret TopObject as an object type with no methods, but it could easily be changed to include methods of the sort contained in Java’s Object class, though all other object type definitions would have to implicitly contain those method names.

The proof of type safety is a straightforward extension of that for \( \mathcal{SOOL} \), and is left as an exercise for the reader. The proof consists of adding cases for bounded polymorphic functions and polymorphic function application. Type expressions involving function applications may be reduced using the congruence rules.
15.3 Summary

In this chapter we showed that the addition of F-bounded polymorphism to SOOL to form PSOOL is straightforward and raises no new issues in the translational semantics. This allows us to model object-oriented languages like GJ.

Next chapter we push the boundaries of statically typed object-oriented languages to begin investigating adding extra expressiveness similar to that found in languages like Eiffel and Beta. However, we will accomplish this while retaining a statically type-safe language.

\[
\begin{align*}
\mathcal{T}_c[\text{TpFunc}(t).U] &\triangleq \lambda(t <: \mathcal{T}_c[t]).\mathcal{T}_c[U] \\
\mathcal{T}_c[F(T)] &\triangleq \mathcal{T}_c[F(\mathcal{T}_c[T])] \\
\mathcal{T}_c[\text{TopObject}] &\triangleq \mathcal{T}_c[\text{ObjectType} | |] \\
\mathcal{T}_c[t] &\triangleq t \\
\mathcal{T}_c[\text{ForAll}(t <: T).U] &\triangleq \forall(t <: \mathcal{T}_c[T]).\mathcal{T}_c[U]
\end{align*}
\]

**Figure 15.4** Translation of type constructors and the new types of PSOOL to corresponding type constructors and types in $\Lambda^T$. 

\[
\begin{align*}
\mathcal{T}_c[C, E \vdash E[T'] : [T'/t] U] &\triangleq \mathcal{T}^X_C[C, E \vdash \text{ForAll}(t <: T).U \mid \mathcal{T}^X_C[T']] \\
\mathcal{T}_c[C, E \vdash \text{polyFunc}(t <: T): U \text{ is } \text{Block}: \text{ForAll}(t <: T).U] &\triangleq \lambda(t <: \mathcal{T}_c[T]).\mathcal{T}_c[C', E \vdash \text{Block}: U]
\end{align*}
\]

**Figure 15.5** Translation of selected expressions of PSOOL to expressions in $\Lambda^T$. 

15.3 Summary

In this chapter we showed that the addition of F-bounded polymorphism to SOOL to form PSOOL is straightforward and raises no new issues in the translational semantics. This allows us to model object-oriented languages like GJ.

Next chapter we push the boundaries of statically typed object-oriented languages to begin investigating adding extra expressiveness similar to that found in languages like Eiffel and Beta. However, we will accomplish this while retaining a statically type-safe language.
In the last chapter, we added F-bounded parametric polymorphism to $\mathcal{S}OOL$ to obtain the language $\mathcal{P}S\mathcal{OOL}$. In this chapter we see how to add, in a statically type-safe way, some of the expressiveness of Eiffel’s like Current construct, which was discussed in Section 4.2. We also compare the use of F-bounded polymorphism to this new construct.

In Chapter 3, we looked at the limits of the expressiveness of statically typed object-oriented languages with invariant typing disciplines. These are languages in which the types of instance variables and methods of a class are not allowed to vary in going from a class to a subclass. We also found that allowing covariant changes to the return types of methods (like C++ does) provided only a small amount of help in overcoming these limitations, as the other restrictions in the language kept us from defining the instance variables and methods in the subclass as directly as we desired.

In particular, limitations on the definition of subclasses still restricted us from expressing the cloning, node, and circle examples in Chapter 3, even after allowing for covariant changes in the return types of methods. It appeared that covariant changes to the types of instance variables and parameters of methods were required to allow the desired expressiveness of the language. Eiffel allows such changes, but we saw in Section 4.2 that allowing such covariant changes could lead to type errors.

The language $\mathcal{P}S\mathcal{OOL}$, which was introduced in the last chapter, added F-bounded polymorphism to $\mathcal{S}OOL$. This added greatly to the expressiveness of the language, but it does not easily help overcome some of the limitations that we described in Chapter 3.¹ In this chapter we show that providing

¹. Adding a type variable for each instance variable or parameter we wish to have change co-
more accurate information on the type of self will turn out to be the key to
overcoming many of these difficulties.

16.1 Typing self with MyType

In Section 14.3 we discussed some of the difficulties involved in assigning a
type to self. In this section we revisit this question in light of some of the
typing problems discussed in Chapter 3.

16.1.1 Weaknesses of type-checking rules for classes

Let us review how the two successive type-checking rules introduced for
classes in $\text{SOOL}$ (and hence $\text{PSoOL}$) handle the type of self. The original
rule for type-checking classes in Figure 10.8 of Section 10.2.3 is similar to
that of languages with simple type systems like C++, Java, Object Pascal,
and Modula-3. It is based on the assumption that self has essentially the
same type as the objects generated from the class being defined. In particular,
the type of self was taken to be $\text{SelfType} = \text{VisObjectType}(\text{IVref}, \text{M})$,
and type-checking rules were provided for accessing instance variables and
sending methods to objects of this type. We also provided a close operation,
that, when applied to an object of type $\text{VisObjectType}(\text{IVref}, \text{M})$, returned
an object of type $\text{ObjectType M}$. This can be applied to self in order to
aid in type checking methods that return self or take self as a parameter.

This rule turned out not to work smoothly with our translation of $\text{SOOL}$
into $\Lambda^\gamma$. As a result, in Figure 12.1 in Section 12.1 we introduced a second
version of the type-checking rule for classes in $\text{SOOL}$. With this rule, occurrences of self in the definition of a class with type $\text{ClassType (IV, M)}$
are assigned the type $\text{VisObjectType (IVR', M')}$, where all that is known
about IVR' and M' is that IVR' $\ll$ IVref and M' $\ll$ M. This rule provided a
better match with our translational semantics since we could ensure that the
methods would be type correct in all possible subclasses.

The rules introduced for sending messages to objects of type $\text{VisObject-
Type (IVR', M')}$ were sufficient to type check message sends to self because
the signature of a particular method of M' was guaranteed to be a sub-
type of the signature of the corresponding method of M. Thus a message send
of the form $\text{self} \leftarrow \text{m(...)}$ could be treated as having the same signature

variantly can help overcome some of these limitations, but only at the cost of nearly incompre-
hensible code.

Team-Fly®
as given in $M$. If methods returned `self` or took `self` as a parameter, extra work was required because $M'$ was only defined in the premise of the type-checking rule; it had no meaning in the conclusion of the rule. Thus `ObjectType M'` could not be used as the return type of a method (or as the parameter type of a publicly available method). The solution, discussed in Section 14.3, was to first apply `close` to `self`, resulting in an expression of type `ObjectType M'`, and then to use subtyping to promote the type of `close(self)` to `ObjectType M`.

Unfortunately, none of these rules solve all of the expressiveness problems we discussed earlier. For example, suppose `C` is a class generating objects of type `CType`. Then in both of the versions of the type-checking rules for classes, a method that takes no parameters and simply returns `self` would be provided with signature `Void → CType` (or some supertype of it).

Here is a simple example:

```plaintext
class C {
    x: Integer = 0;

    function returnMe(): CType is
        { return close(self) }
}
```

(Normally we will omit the application of `close` to `self` in example code, as it will typically be inserted by the type checker. We include it here because we are making a point about typing with `self`.) Assigning `close(self)` the type `CType` requires the use of subtyping, as discussed above.

Now look at what happens when we define a subclass `SC` of `C`.

```plaintext
class SC inherits C {
    function m'(...): ... is {...}
}
```

Suppose the type of object generated from `SC` is `SCType`. One would hope that the return type of the inherited method in `SC` would be `SCType`, but even in a language that allows covariant changes to return types, the type of `returnMe` does not change, because it is inherited without change.

The following code illustrates the problem:

```plaintext
sc: SCType;
sc := new SC;
... (sc ⇔ returnMe()) ⇔ m'(...) ... // type error
```
The last line of code will generate a static type error, even though it would be perfectly safe at run time. The problem is that the type checker will determine the type of the expression `sc ∘ returnMe()` to be `CType`, even though we know (by looking at the body of `returnMe`) that it will really return an object of the same type as the receiver – `SCType` in this case.

The programmer could add a type cast or typecase statement to test and use the dynamic type of the expression, but this simply highlights the failure of the static type system to determine the best type for the expression.

A workaround exists in a language supporting covariant changes to the return type of methods in subclasses. One would simply override `returnMe` in `SC` and copy the code for `returnMe` from the superclass to the subclass (inheritance by cut and paste!):

```plaintext
class SC inherits C modifies returnMe {
    function returnMe(): SCType is
    { return close(self) }

    function m'(...): ... is {...}
}
```

In this case, the fix involves just copying the single line of code that returns `self`, and would cause no difficulty. However in more complex situations it might result in having to copy over large amounts of code, creating a maintenance nightmare. A call to `super.returnMe()` could not be used, because it would also return the wrong type (though again, type casts or a typecase statement could be used to overcome the failure of the static type system to capture the correct dynamic type).

```plaintext
class SC inherits C modifies returnMe {
    function returnMe(): SCType is
    { return super.returnMe() // type error
    }

    function m'(...): ... is {...}
}
```

One would have to insert a downcast (or typecase statement) in order to coerce the results of `super.returnMe()` to be of type `SCType`. In summary, one might code around the problem, but it would be far from ideal.
Exercise 16.1.1 Show how to write the code for \( C \) and \( SC \) using F-bounded polymorphism. Include examples generating new objects from classes \( C \) and \( SC \).

16.1.2 The changing meaning of \texttt{self} and its type, \texttt{MyType}

The root of the problem arising in the use of inheritance with method \texttt{returnMe} above is that the meaning of \texttt{self} changes in the subclass to denote an element of type \texttt{SCType} rather than \texttt{CType}. Type checking the body of \texttt{returnMe} under the assumption that the type of \texttt{self} is a visible version of \texttt{CType} (as we did in the first version of the type checking rules for classes) is weaker than assuming its type is some visible object type that corresponds to a subtype of \texttt{CType}, as we did in the second version.

That is, from the assumption \texttt{close(self)} has type \texttt{CType'} for some (not yet determined) subtype of \texttt{CType}, we can infer by subsumption that \texttt{close(self)} also has type \texttt{CType}. Thus the second version of our type-checking rule actually allows us to type check methods with a stronger assumption on the type of \texttt{self}. The problem is that the type systems of our languages do not allow us to take advantage of this extra information. What we need is extra expressiveness in the \textit{type expressions} of our language so that we can capture the fact that the type of \texttt{self} changes when methods are inherited in subclasses.

To get this expressiveness we introduce the new type expression \texttt{MyType} to represent the type of \texttt{self}. The meaning of \texttt{MyType} will be similar to the Eiffel expression \texttt{like Current}.\textsuperscript{2} \texttt{MyType} can be used in typing methods and instance variables of classes, where it will always stand for the type of the object executing the code, \texttt{self}.

Suppose \( o \) is an object of type \texttt{CType} that was generated by class \( C \). If the keyword \texttt{self} occurs in method \( m \) of \( C \), then when \( o \leftarrow m(...) \) is executed, all occurrences of \texttt{self} in the method body will be interpreted as \( o \). Similarly, all occurrences of \texttt{MyType} encountered in evaluating \( o \leftarrow m(...) \) will be interpreted as \texttt{CType}. Thus the semantic interpretations of \texttt{self} and \texttt{MyType} in a message send are just the interpretations of the receiver and its type. When we restrict our attention just to simple classes and objects, it is not at all clear why we need to use \texttt{MyType}, as its meaning is just \texttt{CType}. The advantages really show up, however, in the use of subclasses, as we see next.

\textsuperscript{2} Aside from Eiffel, Trellis/Owl [SCB\textsuperscript{+} 86] and Emerald [BHJ\textsuperscript{+} 87] also provide constructs giving the type of \texttt{self} an explicit name. \texttt{MyType} is also sometimes written as \texttt{SelfType} in the literature.
In Section 3.2.1 we discussed a more detailed example involving a deep cloning method. The code for it can be found in Figure 3.1. We now show how the use of MyType solves the typing problem discussed there. In that example we presumed that all objects have a pre-defined method clone (inherited from a “Top” class) and wished to define a deepClone method that also cloned instance variables.

In rewriting this example in our more expressive language, we assume that the pre-defined method clone has type Void → MyType. We will write deepClone to have the same type. The code in Figure 16.1 contains a definition of class C with method deepClone. It also contains a definition of a subclass SC that adds a new field, nVar, a new method, setNVar, and redefines deepClone. The body of deepClone in SC calls the old version of deepClone in C (via the call to super), and then sets the value of the instance variable nVar in the new, cloned object to be a clone of the value in self.

Figure 16.2 provides an example of the use of these classes. We will provide formal type-checking rules soon, but, following the argument above on the meanings of self and MyType, we see that if c has type CType, then the type of c := deepClone() should also be CType, while if sc has type SCType, then the type of sc := deepClone() should be SCType. Thus the assignments on lines **2** and **3** are safe. This example shows how useful the MyType construct can be in defining subclasses. Because the meaning of MyType changes in going from a class to a subclass, the types of inherited methods can automatically change in subclasses.

The typing of the method body of deepClone in class SC of Figure 16.1 is also quite interesting. The keyword super provides a mechanism to access the methods of the superclass in a subclass. As discussed in Section 14.1, super has the same type as the record of methods of the superclass. Thus super.deepClone() has type MyType. As a result, the assignment **1** to nClone is well-typed, and hence the message send of setNVar to nClone is type-safe. While it is not needed in this example, an expression self := deepClone() would also be given type MyType.

The use of MyType also can solve the typing problem, discussed in Section 3.2.2, of defining a doubly linked node class as a subclass of a singly linked node class. We have rewritten the Node example from Figure 3.2 in Figure

---

3. Because of these automatic changes in the meaning of types in going from a class to a subclass, we will have to be careful in type checking method bodies to ensure that they remain type-safe in subclasses. More on this later.
Program DeepCloner;

CType = ObjectType {
    deepClone: Void \to MyType;
}

class C {
    ...
    function deepClone(): MyType is
    { super.clone();
        ...
    }
}

SCType = ObjectType {
    deepClone: Void \to MyType;
    setNVar: nObjType \to Void
}

class SC inherits C modifies deepClone {
    nVar: nObjType := nil;

    function setNVar(nVarVal:nObjType): Void is
    { nVar := nVarVal }

    function deepClone(): MyType is
    nClone: MyType;
    { // no type problems now!
        nClone := super.deepClone(); // **1**
        nClone \leftarrow setNVar(nVar \leftarrow clone());
        return nClone
    }

    function nMeth(): Void is { ... }
}

Figure 16.1 Typing deepClone methods with MyType.
Adding MyType to Object-Oriented Programming Languages

\begin{verbatim}
c, c': CType
sc, sc': SCType
{
c := new C;
c' := c \leftarrow deepClone();    // **2**
sc := new SC;
sc' := sc \leftarrow deepClone();   // **3**
    // Assignment safe - type of right side is SCType.
}
\end{verbatim}

Figure 16.2 Typing deepClone methods with MyType, continued.

16.3, replacing all occurrences of NodeType by MyType. We have also used MyType to specify the type of the instance variable next.

We now define DoubleNode as a subclass of Node in Figure 16.4. The use of MyType in the types of instance variables and methods ensures that all occurrences in inherited methods will change uniformly in the subclass.

In particular, examine the body of setNext in DoubleNode. In the first line, the setNext method of the superclass is invoked. In the superclass setNext took a parameter of type MyType, and that is indeed the type of the parameter, newNext. In the next line, the message setPrevious is sent to newNext, which has type MyType. The method setPrevious takes a parameter of type MyType, which is indeed the type of the receiver. The actual parameter is close(self), which by assumption has type MyType. Thus the body of setNext is well-typed.

16.1.3 Loss of subtyping – gain of matching

The types of objects generated by Node and DoubleNode are given in Figure 16.5. As the reader might suspect from the discussion in Section 4.2 on the failure of static type safety in Eiffel, the types NodeType and DoubleNode-Type cannot be subtypes because of the implicit change in the type of the parameter of setNext between the two types.

As a result we will have to be more careful in type checking methods involving expressions with type MyType. It will no longer be the case that
16.1 Typing self with MyType

```haskell
class Node {
  value: Integer := 0;
  next: MyType := nil;

  function getValue(): Integer is
  { return self.value }

  function setValue(newValue: Integer): Void is
  { self.value := newValue }

  function getNext(): MyType is
  { return self.next }

  function setNext(newNext: MyType): Void is
  { self.next := newNext }
}

Figure 16.3  Node class with MyType.

class DoubleNode inherits Node modifies setNext {
  previous: nil := MyType;

  function getPrevious(): MyType is
  { return self.previous }

  function setPrevious(newPrev: MyType): Void is
  { self.previous := newPrev }

  function setNext(newNext: MyType): Void is
  { // no error!
    super.setNext(newNext);
    newNext ← setPrevious(close(self)) }
}

Figure 16.4  Doubly linked node class with MyType.
NodeType = ObjectType {
    getValue: Void → Integer;
    setValue: Integer → Void;
    getNext: Void → MyType;
    setNext: MyType → Void
}

DoubleNodeType = ObjectType {
    getValue: Void → Integer;
    setValue: Integer → Void;
    getNext: Void → MyType;
    setNext: MyType → Void;
    getPrevious: Void → MyType;
    setPrevious: MyType → Void
}

Figure 16.5 Types of objects generated from node classes with MyType.

subclasses automatically give rise to subtypes. In particular, an element of
type DoubleNodeType may not be used in a context expecting an element
of type NodeType. This failure of subtyping might be considered to be a
great loss, but in practice the failure of subtyping in this case will hardly be
missed.

First of all, in a language without MyType, one could not even write DoubleNode
as a direct subclass of Node. Separate class definitions would be
required, and the types of the corresponding object types would certainly
not be in the subtype relation. Thus the use of MyType allows us to write
subclasses that could not be written previously. Thus we have the gain of
being able to write DoubleNode as a subclass rather than losing an existing
subtype.

Second, if a class does not contain occurrences of MyType in its method
definitions, then the types of the corresponding object types will still be in the
subtype relation. More generally, if MyType only occurs in the return types of
methods, the corresponding object types will be in the subtype relation. Thus
in the definitions in Figure 16.1 using MyType, SCType <: CType because
MyType only occurs as the return type of method deepClone.

4. We provide the exact rules for subtyping in the presence of MyType in the next section.
Finally, there is another relation between object types, called matching, that is satisfied by these types.

**Definition 16.1.2** Object type \( T \) matches object type \( U \) (written \( T <\# U \)) iff every method in \( U \) also occurs in \( T \), and the types of corresponding methods in \( T \) and \( U \) are in the subtype relation.

Of course, type \( T \) can also have more methods than \( U \) if \( T <\# U \). The matching relation is the same as subtyping in the absence of the \( \text{MyType} \) construct, but differs in the presence of \( \text{MyType} \) because \( \text{MyType} \) implicitly has different meanings in different types. In determining matching, occurrences of \( \text{MyType} \) in both types are treated as being the same. Thus \( \text{Double NodeType} \) matches \( \text{NodeType} \), even though they are not subtypes.

While the matching relation does not allow values of one object type to masquerade as values of the other, as is the case with subtyping, it does provide information on the availability and types of methods that is often sufficient in situations where subtyping might seem to be needed.

The most important point to keep in mind with matching is that it provides information about the methods available and their types. Thus knowing that type \( N \) matches \( \text{NodeType} \) provides the information that \( N \) has at least the methods \( \text{getValue}, \text{setValue}, \text{getNext}, \text{and setNext}, \) and that their signatures conform to those given in \( \text{NodeType} \).

Most of the time, this sort of information is all that is needed in object-oriented programming. The extra information that one type can masquerade as another in all possible circumstances is not needed. In the next chapter we will see that we can replace many occurrences of F-bounded polymorphism by the use of simple match-bounded polymorphism, providing another benefit of matching.

### 16.2 MOOL: Adding MyType to SOOL

In this section we provide the formal definition of the language \( \text{MOOL} \), which is obtained by adding \( \text{MyType} \) to \( \text{SOOL} \). In the next chapter we will extend \( \text{MOOL} \) with match-bounded polymorphism in order to obtain \( \text{PMOOL} \).

While the source language \( \text{MOOL} \) does not yet actually contain polymorphic functions, it will be notationally convenient to be able to write certain type expressions as the application of a type function to a type and to write certain value expressions as the application of a polymorphic function.
to a type expression. We will do this in order to be able to parameterize methods and instance variables by $MyType$. For example, the type of class Node in Figure 16.3 can be written as $\text{ClassType}(\text{NodeIV}(\text{MyType}), \text{NodeM}(\text{MyType}))$ using the parameterized types NodeIV and NodeM defined by:

$$\text{NodeIV}(MT) = \{\text{val: Integer}; \text{next: } MT\}$$

$$\text{NodeM}(MT) = \{\text{getValue: } \ldots; \text{setValue: } \ldots; \text{getNext: } \text{Void } \rightarrow \text{MT}; \text{setNext: } \text{MT } \rightarrow \text{ Void}\}$$

that are formed by replacing $MyType$ by a type variable $MT$ in the types of the class features. While allowing these type variables will have little impact on the language, it will make it more convenient to define the translation into $\mathcal{P\mathcal{OOL}}$. Moreover, we will soon extend the language by the addition of type functions and polymorphic functions. Using this notation now will make it simpler later to make the transition to the polymorphic language.

The kinds and types are the same as those in $\mathcal{P\mathcal{OOL}}$, except that for now we omit the types of polymorphic functions. As in $\mathcal{S\mathcal{OOL}}$, the type identifier $\text{SelfType}$ will refer to the type of $\text{self}$ within the body of a method and will allow access to instance variables as well as methods. $MyType$ will refer to the type of $\text{self}$ as seen from the outside when only methods are visible. The new type keyword $MyType$ is allowed to appear in $\mathcal{MOOL}$ programs in the types of instance variables and methods. However, as before, signatures of instance variables and methods may not reference $\text{SelfType}$; it is only used in type checking method bodies.

When writing object and class type expressions, we will write the types of the records of instance variables and methods as functions applied to $MyType$. In a sense this use of higher-order constructors is merely a notational convenience. Yet it makes it much easier to write down the type-checking rules and translation of $\mathcal{MOOL}$ type and value expressions because classes get translated as polymorphic functions. Thus we will write the expressions as

$$\text{ObjectType } M(MyType),$$

$$\text{VisObjectType } (\text{IVR(MyType)}, M(MyType)),$$

and

$$\text{ClassType } (\text{IV(MyType)}, M(MyType)) .$$

Each of these type expressions bind all free occurrences of $MyType$. Thus $MyType$, like $\text{self}$, is used as a keyword of $\mathcal{MOOL}$. 

"Team-Fly®"
We generalize the notion of a type constraint system, $C$, from that given in $\mathcal{P}SOOL$, so that $C$ may contain subtype constraints between elements of higher kinds.

**Definition 16.2.1** Relations of the form $x^e = G$ and $x^e <: G$, where $x^e$ is a type constructor variable and $G$ is a type constructor expression of the same kind are said to be type constructor constraints. A type constructor constraint system is defined as follows:

1. The empty set, $\emptyset$, is a type constructor constraint system.

2. If $C$ is a type constructor constraint system, $x^e$ is a constructor identifier that does not appear in $C$ or $G$, and $G$ is a constructor expression of the same kind, then $C \cup \{ x^e = G \}$ is a type constructor constraint system.

3. If $C$ is a type constructor constraint system, $x^e$ is a constructor identifier that does not appear in $C$, and $G$ is a constructor expression of the same kind, then $C \cup \{ x^e <: G \}$ is a type constructor constraint system.

We extend the definition of subtyping to handle constructors of higher kinds, and we redefine subtyping for object types to take the presence of $MyType$ into consideration. All other subtyping rules remain the same. The new rules are presented in Figure 16.6. The rule for subtyping type functions could be further generalized by allowing contravariant changes to the bounds on the arguments, but we will only need this simpler version.

The condition relating $MT'$ and $MT$ on the left side of the hypothesis of the $Object <: -$ rule ensures that the hypothesis of the rule fails (and hence the conclusion) if $M$ contains any negative occurrences of $MyType$. (See example below.) If $M(MyType)$ and $M'(MyType)$ result in type expressions that do not contain $MyType$ then the rule reduces to our earlier subtyping rule for object types. The hypothesis of this rule should remind the reader of the hypothesis for subtyping of recursive types in $\Lambda^{C_5}$. In fact, object types will translate to recursive types so the subtyping rule for recursive types will be needed to verify that the translation from $MOOL$ to $\Lambda^{C_5}$ preserves subtypes.

The other subtyping rules for types are as for $SOOL$.

The matching relation ($<$# ) is only defined on type expressions that represent object types. The matching rules are given in Figure 16.7.

All object types match $TopObject$, by rule $TopObject \downarrow$. Rule $Cong \downarrow$ states that matching is preserved by congruence.

The $Object <: -$ rule for subtyping is much more restrictive than the $Object \downarrow$ rule for matching, in that the subtyping rule for object types has a hypothe-
Adding MyType to Object-Oriented Programming Languages

TypeFcn \ll:< \quad \frac{C \vdash U' \ll: U}{C \vdash \text{TpFunc}(t).U' \ll: \text{TpFunc}(t).U}

where \( t \) does not occur in \( C \).

TypeFcn App \ll:< \quad \frac{C \vdash F' \ll: F}{C \vdash F'(T) \ll: F(T)}

Object \ll:< \quad \frac{C \cup \{MT' \ll: MT\} \vdash M'(MT') \ll: M(MT)}{C \vdash \text{ObjectType} M'(MyType) \ll: \text{ObjectType} M(MyType)}

where \( MT \) and \( MT' \) are new type identifiers not appearing in \( C, M, \) or \( M' \).

**Figure 16.6** Subtyping rules for higher kinds and replacement subtyping rule for object types.

TopObject \# \quad C \vdash \text{ObjectType} M(MyType) \ll: \# \text{TopObject}

Reflex \# \quad C \vdash \text{ObjectType} M(MyType) \ll: \# \text{ObjectType} M(MyType)

Transitive \# \quad \frac{C \vdash S \ll: T, \quad C \vdash T \ll: U}{C \vdash S \ll: U}

Object \# \quad C \vdash M' \ll: M

Object \ll:< \quad \frac{C \vdash \text{ObjectType} M'(MyType) \ll: \text{ObjectType} M(MyType)}{C \vdash M' \ll: M}

where \( M' \) and \( M \) have kind \( \ast \Rightarrow \ast \).

Type Abbrev \# \quad \frac{C \vdash C(T') \ll: C(T)}{C \vdash T' \ll: \# \ T}

Cong \# \quad \frac{S \equiv S', \quad T' \equiv T, \quad C \vdash S \ll: \# \ T}{C \vdash S' \ll: \# \ T'}

**Figure 16.7** Matching rules for object types.
sis that is harder to establish. That is, for proving matching, we can assume the types representing MyType are the same. With subtyping we are only allowed the weaker assumption that the MyType' in the first is a subtype of the MyType of the second. As a result it is harder to show that

\[ C \cup \{ MyType' <: MyType \} \vdash M'(MyType') <: M(MyType) \]

than

\[ C \vdash M' <: M. \]

However, we can easily show that if the first assertion is derivable then the second must also be derivable. (Use the same proof, but replace all occurrences of MyType' with MyType. Then apply rule TypeFcn <: . Thus if

\[ C \vdash ObjectType M'(MyType) <: ObjectType M(MyType) \]

then

\[ C \vdash ObjectType M'(MyType) <\# ObjectType M(MyType). \]

The types DoubleNodeType and NodeType, presented in Figure 16.5, are in the matching relation, but they are not subtypes. The difficulty in subtyping results from trying to show that the types of the setNext methods are in the subtype relation. This involves trying to show

\[ C \cup \{ MyType' <: MyType \} \vdash MyType' \rightarrow Void <: MyType \rightarrow Void \]

We cannot show this because of the contravariance in the domain of function types. To prove it would require an assumption that MyType <: MyType', the opposite of what is given.

Because the rules for matching are so simple, and match-bound constraints do not occur in \( C \), the following result should be obvious.

**Lemma 16.2.2** Suppose \( C \vdash T' <\# T \) where \( T \) is not TopObject. Then \( C(T') = ObjectType M'(MyType) \) and \( C(T) = ObjectType M(MyType) \) for some type functions \( M' \) and \( M \) such that \( C \vdash M' <: M \).

Figure 16.8 provides the rules for a variant, <\#\_vis, of the matching relation for visible object types. This relation also is reflexive and transitive, though we have omitted those rules. The types of the records of instance variables and the types of the records of methods must be subtypes if the visible object types are to be in the <\#\_vis relation. This relation will be useful in formulating typing rules.

As above, we obtain the following straightforward lemma:
314  Adding MyType to Object-Oriented Programming Languages

*VisObj* \(\#_\text{vis}\)  
\[
\frac{C \vdash M' <: M \quad C \vdash IVR' <: IVR}{C \vdash \text{VisObjectType}(IVR'(MyType), M'(MyType)) <: \#_\text{vis} \text{VisObjectType}(IVR(MyType), M(MyType))}
\]

where IVR', IVR, M', and M have kind \(\ast \Rightarrow \ast\),

*Type Abbrev* \(\#_\text{vis}\)  
\[
\frac{C \vdash C(T') <: \#_\text{vis} C(T)}{C \vdash T' <: \#_\text{vis} T}
\]

*Cong* \(\#_\text{vis}\)  
\[
\frac{S \cong S', \quad T' \cong T, \quad C \vdash S <: \#_\text{vis} T}{C \vdash S' <: \#_\text{vis} T'}
\]

**Figure 16.8** Rules for \(<: \#_\text{vis}\).

**Lemma 16.2.3** Suppose \(C \vdash T' <: \#_\text{vis} T\). Then

\[
C(T') = \text{VisObjectType}(IVR'(MyType), M'(MyType)),
\]

and

\[
C(T) = \text{VisObjectType}(IVR(MyType), M(MyType)),
\]

for some type functions IVR', IVR, M', and M such that \(C \vdash IVR' <: IVR\) and \(C \vdash M' <: M\).

The expressions of \(MOOL\) are the same as those of \(SOOL\) (modulo the addition of \(MyType\) as a type keyword). The difference in expressiveness comes entirely through the use of \(MyType\) in the types of methods and instance variables.

The type-checking rules of \(MOOL\) are similar to those for \(SOOL\), except for those for classes, subclasses, and message sends, which now involve \(MyType\) and matching. Figures 16.9 and 16.10 contain the new type checking rules for these expressions. As mentioned earlier, for notational convenience, we write the record of methods and values of instance variables as if they were polymorphic functions that take type parameters.

There are several differences between these rules and the type-checking rules for classes and subclasses for \(SOOL\) given in Figure 12.1. As before, new subtype constraints are added to \(C\) for the purpose of type checking. However, this time the variables, \(M'\) and \(IVR'\), and their constraints, \(M\) and \(IVref\), are functions from types to types. We also add to \(C\) a definition for \(MT\),
16.2 **MOOL: Adding MyType to SOOL**

\[
\begin{align*}
C', \mathcal{E} \vdash \text{inst}[MT] : IV(MT), \\
C', \mathcal{E}' \vdash \text{meth}[MT][\text{SelfType}] : M(MT)
\end{align*}
\]

where

- \( C' = C \cup \{ M' \leftarrow M, IVR' \leftarrow IV^{M'}, MT = \text{ObjectType } M'(MyType), \text{SelfType} = \text{VisObject} Type (IVR'(MyType), M'(MyType)) \} \),
- \( \mathcal{E}' = \mathcal{E} \cup \{ \text{self: SelfType, close: SelfType \rightarrow MT} \} \),
- None of \( M', IVR', MT, \text{or SelfType} \) occurs free in \( \text{inst, meth, IV, M, C, or } \mathcal{E} \).

\[
\begin{align*}
C', \mathcal{E} \vdash E : \text{ClassType}(IV^{sup}(MyType), M^{sup}(MyType)), \\
C', \mathcal{E} \vdash \text{inst}[MT] : IV^{sub}(MyType), \\
C', \mathcal{E}' \vdash \text{meth}[MT][\text{SelfType}] : M^{sub}(MyType), \\
C' + T'_i \leftarrow T_i, \text{ for } 1 \leq j \leq m
\end{align*}
\]

Subclass

\[
\begin{align*}
C', \mathcal{E} \vdash \text{class inherits E modifies } l_1, \ldots, l_m \\
\text{inst}[MyType], \text{meth}[MyType][\text{SelfType}] : \\
\text{ClassType}(IV(MyType), M(MyType))
\end{align*}
\]

where

- \( IV = IV^{sup} \oplus IV^{sub} \) and \( M = M^{sup} \oplus M^{sub} \),
- There is no overlap in the labels occurring in \( IV^{sup} \) and \( IV^{sub} \), and
- The overlapping labels in \( M^{sup} \) and \( M^{sub} \) are exactly \( l_1, \ldots, l_m \),
- The type of \( l_i \) in \( M^{sub}(MT) \) is \( T'_i \), while the corresponding type in \( M^{sup}(MT) \) is \( T_i \).
- \( C' = C \cup \{ M' \leftarrow M, IVR' \leftarrow IV^{M'}, MT = \text{ObjectType } M'(MyType), \text{SelfType} = \text{VisObject} Type (IVR'(MyType), M'(MyType)) \} \),
- \( \mathcal{E}' = \mathcal{E} \cup \{ \text{self: SelfType, close: SelfType \rightarrow MT} \} \),
- None of \( M', IVR', MT, \text{or SelfType} \) occurs free in \( \text{inst, meth, IV, M, C, or } \mathcal{E} \).

**Figure 16.9** Typing rules for classes and subclasses in **MOOL**.
an identifier that will eventually be interpreted as MyType, the type of self from outside the object.

As before, in type checking method bodies, \( E \) is extended to include a type for self, and the function close is also added with type \( \text{SelfType} \to \text{MT} \). As mentioned earlier, in the source language, the type checker would generally be smart enough to be able to insert close automatically wherever it is needed so that the user would not even have to be aware of it. See the example in the next section.

Even though \( M' \) and \( IVR' \) are introduced in new constraints, they may not appear in the methods or instance variables. Rather the identifier \( \text{MT} \), whose definition in \( C' \) involves \( M' \), replaces MyType in instance variables and method bodies when they are type checked. Notice that

\[
C' \vdash \text{MT} < \# \ \text{ObjectType} M \{ \text{MyType} \}
\]

because \( C' \vdash M' <: M \) and \( C'(\text{MT}) = \text{ObjectType} M' (\text{MyType}) \). Similarly

\[
C' \vdash \text{SelfType} < \# \ \text{vis} \ \text{VisObjectType}(IVR'(\text{MyType}), M(\text{MyType}))
\]

because of the definition of SelfType, \( C' \vdash IVR' <: IVR' \), and \( C' \vdash M' <: M \).

When the user is checking to see whether the instance variables and methods type check, it will generally be by using these matching relations rather than the higher-order subtyping. Anything provable from these matching relations will also be provable from \( C' \). When presenting this system to a programmer, only these relations (rather than the higher-order subtyping) will normally be presented in explaining how programs are type checked. The stronger rule presented here is needed only for the proof of soundness of the type system.

Later we will show how to translate class expressions to the extended lambda calculus, and that our translation is sound. For now we simply remark that, as in \( SLOC \), the constraints on the type functions used in defining SelfType and MyType ensure that methods that type check correctly in a class will remain type safe in all possible subclasses.

The type-checking rule for subclasses is obtained from the previous rule for type checking subclasses in a way similar to the changes made for classes. As before, we have not included the use of super in type checking subclasses. It can be added similarly to the way indicated in Section 14.1.

The new rules for message sending and instance variable access are given in Figure 16.10. One change in the new message-sending rule is that the type of an expression representing sending a message to an object is the type of
16.2 MOOL: Adding MyType to SOOL

\[
\text{Message} \quad \frac{C, \mathcal{E} \vdash E : U, \quad C \vdash U \triangleleft \# \ \text{ObjectType} \ \{ m : T(\text{MyType}) \}}{C, \mathcal{E} \vdash E \leftarrow m : T[U]}
\]

\[
\text{Inst Vble} \quad \frac{C, \mathcal{E} \vdash E : U, \quad C \vdash U \triangleleft \# \ \text{vis} \ \text{VisObjectType} \ \{ l : T(\text{MyType}) \}, \ { 0 \}}{C, \mathcal{E} \vdash E.l : T[V]}
\]

where \( C(U) = \text{VisObjectType}(\text{IVR}(\text{MyType}), \text{M(MyType)}) \), for some type constructors \( \text{IVR} \) and \( \text{M} \), and \( V \) is a type such that \( C(V) = \text{ObjectType}(\text{M(MyType)}) \).

\[
\text{VisObj Message} \quad \frac{C, \mathcal{E} \vdash E : U, \quad C \vdash U \triangleleft \# \ \text{vis} \ \text{VisObjectType} \ \{ 0 \}, \ { m \in T(\text{MyType}) \}}{C, \mathcal{E} \vdash E \leftarrow m : T[V]}
\]

where \( C(U) = \text{VisObjectType}(\text{IVR}(\text{MyType}), \text{M(MyType)}) \), for some type constructors \( \text{IVR} \) and \( \text{M} \), and \( V \) is a type such that \( C(V) = \text{ObjectType}(\text{M(MyType)}) \).

**Figure 16.10** Typing rules for message sending and instance variables in MOOL.

the method, updated so that all occurrences of MyType are replaced by the type of the receiver. For example, if \( n : \text{NodeType} \) and the type of method \( \text{getNext} \) in \( \text{NodeType} \) is \( \text{Void} \rightarrow \text{MyType} \), then the type of \( n \leftarrow \text{getNext} \) is \( \text{Void} \rightarrow \text{NodeType} \). Similarly, \( n \leftarrow \text{setNext} \) has type \( \text{NodeType} \rightarrow \text{Void} \) because \( \text{setNext} \) has type \( \text{MyType} \rightarrow \text{Void} \). In each case, all occurrences of MyType in the method signature are replaced by the type of the receiver, NodeType.

As before, we have generalized the hypothesis for the type-checking rule for message sends to handle the case in which the only information known about the type of the methods in the receiver is via an upper bound with respect to the matching ordering. This generalized form is used when we send a message to an object with type MyType.

For example, in the body of method \( \text{setNext} \) of \( \text{DoubleNodeClass} \), the method \( \text{setPrevious} \) is sent to instance variable \( \text{newNext} \) of type MyType. When type checking the class, the type constraints include the information that MyType \( \triangleleft \# \ \text{DoubleNodeType} \), where DoubleNodeType includes the method \( \text{setPrevious} \) with type \( \text{MyType} \rightarrow \text{Void} \). As a result, the type
of newNext ⇐ setPrevious is MyType → Void,\(^5\) and hence the type of
newNext ⇐ setPrevious(close(self)) is Void.

This new rule for type checking message passing formalizes the earlier dis-
cussion of MyType and message sending, and should make intuitive sense in
that MyType is supposed to stand for the type of the receiver of a message.
Thus when we actually know the receiver and its type, we can replace My-
Type by the actual type. Because the keyword MyType is a bound variable
inside the object type, it would be a serious mistake to allow it as a free vari-
able in the type of the message send expression. Thus we must replace it
when the type of the method is extracted from the object type.

The rules for extracting instance variables and sending messages to ex-
pressions whose types are visible object types are similar. Note that while
the expression itself has a visible object type, \(U\), we must replace occurrences
of MyType in the method body or instance variable by the object type, \(V\),
which corresponds to the visible object type, \(U\). A consequence of this corre-
spondence is that a value of object type \(V\) is obtained by applying close to
an expression of type \(U\).

In practice, each of these rules will typically only be applied in type check-
ing the body of methods when the receiver, \(E\), is actually the expression self
with type SelfType. In this case, SelfType corresponds to \(U\), and MT will
be the corresponding \(V\) to be substituted for MyType in the conclusion of the
rule. (After all, \(MT = ObjectType M'(MyType), \text{ and } SelfType = VisOb-
jectType(IVR'(MyType),M'(MyType))\) in \(C'\).)

Rather than writing a rule that applies only to self:SelfType, we have
included a more general rule. However, the simplified rules for when we
have an expression of type SelfType are given below.

\[
C, \mathcal{E} \vdash E : \text{SelfType,} \\
\text{Inst Vble} \quad C \vdash \text{SelfType} <\#_{\text{vis}} \text{ VisObjectToy}(\{\{1 : T(MyType)\},\{\} \}) \\
C, \mathcal{E} \vdash E.1 : T(MT)
\]

where close: SelfType → MT ∈ \(E\), MT = ObjectType M'(MyType) ∈ \(C\),

---

\(^5\) We have replaced all occurrences of MyType by the type of the receiver, MyType, so it looks
like we haven’t made any changes. However, we followed the type-checking rule and made the
required replacement of all occurrences of MyType in the type of the method by the type of the
receiver, also MyType, which doesn’t actually change anything!
and SelfType = VisObjectType(IVR'(MyType), M'(MyType)) ∈ C for some type constructor identifiers, IVR' and M'.

\[
\text{VisObj Message} \quad C, \mathcal{E} \vdash E : \text{SelfType},
\]

\[
\text{VisObjectMessage}(\parallel, \mid m \in \text{T(MyType)} \mid)
\]

\[
C, \mathcal{E} \vdash E \iff m : \text{T(MT)}
\]

where close : SelfType → MT ∈ \mathcal{E}, MT = ObjectType M'(MyType) ∈ C, and SelfType = VisObjectType(IVR'(MyType), M'(MyType)) ∈ C for some type constructor identifiers, IVR' and M'.

16.3 Translational semantics of \MOOC

The translation of \MOOC into \\\textit{\L} is only a bit more complex than before. One of the main differences is that the types of methods and instance variables are parameterized by MyType. Thus in the translation of classes and their types we will use subtyping on functions from types to types rather than just on simple types. We must also take a fixed point in order to obtain the meaning of MyType.

The translations of type expressions and type constructor constraint systems are mutually recursive. We will begin with the translation of type constructor constraint systems. Our definition will follow the cases in the definition of type constructor constraint systems. Let C' be a type constructor constraint system. Then,

1. \( T_{C'}[\emptyset] \overset{\text{def}}{=} \emptyset. \)

2. If \( C \cup \{ x^c = G \} \) is a type constructor constraint system, then \( T_{C'}[C \cup \{ x^c = G \}] \overset{\text{def}}{=} T_{C'}[C]. \)

3. If \( C \cup \{ x^c <: G \} \) is a type constructor constraint system, then \( T_{C'}[C \cup \{ x^c <: G \}] \overset{\text{def}}{=} T_{C'}[C] \cup \{ x^c <: T_{C'}[G] \}. \)

As before, type definitions are defined away, and subtype constraints are translated in a straightforward way.

The translation of \( \mathcal{E} \) is simply done pointwise. That is, \( x : T \in \mathcal{E} \) iff \( x : T_{C'}[T] \in T_{C'}[\mathcal{E}]. \)
Adding MyType to Object-Oriented Programming Languages

\[ T_C[\text{TpFunc}(t).U] \overset{\Delta}{=} \lambda t.\ T_C[U], \]

\[ T_C[\text{P}(U)] \overset{\Delta}{=} T_C[P(T_C[U])] \]

\[ T_C[\text{ObjectType}' \text{MyType}'] \overset{\Delta}{=} \text{Obj}(T_C[M]) \]

\[ T_C[\text{VisObjectType}(\text{IVR}[\text{MyType}], M[\text{MyType}])] \overset{\Delta}{=} \text{VisObj}(T_C[\text{IVR}](\text{Obj}(T_C[M])), T_C[M](\text{Obj}(T_C[M]))) \]

\[ T_C[\text{ClassType}(\text{IV}[\text{MyType}], M[\text{MyType}])] \overset{\Delta}{=} \forall (M' <: T_C[M]).\forall (IV'[<: T_C[\text{IV'}]).\ T_C[\text{IV}][\text{Obj}(M')] \times (\text{VisObj}(\text{IVR}'(\text{Obj}(M')), M'(\text{Obj}(M'))) \rightarrow T_C[M](\text{Obj}(M'))) \]

where

\[ \text{Obj}(M) \overset{\Delta}{=} \text{Fix}(\lambda \text{MT}. \exists Y. \ Y \times (Y \rightarrow M(\text{MT}))) \]

\[ = \exists Y. \ Y \times (Y \rightarrow \text{M}(\text{Obj}(M))). \]

\[ \text{VisObj}(\text{IVR}, M) \overset{\Delta}{=} \text{IVR} \times (\text{IVR} \rightarrow M). \]

Figure 16.11 Translation of new types and constructors.

Figure 16.11 contains the translations of the new types and type constructors of \( \mathcal{M}COC \), as well as those that changed because of the addition of My-Type. Most of the basic non-object types are interpreted as before, and hence are not included in the figure.

The translation of type functions is straightforward as they are simply translated pointwise. Type function applications are also translated as the application of the translation of the function applied to the translation of the argument.

In the following paragraphs, let IV, M:: * \rightarrow * represent type functions that, when provided with the meaning of MyType, return the types of the records of instance variables and methods of an object. (See the examples of NodeIV and NodeM in Section 16.2.)

Before the introduction of MyType, visible object types were translated as pairs of the type of instance variables and method suite (parameterized by the instance variables). Object types were defined similarly except that the
translational semantics of MOOL

16.3 Translational semantics of MOOL

Type of the record of instance variables was abstracted away using an existential type. The only thing different here is that we must interpret \texttt{MyType}. Because \texttt{MyType} is supposed to represent the type of the object, we can define it using a recursive type. Thus the translation of object types in the figure begins with a fixed point on a type variable, \( \texttt{MT} \), representing \texttt{MyType}:

\[
\texttt{T}_C[\texttt{ObjectType M(MyType)}] \triangleq \text{Fix}(\lambda \texttt{MT}. \exists Y. Y \times (Y \rightarrow \texttt{T}_C[\texttt{M}(\texttt{MT})]))
\]

The definition in the figure has been simplified by using the updated abbreviation for \texttt{Obj(M)} contained in the figure.

Because of the strong hypotheses in the \texttt{Fix} rule in Figure 9.6 for proving subtyping of recursive types, knowing that \( M' <: M \) (as functions from types to types) is not sufficient to guarantee that

\[
\texttt{T}_C[\texttt{ObjectType M'}(\texttt{MyType})] <: \texttt{T}_C[\texttt{ObjectType M(MyType)}],
\]

unless the type argument of \( M \) (representing \texttt{MyType}) occurs only positively, \textit{i.e.}, as a return type, in \( M \).

In the interpretation of “visible object types”, \texttt{MyType} will again be interpreted as \texttt{Obj(T}_C[\texttt{M}]) , but now the types of the instance variables will also be visible.

\[
\texttt{T}_C[\texttt{VisObjectType(IVR(MyType), M(MyType))}] \triangleq \\
\texttt{T}_C[\texttt{IVR}(\texttt{Obj}(\texttt{T}_C[\texttt{M}]))] \times (\texttt{T}_C[\texttt{IVR}(\texttt{Obj}(\texttt{T}_C[\texttt{M}]))] \rightarrow \texttt{T}_C[\texttt{M}(\texttt{Obj}(\texttt{T}_C[\texttt{M}]))]).
\]

(Recall that we use \texttt{IVR} to suggest that this is the type of the record of instance variables rather than just their values.) The definition in the figure utilizes the (old) abbreviation \texttt{VisObj(IVR,M)} to simplify notation.

As before, the meaning of classes is parameterized by the types of all possible subclasses. Again, because both \( \texttt{T}_C[\texttt{IV}] \) and \( \texttt{T}_C[\texttt{M}] \) are functions from types to types, they will need to be applied to the meaning of \texttt{MyType} in the subclass, \texttt{Obj(M')}. Moreover, the bounded quantification in the translation of class types is based on subtyping of functions from types to types.

As with the earlier translation of classes, the translation is parameterized over all extensions of the types of the instance \textit{variables} (not just the types of their \textit{values}) and methods, and returns a pair composed of the record of initial \textit{values} and the methods, where the methods are parameterized by the meaning of \texttt{self} (which has type \( \texttt{VisObj(IVR(Obj(M')), M'(Obj(M')))} \)).

Comparing this definition with the one given in Section 12.1, the main differences are that \texttt{IV} and \( M \) are now functions from types to types, and
thus \( \tau_C[\text{IV}] \) and \( \tau_C[\text{M}] \) must be applied to \( \text{Obj}(M') \) in order to provide the interpretation of \( \text{MyType} \).

We now provide the new translations of expressions for \( \text{MOOL} \). The translation of most expressions is the same as for \( \text{SOOL} \). The only ones that have changed are for classes, subclasses, new expressions, message sending, and access to instance variables.

\[
\mathcal{C}, \mathcal{E} \vdash \text{closeobj}; \forall \text{IVR}. \forall \text{M}. \text{VisObj}(\text{IVR}((\text{Obj}(\text{M}))), \text{M}((\text{Obj}(\text{M})))) \rightarrow \text{Obj}(\text{M})
\]

Thus the interpretation of \( \text{close} \) is a function with type \( \tau_C[\text{SelfType}] \rightarrow \tau_C[\text{MyType}] \).

The new translation of classes, subclasses, and new expressions is given in Figure 16.12. In the translation for both classes and subclasses, the type parameters \( M' \) and \( \text{IVR}' \) now represent functions from types to types. Notice also that the interpretation of \( \text{MyType} \) in both instance variables and methods is \( \text{Obj}(M') \). Thus \( \text{MyType} \) is interpreted as the type of objects generated when the type function determining the type of the method suite is \( M' \). Similarly, \( \text{SelfType} \) is interpreted as \( \text{VisObj}(\text{IVR}'((\text{Obj}(M'))), \text{M}'((\text{Obj}(M')))) \), the visible object type that results when the type function determining the type of the instance variables is \( \text{IVR}' \), and the type function \( M' \) determines the type of the methods. Notice that we follow our earlier convention and translate classes and subclasses as having the most expressive type, i.e., the one that includes all instance variables and methods.

The expression \( \text{close} \), which may occur in method definitions, is interpreted as an instantiation of the function \( \text{closeobj} \), which was defined in Section 11.2, and takes visible object types to object types.

### 16.4 Soundness of translation for \( \text{MOOL} \)

In this section we show that the translational semantics for \( \text{MOOL} \) is sound. We will not bother to repeat the portions of the proof that are nearly identical to those for \( \text{SOOL} \), but instead focus on the new encoding of classes and objects. We begin by showing the preservation of subtyping.

**Theorem 16.4.1** Let \( \mathcal{C} \) and \( \mathcal{T} \) be type constructor expressions of \( \text{MOOL} \) such that \( \mathcal{C} \vdash \mathcal{S} \prec: \mathcal{T} \). Then \( \tau_C[\mathcal{C}] \vdash \mathcal{C}_{\tau_C}[\mathcal{S}] \prec: \mathcal{T}_{\tau_C}[\mathcal{T}] \).

**Proof.** We only provide the proof for object types. The other cases are easy.
\( T_C \vdash \text{class} \{ \text{inst [MyType], meth [MyType]} \} \{ \text{SelfType} \} : \\
\text{ClassType}(\text{IV [MyType]}, M[\text{MyType}]) \quad \triangleq \\
\Lambda(M' \leftarrow T_C[M], \Lambda(IVR' \leftarrow T_C[IV'^ref]). \\
\langle T_C[C', E \vdash \text{inst [MyType]} : \text{IV [MyType]}], \text{methfun} \rangle \\
\text{where MyType} = \text{Obj} (M'), \text{SelfType} = \text{VisObj}(IVR' (MyType), M' (MyType)), \\
\text{methfun} = \lambda (\text{self: SelfType}). \\
\text{let close: SelfType} \rightarrow \exists Y. \text{VisObj}(Y, M' (MyType)) \\
= \text{closeobj} [IVR' (MyType)] [M'(MyType)]. \\
C' = C \cup \{ M' \leftarrow M, IVR' \leftarrow IV'^ref, MT = \text{ObjectType} M' (MyType), \\
\text{SelfType} = \text{VisObjectType}(IVR' (MyType), M' (MyType)), \} \\
\text{and} \text{inst' is formed by adding ref's to all fields of inst}, \\
\text{methfun' = fix} [IVR \rightarrow M] \lambda (fm: IVR \rightarrow M). \lambda (\text{inst': IVR}). \\
\text{methfun} (\langle \text{inst'}, fm \rangle). \\
\)

Figure 16.12 Translation of classes and subclasses of \( MOOC \) to \( \Lambda_C \).
Suppose $C \vdash \text{ObjectType} M'(\text{MyType}) <: \text{ObjectType} M(\text{MyType})$. By the subtyping rules for $\mathcal{MOOL}$, this can only occur if $C' \vdash \text{MyType} <: M(\text{MyType})$ where $C' = C \cup \{\text{MyType} <: M(\text{MyType})\}$.

By induction, $\mathcal{T}_C[C'] \vdash \mathcal{T}_C[Z'[(\text{MyType}')]] <: \mathcal{T}_C[Z[M(\text{MyType})]]$. By Lemma 13.1.2, it follows that $\mathcal{T}_C[Z] = \mathcal{T}_C[Z']$, for all constructors $Z$. By this, the function and product type subtyping rules of $\lambda^P$, it follows that

$$\mathcal{T}_C[C] \vdash Y \times (Y \rightarrow \mathcal{T}_C[Z'(\text{MyType}')]) <: Y \times (Y \rightarrow \mathcal{T}_C[Z[M(\text{MyType})]].$$

Because $\mathcal{T}_C[C] = \mathcal{T}_C[Z] \cup \{\text{MyType} <: MT\}$, $\mathcal{T}_C[Z'(\text{MyType}')] = \mathcal{T}_C[Z'(\text{MyType})]$, and $\mathcal{T}_C[Z[M(\text{MyType})]] = \mathcal{T}_C[Z[M(\text{MyType})]]$

Then, by rule $\text{Exist} <: $

$$\mathcal{T}_C[C] \cup \{MT' <: MT\} \vdash Y \times (Y \rightarrow \mathcal{T}_C[Z'(\text{MyType}')]) <:
\exists Y. Y \times (Y \rightarrow \mathcal{T}_C[Z[M(\text{MyType})]].$$

Finally by the subtyping rule for recursive types, $\text{Fix} <: $

$$\mathcal{T}_C[C] \vdash \text{Fix}(\lambda \text{MyType}. \exists Y. Y \times (Y \rightarrow \mathcal{T}_C[Z'(\text{MyType}')]) <:
\text{Fix}(\lambda \text{MyType}. \exists Y. Y \times (Y \rightarrow \mathcal{T}_C[Z[M(\text{MyType})]]).$$

But this simply shows that

$$\mathcal{T}_C[C] \vdash \mathcal{T}_C[Z[\text{Object}Type M'(\text{MyType})]] <: \mathcal{T}_C[Z[\text{Object}Type M(\text{MyType})]].$$

Now that we know that the translation preserves subtypes, we continue to show that the translation preserves the types of expressions.

**Theorem 16.4.2** Suppose $C, E \vdash E : S$. Then $\mathcal{T}_C[C], \mathcal{T}_C[E] \vdash \mathcal{T}_C[E] : \mathcal{T}_C[S]$.

**Proof.** As before, the proof will be by induction on the proof of the typing of expressions. Most of the proof proceeds just like Theorem 13.1.10. We will only include here those parts of the proof that have changed.

**Class** Suppose that

$$C, E \vdash \text{class}(\text{inst}[\text{MyType}], \text{meth}[\text{MyType}][\text{SelfType}]),
\text{ClassType}(IV(\text{MyType}), M(\text{MyType}))$$
16.4 Soundness of translation for MOOL

because

\[ C', \mathcal{E} \vdash \text{inst}[MT]: IV(MT) \]

and

\[ C', \mathcal{E}' \vdash \text{meth}[MT][\text{SelfType}]: M(MT) \]

where

- \( C' \overset{\Delta}{=} C \cup \{ M' \prec C, IV' \prec IV^{\sigma}, MT = \text{ObjectType}(M'(\text{MyType})), \text{SelfType} = \text{VisObject}(IV'(\text{MyType}), M'(\text{MyType})) \} \)
- \( \mathcal{E}' \overset{\Delta}{=} \mathcal{E} \cup \{ \text{self}: \text{SelfType}, \text{close}: \text{SelfType} \to MT \} \), and
- none of \( M', IV', MT, \) or \( \text{SelfType} \) occurs free in \( \text{inst}, \text{meth}, IV, M, C, \) or \( \mathcal{E} \).

Now

\[
\begin{align*}
\mathcal{T}_C(\text{class}(\text{inst}[\text{MyType}], \text{meth}[\text{MyType}][\text{SelfType}])) & \overset{\Delta}{=} \\
\Lambda(\text{M}' \prec \mathcal{T}_C[\text{M}]). \Lambda(IV' \prec \mathcal{T}_C[IV^{\sigma}]). \\
& (\mathcal{T}_C[\text{inst}][\text{MyType}, \text{methfun}] )
\end{align*}
\]

where \( \text{MyType} = \text{Obj}(M') \),
\( \text{SelfType} = \text{VisObj}(IV'(\text{MyType}), M'(\text{MyType})) \),
\( \text{methfun} = \lambda(\text{self}: \text{SelfType}) \).

let \( \text{close}: \text{SelfType} \to \exists Y. \text{VisObj}(Y, M'(\text{MyType})) \)

\( = \text{closeobj}[IV'(\text{MyType})][M'(\text{MyType})] \)

in \( \mathcal{T}_C[\text{meth}][\text{MyType}][\text{SelfType}] \).

Also

\[
\begin{align*}
\mathcal{T}_C[\text{ClassType}(IV(\text{MyType}), M(\text{MyType}))] & \overset{\Delta}{=} \\
\forall (M' \prec \mathcal{T}_C[\text{M}]). \forall (IV' \prec \mathcal{T}_C[IV^{\sigma}]). \\
& \mathcal{T}_C[IV](\text{Obj}(M')) \times \\
& (\text{VisObj}(IV'(\text{Obj}(M')), M'(\text{Obj}(M'))) \to \mathcal{T}_C[\text{M}](\text{Obj}(M'))) 
\end{align*}
\]

For simplicity, we will use the abbreviations for \( \text{MyType}, \text{SelfType}, \) and \( \text{meth} \) for the rest of this part of the proof. For example, we can rewrite the above equation as:

\[
\begin{align*}
\mathcal{T}_C[\text{ClassType}(IV(\text{MyType}), M(\text{MyType}))] & \overset{\Delta}{=} \\
\forall (M' \prec \mathcal{T}_C[\text{M}]). \forall (IV' \prec \mathcal{T}_C[IV^{\sigma}]). \\
& \mathcal{T}_C[IV](\text{MyType}) \times (\text{SelfType} \to \mathcal{T}_C[\text{M}](\text{MyType}))
\end{align*}
\]
By induction,
\[ T_C[\mathcal{C}'], T_C[\mathcal{E}] \vdash T_C[\text{inst} [\text{MT}]]: T_C[\text{IV}(\text{MT})] \]
and
\[ T_C[\mathcal{C}'], T_C[\mathcal{E}'] \vdash T_C[\text{meth}[\text{MT}][\text{SelfType}]]: T_C[\text{M}(\text{MT})], \]
By the definition of the translation of type variables,
\[ T_C[\text{IV}(\text{MT})] = T_C[\text{IV}][\text{Obj}(\text{M}')] = T_C[\text{IV}](\text{MyType}) \]
and
\[ T_C[\text{M}(\text{MT})] = T_C[\text{M}][\text{Obj}(\text{M}')] = T_C[\text{M}](\text{MyType}). \]
Also,
\[ T_C[\text{inst} [\text{MT}]] = T_C[\text{inst}][\text{Obj}(\text{M}')] = T_C[\text{inst}][\text{MyType}], \]
and
\[ T_C[\text{meth}[\text{MT}][\text{SelfType}]] \]
\[ = T_C[\text{meth}][\text{Obj}(\text{M}')[\text{VisObj} (\text{IVR}'(\text{Obj}(\text{M}'),\text{M}'(\text{Obj}(\text{M}'))))]
\[ = T_C[\text{meth}][\text{MyType}][\text{SelfType}]. \]
Thus,
\[ T_C[\mathcal{C}'], T_C[\mathcal{E}] \vdash T_C[\text{inst}][\text{MyType}]: T_C[\text{IV}](\text{MyType}) \]
and
\[ T_C[\mathcal{C}'], T_C[\mathcal{E}'] \vdash T_C[\text{meth}][\text{MyType}][\text{SelfType}]: T_C[\text{M}](\text{MyType}). \]
Now
\[ T_C[\mathcal{C}'] = T_C[\mathcal{C} \cup \{M' <: M, \text{IVR}' <: \text{IV}^\text{ref}, \]
\[ \text{MT} = \text{ObjectType} \text{M'}(\text{MyType}), \]
\[ \text{SelfType} = \text{VisObjectType} (\text{IVR}'(\text{MyType}), \text{M}'(\text{MyType}))\}]
\[ = T_C[\mathcal{C}] \cup \{M' <: T_C[\text{M}], \text{IVR}' <: T_C[\text{IV}^\text{ref}]\}
\[ = T_C[\mathcal{C}] \cup \{M' <: T_C[\text{M}], \text{IVR}' <: T_C[\text{IV}^\text{ref}]\} \]
and
\[ T_C[\mathcal{E}'] = T_C[\mathcal{E} \cup \{\text{self: SelfType, close: SelfType} \rightarrow \text{MT}\}]
\[ = T_C[\mathcal{E}] \cup \{\text{self: SelfType, close: SelfType} \rightarrow \text{MyType}\}. \]
In each of the above derivations, we can replace expressions of the form \(T_C[\ldots]\) by \(T_C[\ldots]\) in the last line because the argument of \(T_C[\ldots]\) involves only type variables already in \(C\).

Recall that

\[
\text{methfun} \triangleq \lambda (\text{self}: \text{SelfType}).
\]

\[
\text{let close: SelfType} \rightarrow \exists Y. \text{VisObj}(Y, M'(\text{MyType}))
= \text{closeobj}[\text{IVR}'(\text{MyType})][M'(\text{MyType})]
\]

\[
in T_C[\text{meth}][\text{MyType}][\text{SelfType}].
\]

By the Let and Function typing rules,

\[
T_C[\mathcal{C}', T_C[\mathcal{E}]] \vdash \text{methfun}: \text{SelfType} \rightarrow T_C[\mathcal{M}](\text{MyType}).
\]

It follows from the above and the type-checking rules Function and Pair of \(\Lambda^P\) that

\[
T_C[\mathcal{C}', T_C[\mathcal{E}]] \vdash \langle T_C[\text{inst}], \text{methfun} \rangle:
T_C[\text{IV}](\text{MyType}) \times (\text{SelfType} \rightarrow T_C[\mathcal{M}](\text{MyType})).
\]

Finally, by the type-checking rule BdPolyFunc in Figure 9.9,

\[
T_C[\mathcal{C}', T_C[\mathcal{E}]] \vdash \Lambda (M' <: T_C[\mathcal{M}]). \Lambda (IV'R' <: T_C[\text{IV}']) .

\langle T_C[\text{inst}][\text{MyType}], \text{methfun} \rangle:
\forall (M' <: T_C[\mathcal{M}]). \forall (IV'R' <: T_C[\text{IV}']).
T_C[\text{IV}](\text{MyType}) \times (\text{SelfType} \rightarrow T_C[\mathcal{M}](\text{MyType})).
\]

Thus

\[
T_C[\mathcal{C}', T_C[\mathcal{E}]] \vdash T_C[\text{class} (\text{inst, meth})]: T_C[\text{ClassType}](\text{IV}, \mathcal{M})].
\]

**Subclass**  Omitted here. Similar to class.

**Message**  Suppose

\[
C, E \vdash E \leftarrow m: T(U)
\]

because

\[
C, E \vdash E: U
\]

and

\[
C \vdash U \leftarrow \# \text{ObjectType} \{ m: T(\text{MyType}) \}
\]
By Lemma 16.2.2, \( C(U) = \text{ObjectType} M(\text{MyType}) \) for some \( M \) such that
\[
C \vdash M \ll T_{\text{Func}}(MT) \cdot \| m : T(MT) \|.
\]

By Theorem 16.4.1,
\[
\mathcal{T}_C[C] \vdash \mathcal{T}_C[M] \ll : \mathcal{T}_C[T_{\text{Func}}(MT)] \cdot \| m : T(MT) \|,
\]
where \( \mathcal{T}_C[T_{\text{Func}}(MT)] \cdot \| m : T(MT) \| = \lambda(MT) \cdot \| m : \mathcal{T}_C[T(MT)] \|.

Hence for all type identifiers, \( IVR', MT' \),
\[
\mathcal{T}_C[C] \vdash IVR' \times (IVR' \rightarrow \mathcal{T}_C[M](MT')) \ll : IVR' \times (IVR' \rightarrow \| m : \mathcal{T}_C[T(MT')] \|).
\]

By induction,
\[
\mathcal{T}_C[C], \mathcal{T}_C[E] \vdash \mathcal{T}_C[E] : \mathcal{T}_C[U].
\]

Let
\[
MT = \mathcal{T}_C[U] = \mathcal{T}_C[\text{ObjectType} M(\text{MyType})] = \text{Obj} (\mathcal{T}_C[M])
= \exists Y. Y \times (Y \rightarrow \mathcal{T}_C[M](MT)).
\]

By definition,
\[
\mathcal{T}_C[E \ll n] = \text{open} \mathcal{T}_C[E] \text{ as } \langle IVR, vo \rangle \text{ in } (\text{proj}_2(\text{vo})(\text{proj}_1(\text{vo})) \cdot m
\]

By equation 16.1 and subsumption,
\[
\mathcal{T}_C[C], \mathcal{T}_C[E] \cup \{ vo : IVR \times (IVR' \rightarrow \mathcal{T}_C[M](MT')) \} \vdash
vo : IVR \times (IVR' \rightarrow \| m : \mathcal{T}_C[T(MT')] \|)
\]

By the projection typing rules and the Selection rule of \( \lambda C' \),
\[
\mathcal{T}_C[C], \mathcal{T}_C[E] \cup \{ vo : IVR \times (IVR' \rightarrow \mathcal{T}_C[M](MT')) \} \vdash
(\text{proj}_2(\text{vo})(\text{proj}_1(\text{vo})) \cdot m : \mathcal{T}_C[T](MT)).
\]

It follows from the Unpack rule that
\[
\mathcal{T}_C[C], \mathcal{T}_C[E] \vdash \text{open} \mathcal{T}_C[E] \text{ as } \langle IVR, vo \rangle \text{ in } (\text{proj}_2(\text{vo})(\text{proj}_1(\text{vo})) \cdot m : \mathcal{T}_C[T](MT)
\]

Hence
\[
\mathcal{T}_C[C], \mathcal{T}_C[E] \vdash \mathcal{T}_C[E \ll n] : \mathcal{T}_C[T(U)].
\]

The reader should be sure to understand why the match, rather than subtype, bound on \( U \) was sufficient for proving the soundness of this rule.
16.4 Soundness of translation for \( \text{MOOL} \)

**Inst Vble** Suppose

\[ C, \mathcal{E} \vdash \text{E.1: } T(V) \]

because

\[ C, \mathcal{E} \vdash \text{E: } U \]

and

\[ C \vdash U \lessdot \text{vis VisObject_Type(} \text{V} | T(\text{My_Type}) \text{), } \text{V} | U \text{)} \]

where \( C(U) = \text{VisObject_Type(IVR(My_Type), M(My_Type))} \) and \( C(V) = \text{Object_Type M(My_Type)} \).

By Lemma 16.2.3,

\[ C \vdash \text{IVR} \lessdot \text{TpFunc(MT). } \text{V} | T(\text{MT}) \text{)} \]

and hence by Theorem 16.4.1,

\[ \mathcal{T}_C[\mathcal{C}] \vdash \mathcal{T}_C[\text{IVR} \lessdot \text{TpFunc}(\text{MT}). \text{V} | T(\text{MT}) \text{)}] \]

By induction,

\[ \mathcal{T}_C[\mathcal{C}], \mathcal{T}_C[\mathcal{E}] \vdash \mathcal{T}_C[\text{VisObject_Type(IVR(My_Type), M(My_Type))}] \]

while, by definition,

\[ \mathcal{T}_C[\text{VisObject_Type(IVR(My_Type), M(My_Type))}] = \text{VisObj}(\mathcal{T}_C[\text{IVR}(\text{MT}), \mathcal{T}_C[\text{M}](\text{MT})]) = \mathcal{T}_C[\text{IVR}(\text{MT}) \times (\mathcal{T}_C[\text{IVR}(\text{MT}) \rightarrow \mathcal{T}_C[\text{M}](\text{MT}))], \]

where \( \text{MT} = \mathcal{T}_C[\mathcal{V}] = \mathcal{T}_C[\text{Object_Type M(My_Type)}]) = \text{Obj}(\mathcal{T}_C[\text{M}]). \)

By rule Proj,

\[ \mathcal{T}_C[\mathcal{C}], \mathcal{T}_C[\mathcal{E}] \vdash \text{proj}_1(\mathcal{T}_C[\text{E}]): \mathcal{T}_C[\text{IVR}(\text{MT})] \]

and thus by subsumption in \( \Lambda^P_\text{C} \),

\[ \mathcal{T}_C[\mathcal{C}], \mathcal{T}_C[\mathcal{E}] \vdash \text{proj}_1(\mathcal{T}_C[\text{E}]): \text{V} | 1: \mathcal{T}_C[\text{M}](\text{MT}) \]

By definition,

\[ \mathcal{T}_C[\text{E.1}] = \text{proj}_1(\mathcal{T}_C[\text{E}]).1 \]

Therefore, by the Selection type-checking rule of \( \Lambda^P_\text{C} \),

\[ \mathcal{T}_C[\mathcal{C}], \mathcal{T}_C[\mathcal{E}] \vdash \mathcal{T}_C[\mathcal{E.1}]: \mathcal{T}_C[\text{M}](\text{MT}) \]

and thus

\[ \mathcal{T}_C[\mathcal{C}], \mathcal{T}_C[\mathcal{E}] \vdash \mathcal{T}_C[\text{T(V)}] \]
Self Message This case is similar to sending messages to regular objects except that it is not necessary to unpack the existential.

16.5 Summary

In this chapter we designed the language MOOL by adding a MyType construct to SOOL. The type expression MyType represents the type of self inside of a class definition. As the meaning of self changes automatically when moving from a class to a subclass, so does the meaning of MyType.

Just as this automatic change of the meaning of self is extremely helpful when inheriting methods from a superclass, so is the automatic change in the meaning of MyType. In particular, it provides a way to support covariant changes to types in classes, whether they are the types of instance variables, method parameters, or method return types.

While subclasses of classes whose methods have parameters with type MyType do not generate object types that are in the subtype relation, they do generate object types that match. While the loss of subtyping might be seen to be a disadvantage, we saw that matching is often all that is needed. In particular, if type $T <\# S$ then any method of $S$ will also be in $T$, and the type of that method in $T$ will be a subtype of that in $S$.

In the next chapter we will extend MOOL with match-bounded polymorphism and show the expressiveness of the combination of MyType and matching.
17 Match-Bounded Polymorphism

In this chapter we add bounded polymorphism to \textit{MOOL}. Of course we have already done this in extending \textit{SOOL} to \textit{PSOOL}. The difference is that this time we will be more interested in introducing match-bounded polymorphism rather than polymorphism in which the bounds are expressed in terms of the subtyping relation.

We have already seen that in languages with \textit{MyType}, subclasses generate types that are in the matching relation, and thus seems a more natural relation than subtyping. We will also see that we can simplify many occurrences of F-bounded polymorphism to simple match-bounded polymorphism. Match-bounded polymorphism also has the added benefit of fitting more smoothly with subclasses than does F-bounded polymorphism.

17.1 Benefits of match-bounded polymorphism

While the matching relation does not allow values of one object type to masquerade as values of the other, as is the case with subtyping, we have seen it provides useful information on the availability and types of methods. Interestingly, it is also very useful as a constraint on type variables for polymorphism. In fact, the use of \textit{MyType} and matching will allow the possibility of dispensing with most of the uses of F-bounded polymorphism.

Recall from the last chapter the following example of F-bounded polymorphism.

\begin{verbatim}
OrderByF(t <: TopObject) = ObjectType {
    equal: t \rightarrow Boolean;
    greaterThan: t \rightarrow Boolean;
    lessThan: t \rightarrow Boolean
}
\end{verbatim}
The use of F-bounded polymorphism was needed in parameterized class BPOrderedList because the constraint on Elt was that it have equal, greaterThan, and lessThan methods that took parameters of type Elt. Thus when one of these methods is sent to an element of type Elt, it should have the same type as the receiver. Of course this should suggest to us the use of MyType.

Define

```
OrderableMT = ObjectType {
    equal: MyType → Boolean;
    greaterThan: MyType → Boolean;
    lessThan: MyType → Boolean
}
```

Unlike OrderableF, the type OrderableMT need not be parameterized. The class BPOrderedListMT is still defined with bounded polymorphism, but the bound is now expressed with the matching relation, and no F-bounded polymorphism is required.

```
class BPOrderedListMT(Elt <: OrderableMT) { ... }
```

Thus the uses of F-bounded polymorphism occurring in our earlier examples can be replaced with simple match-bounded polymorphism (bounded polymorphism in which the bound is expressed with matching) with the introduction of MyType.

The most important point to keep in mind with matching is that it provides information about the methods available and their types. Thus knowing that type Elt matches OrderableMT provides the information that Elt has at least the methods equal, greaterThan, and lessThan, and that their signatures conform to those given in OrderableMT. Most of the time, this sort of information is all that is needed in object-oriented programming. The extra information that one type can masquerade as another in all possible circumstances is not needed. We will present a more careful comparison of F-bounded polymorphism and match-bounded polymorphism after we discuss the type-checking rules and translation of this extended language.
17.2 Introducing \( \mathcal{PMOOL} \)

In Section 17.1 we discussed the benefits of using match-bounded polymorphism rather than using subtyping to restrict type variables. Adding match-bounded polymorphism to \( \mathcal{MOOL} \) to form \( \mathcal{PMOOL} \) is straightforward, though the translational semantics will be a bit tricky.

We begin by generalizing type constructor constraint systems to allow matching relations.

**Definition 17.2.1** Relations of the form \( x^e = G, \ t <\# T, \) and \( x^e <: G, \) where \( x^e \) is a type constructor variable, \( G \) is a type constructor expression of the same kind, \( t \) is a type variable, and \( T \) is a type expression are said to be.

Because matching is only defined for types, rather than constructors of higher kind, constraints of the form \( t <\# T \) will be restricted to those in which \( t \) and \( T \) are both types, and \( T \) represents an object type. A is defined as follows:

1. The empty set, \( \emptyset \), is a generalized type constructor constraint system.

2. If \( \mathcal{C} \) is a generalized type constructor constraint system, \( x^e \) is an identifier that does not appear in \( \mathcal{C} \) or \( G \), and \( G \) is a constructor expression of the same kind, then \( \mathcal{C} \cup \{ x^e = G \} \) is a generalized type constructor constraint system.

3. Let \( \mathcal{C} \) be a generalized type constructor constraint system, let \( t \) be a type identifier that does not appear in \( \mathcal{C} \), and let \( T \) either be of the form \( \text{ObjectType} \ M \), for some \( M \), or be a type identifier representing an object type with respect to \( \mathcal{C} \). Then \( \mathcal{C} \cup \{ t <\# T \} \) is a generalized type constructor constraint system.

4. If \( \mathcal{C} \) is a generalized type constructor constraint system, \( x^e \) is a type constructor identifier that does not appear in \( \mathcal{C} \), and \( G \) is a constructor expression of the same kind, then \( \mathcal{C} \cup \{ x^e <: G \} \) is a generalized type constructor constraint system.

The definition above allowed type identifiers to serve as upper bounds of matching relationships only if the upper bounds themselves represented object types. The formal definition follows:

**Definition 17.2.2** A type identifier \( t \) represents an object type with respect to \( \mathcal{C} \) if \( (t = T) \in \mathcal{C} \), \( (t <: T) \in \mathcal{C} \), or \( (t <\# T) \in \mathcal{C} \), where \( T \) is \( \text{TopObject} \), is of the form \( \text{ObjectType} \ M \) for some \( M \), or is a type identifier representing an object type with respect to \( \mathcal{C} \) with the constraint on \( t \) removed.

That is, \( t \) represents an object type if it is explicitly bounded above or equal to an object type or it is bounded above or equal to another type identifier that represents an object type.
We could loosen the definition to allow bounds, $T$, where $T$ only reduces to an object type, but that would greatly complicate our translation function later. In fact, such expressions would normally be reduced during type-checking anyway.

Now that our type constraints include matching relations, we add match-bounded polymorphic expressions and their typing rules in Figure 17.1. While these rules allow F-bounded matching, the use of matching removes the necessity for most uses requiring the bounded type variable to appear in the bound.

### 17.3 Examples and comparison with F-bounded polymorphism

A complete example of a $\mathcal{P.MOOL}$ program manipulating ordered lists is given in Figures 17.2, 17.3, and 17.4. It uses match-bounded polymorphism to create ordered lists that hold elements of any (fixed) type that matches $\text{OrderableMT}$. (The MT in $\text{OrderableMT}$ stands for $\text{MyType}$.)

The $\text{NodeType}$ type constructor takes type $T$ as a parameter to form a type for objects providing methods to set and get values of type $T$ and providing methods to access and change a next field of the same type. Class $\text{Node}$ is polymorphic, taking a type parameter $T$ that is restricted to be an object type, and a value of type $T$, generating objects of type $\text{NodeType}(T)$.

---

1. We implicitly extend our abbreviations available for $\mathcal{P.MOOL}$ programs in order to write these examples more simply.
The type constructor BPOrdListMTType takes a type parameter, T, and returns an object type with find and add methods for a list with elements of type T. The polymorphic class BPOrdListMT takes a type parameter T that is restricted to be an object type with equal, greaterThan, and lessThan methods that can be used to compare objects of the same type. The definitions of methods add and find use the comparison relations on T.

The main part of the program depends on the assumption that StringMT <# OrderableMT. It creates a new ordered list, adds a string, and then checks to see if a different string is contained in it.

In Section 4.1.4 we remarked that F-bounded polymorphism did not interact well with the subtype or subclass hierarchies in object-oriented languages. We recast the example in that section in terms of PMOOL.

Recall the examples in Section 15.1. We had parameterized type OrderableF, and parameterized class BPOrdList that takes a type parameter Elt <: OrderableF(Elt). We also had object type String such that String <: OrderableF(String).

Suppose CaseString is a subtype of String that also includes a method setUpper:Void → Void that is to convert the string to all caps. This could easily be obtained as a subclass of a class generating objects of type String. While CaseString is a subtype of String, the type CaseString does not satisfy the F-bounded constraint on BPOrdList because CaseString is a subtype of OrderableF(String) rather than being a subtype of OrderableF(CaseString).

An important feature of match-bounded polymorphism is that this problem goes away. If StringMT matches OrderableMT, i.e., its methods all have type MyType → Boolean, then adding new methods like setUpper result in types matching StringMT, and hence OrderableMT. Thus we could apply parameterized class BPOrdListMT to these extensions. This consistency with extensions of both classes and object types is an important advantage of match-bounded polymorphism over F-bounded polymorphism.

17.4 Translational semantics of PMOOL

The only new cases we must deal with in the translation of PMOOL into $\Lambda^p_2$ are those corresponding to match-bounded polymorphism. However, because we modified the definition of constraint systems, we will need to modify their translations as well.
program linkedlist;

OrderableMT = ObjectType {
    equal: MyType → Boolean;
    greaterThan: MyType → Boolean;
    lessThan: MyType → Boolean
} ;

// Type of singly linked node objects
NodeType(T) = ObjectType {
    getValue: Void → T;
    setValue: T → Void;
    getNext: Void → MyType;
    setNext: MyType → Void
} ;

// Type of ordered linked lists of T
BPOrdListMTType(T) = ObjectType {
    find: T → Boolean;
    add: Void → T
} ;

// class for singly linked nodes w/values of type T
class Node(T <# TopObject; a: T) {
    value: T := a;
    next: MyType := nil;

    function getValue(): T is
    { return self.value ; }

    function setValue(newValue: T): Void is
    { self.value := newValue ; }

    function getNext(): MyType is
    { return self.next ; }

    function setNext(newNext: MyType): Void is
    { self.next := newNext
    }
};

Figure 17.2  Linked list example in P.M.O.O.L. Part 1.
17.4 Translational semantics of $\mathcal{P}M\mathcal{OOL}$

```plaintext
class BPOrdListMT(T < OrderableMT) {
    head: NodeType(T) := nil;
    function find(match: T): Boolean is
        done: Boolean := false;
        current: NodeType(T) := head;
        while not(done) & not(current = nil) do
            done := current $\Leftarrow$ getVal() $\Leftarrow$ eq(match);
            if not(done) then
                current := current $\Leftarrow$ getNext();
        
        return (done);
    
    function add(a: T): Void is
        prev: NodeType(T); current: NodeType(T);
        newNode: NodeType(T) = new Node(T, a);
        if head = nil then
            head := newNode;
            newNode $\Leftarrow$ setNext(nil); }
        else if head $\Leftarrow$ getVal() $\Leftarrow$ ge( 
            newNode $\Leftarrow$ getVal()) then
            newNode $\Leftarrow$ setNext(head);
            head := newNode; }
        else
            prev := head;
            current := head $\Leftarrow$ getNext();
            while not(current = nil) &
                current $\Leftarrow$ getVal() $\Leftarrow$ ge( 
                    newNode $\Leftarrow$ getVal()) do
                prev := current;
                current := current $\Leftarrow$ getNext();
            if current = nil then
                prev $\Leftarrow$ setNext(newNode);
                newNode $\Leftarrow$ setNext(nil); 
            else
                newNode $\Leftarrow$ setNext(current);
                prev $\Leftarrow$ setNext(newNode); 
        } 
}
```

Figure 17.3 Linked list example in $\mathcal{P}M\mathcal{OOL}$. Part 2.
\[
\begin{align*}
// \text{Assume } \text{StringType } <\# \text{ OrderableMT} \\
aString : \text{StringMT} := "\text{First Node}";  \\
lnode : \text{NodeType(StringMT)};  \\
// \text{singly linked list}  \\
slist : \text{BPOrdListMTType(StringMT)}; \\
\{  \\
\text{slist} := \text{new (BPOrdListMT[StringMT])};  \\
\text{slist} \leftarrow \text{add("1st Node")};  \\
\text{printBool (slist} \leftarrow \text{find (aString));}  \\
\}
\end{align*}
\]

\textbf{Figure 17.4} Linked list example in \texttt{PMOOL}. Part 3.

17.4.1 Translational semantics of types

The difficulty here is that \( \lambda^\alpha \) does not include syntax for the matching relation. As a result, we must encode it using higher-order subtyping. We encode matching as higher-order subtyping using the following correspondence:

\[
\mathcal{C} \vdash \text{ObjectType } M' (\text{MyType}) <\# \text{ ObjectType M (MyType)}  \\
\text{iff} \quad \mathcal{C} \vdash M' <: M.
\]

Of course we have already been using this encoding implicitly when writing down type-checking rules and the translations for classes and subclasses.

This encoding works fine as long as the type expressions on both sides of \(<\#\) are object type expressions, but what do we do in the case where one or both are type variables? To solve this problem we will interpret a type identifier \( t \) as a type constructor identifier \( t^\Rightarrow x^\Rightarrow \) of \( \lambda^\alpha \) with kind \( x^\Rightarrow \). (We will use the same letter for the translated type identifier, even though the original has kind \( * \) while the translation has kind \( x^\Rightarrow \).) Then when we are ready to actually use the variable in a context requiring an object type, we will convert it into an object type by using \( \text{Obj}(t) \).

This brings up the unfortunate complication that we must have two ways of interpreting object types and type variables. We need one translation where both are translated as functions from types to types so that they may be used in match-bounded polymorphism. The second translation is used where we actually need these to be object types and used as types of expressions.
When we translate $\text{ForAll}(t : U). V$, the type variable $t$ and its bound $U$ will be translated as functions from types to types, while $V$ should be translated as a regular type. Thus we will need context sensitive translations of types in order to translate the extension of the language with match-bounded polymorphism.

Moreover we will need to distinguish between identifiers introduced in match-bounded polymorphic types and those occurring as formal parameters of type constructors. That is, we will translate the identifier $t$ introduced in match-bounded polymorphism of the form $\text{ForAll}(t : U). V$ differently from other type identifiers.

We distinguish between these two translations by introducing the notation $\mathcal{T}_{c}^{\mathcal{M},X}[u]$ for the translation of type variables in $X$ and object types to be $\Lambda_{c,\mathcal{M}}$ functions from types to types. It is undefined on other types. (Think of the superscript $\mathcal{M}$ as referring to “match-bounded”.)

The notation $\mathcal{T}_{c}^{X}[u]$ will represent the translation of types of $\mathcal{P} \mathcal{M} \mathcal{O} \mathcal{L}$ to types of $\Lambda_{c,\mathcal{M}}$. It is very similar to the translation function $\mathcal{T}_{c}[u]$ that we used in our earlier translations.

The translation of polymorphic types and type variables is given in Figure 17.5. Translation of types not included are as before, except with $\mathcal{T}_{c}[\ldots]$ replaced by $\mathcal{T}_{c}^{X}[\ldots]$.

At the top level, we begin translating types using $\mathcal{T}_{c}^{X}[u]$. The reader will note that the alternative translation function, $\mathcal{T}_{c}^{\mathcal{M},X}[u]$, is introduced only in the translation of polymorphic types where the new type variable is match-bound by another type. In that cases both the type variable and its bound is translated using $\mathcal{T}_{c}^{\mathcal{M},X}[\ldots]$.

We illustrate the new functions by translating the three types introduced in the program `linkedlist` in Figure 17.2. We begin by translating the type `NodeType` defined by:

\[
\text{NodeType} = \text{TpFunc}(T). \text{ObjectType} \{  
\text{getValue: Void} \rightarrow T; 
\text{setValue: T} \rightarrow \text{Void}; 
\text{getNext: Void} \rightarrow \text{MyType}; 
\text{setNext: MyType} \rightarrow \text{Void} \} 
\]

(We have given the definition in the form without abbreviations here.) We will omit the translation of methods `setValue` and `setNext` because they are similar to the two corresponding “get...” methods. By the definition
Figure 17.5 Translations of selected types and constructors of $\mathcal{P}_A$.

of $\mathcal{T}_C^X[u]$ in Figure 17.5:

- $\mathcal{T}_C^{\text{ObjectType}}[u] = \lambda t. \mathcal{T}_C^X[u]$, where $t$ is a type variable and does not occur in $C$.
- $\mathcal{T}_C^{\text{Var}}[u] = \mathcal{T}_C^X[u]$, where $t$ is a type variable and does not occur in $C$.
- $\mathcal{T}_C^{\text{ObjectType}}[\text{IVR}(\text{MyType}), M(\text{MyType})] = \text{VisObject}(\mathcal{T}_C^X[M], M(\text{MyType}))$.
- $\mathcal{T}_C^{\text{ClassType}}[\text{IVR}(\text{MyType}), M(\text{MyType})] = \forall (M' <: T_C^X[M]), \forall (\text{IVR}' <: T_C^X[I\text{nd}])$.

where

- $\text{Obj}(M) \triangleq \text{Fix}(\lambda \text{MT}. \exists Y. Y \times (Y \to M(\text{MT})))$.
- $\exists Y. Y \times (Y \to M(\text{Obj}(M)))$.
- $\text{VisObj}(\text{IVR}, M) \triangleq \text{IVR} \times (\text{IVR} \to M)$.
The type OrderableMT, which is defined by

\[
\text{OrderableMT} = \text{ObjectType} \ |
\begin{align*}
\text{equal: } & \text{MyType} \rightarrow \text{Boolean}; \\
\text{greaterThan: } & \text{MyType} \rightarrow \text{Boolean}; \\
\text{lessThan: } & \text{MyType} \rightarrow \text{Boolean} \\
\end{align*}
\]

appears as a type bound in the definition of BPOrdListMT. As a result, we will need to compute the alternate translation, \( \mathcal{T}_c^{[\text{MT}]}[\text{BPOrdListMTType}] \). Because the three methods of OrderableMT all have the same signature, we will only include the equal method in our translation.

\[
\mathcal{T}_c^{[\text{MT}]}[\text{OrderableMT}] = \mathcal{T}_c^{[\text{MT}]}[\text{ObjectType} \ |
\begin{align*}
\text{equal: } & \text{MyType} \rightarrow \text{Boolean} \\
\end{align*}
\] 

whereas

\[
\mathcal{T}_c^{[\text{MT}]}[\text{OrderableMT}] = \text{Obj}(\lambda \text{mt}. \{\text{equal: mt} \rightarrow \text{Boolean}\}).
\]

With this, we can now translate BPOrdListMTType:

\[
\text{BPOrdListMTType} = \text{TpFunc}(T). \text{ObjectType} |
\begin{align*}
\text{find: } & T \rightarrow \text{Boolean}; \\
\text{add: } & \text{Void} \rightarrow T \\
\end{align*}
\]

as

\[
\mathcal{T}_c^{[\text{MT}]}[\text{BPOrdListMTType}] = \lambda T. \mathcal{T}_c^{[\text{MT}]}[\text{ObjectType} \ |
\begin{align*}
\text{find: } & T \rightarrow \text{Boolean}; \\
\text{add: } & \text{Void} \rightarrow T \\
\end{align*}
\]
17.4.2 Translational semantics of value expressions

Not surprisingly, the extra complications of translating types will result in similar complications for translating expressions, especially those reflecting match-bounded polymorphism and applications of polymorphic functions to types. The translation for selected expressions is given in Figure 17.6. We only include those expressions where types play a visible role. Others are translated similarly to before.

Only a single translation function, \( T^{X} \), is used for value expressions, though \( T^{M,X} \) is used for translating some of the types.

Notice the difference between the translations of subtype-bounded polymorphic functions and match-bounded polymorphic functions. In the former, the bound type identifier and its bound are both translated as types, while in the latter, both are translated as functions from types to types. Similarly, there is a distinction in the translation of expressions representing the application of polymorphic functions to types. If they represent match-bounded polymorphism then the type arguments are translated as functions from types to types, while with subtype-bounded polymorphism the type arguments are translated as types.

The translation of other expressions is exactly as before except that \( T^{X} \) is used in place of \( T^{M,X} \). We will come back to this topic later, but we remark now that in a language with match-bounded polymorphism, there is little need to also support subtype-bounded polymorphism. This provides for a simpler language at little cost in expressiveness.

17.5 Soundness of the translation of \( \mathcal{PMMM} \)

In order to prove the translation of \( \mathcal{PMMM} \) is sound we must first determine how to translate type constraints. Our definition is by induction on the definition of generalized type constructor constraint systems. Let \( C' \) be a generalized type constructor constraint system and \( X \) a set of variables of \( \Lambda_{<} \). Then

1. \( T^{X}[][] = 0 \).
2. \( T^{X}[\mathcal{C} \cup \{ t = T \}] = T^{X}[[\mathcal{C}]] \).
3. \( T^{X}[\mathcal{C} \cup \{ t <: T \}] = T^{X}[[\mathcal{C}]] \cup \{ t <: \mathcal{T}^{X}[[T]] \} \).
4. \( T^{X,\{ t \} [[\mathcal{C}\cup\{t <\# T\}]} = T^{X}[[\mathcal{C}]] \cup \{ t <: T^{M,X,\{ t \}[[T]]} \}, \) where \( t \) is a type
17.5 Soundness of the translation of $\mathcal{PMOOL}$

\[ T^X_c[C, E] \vdash \text{polyFunc}(t <: T): U \text{ is Block: } \forall t: (t <: T). U \]

where $t$ has kind $\ast$.

\[ T^X_c[C, E] \vdash \text{polyFunc}(t <\# T): U \text{ is Block: } \forall t: (t <\# T). U \]

where $t$ has kind $\ast \Rightarrow \ast$.

\[ T^X_c[C, E] \vdash E[T' \vdash [T'/t] U] \]

\[ \begin{cases} \forall \xi \in X, T^X_c[C, E] \vdash E: \forall t: (t \ll T). U \end{cases} [T^X_{c \cup \xi}[T']] \]

if $E$ is match-bounded

\[ \forall \xi \in X, T^X_c[C, E] \vdash E: \forall t: (t \ll T). U \end{cases} [T^X_{c \cup \xi}[T']] \]

if $E$ is subtype-bounded

\[ T^X_c[C, E] \vdash \text{function}(id_1: T_1, \ldots, id_n: T_n): T \text{ is Block: } T \]

\[ \lambda(id_1: T^X_c[T_1], \ldots, id_n: T^X_c[T_n]). T^X_c[C, E] \vdash \text{Block: } T \]

\[ T^X_c[C, E] \vdash \{ l_1: T_1 = E_1; \ldots; l_n: T_n = E_n \} \]

\[ \{ l_1: T^X_c[T_1] = T^X_c[C, E] \vdash E_1; T_1 \}; \ldots; \]

\[ l_n: T^X_c[T_n] = T^X_c[C, E] \vdash E_n; T_n \}

\[ T^X_c[C, E] \vdash \text{Block: } T \text{ = let } l_1 = T^X_c[T_1] \text{ in } \ldots \text{ let } l_n = T^X_c[T_n] \text{ in } \]

\[ \lambda(id_1: T^X_c[U_1], \ldots, id_n: T^X_c[U_n]). T^X_c[C, E] \vdash \text{Block: } T \text{ end } \ldots \text{ end } \]

where Block is type $T_1 = T_1; \ldots; T_n = T_n$

\[ \text{const id}_1: U_1 = E_1; \ldots; \text{id}_k: U_k = E_k \}

\[ \{ S \text{ return } E \}

**Figure 17.6** Translation of $\mathcal{PMOOL}$ expressions involving polymorphism.

identifier of $\mathcal{PMOOL}$, and $t$ is a type constructor variable that has kind $\ast \Rightarrow \ast$.

The translation of $E$ is similar to that given earlier:

\[ T^X_c[E] = \{ x: T^X_c[T] \mid (x: T) \in E \} \]

Before proving the soundness theorems for this extended system we must first determine the appropriate initial value of $X$, the set of type constructor identifiers of kind $\ast \Rightarrow \ast$ to be treated as types. The rather simple solution
is to initialize $X$ to the set of all type identifiers that are declared with a match-bound in the generalized type constructor constraint system used in the corresponding typing, subtyping, or matching rule.

The following theorem corresponds to Theorem 13.1.9 and is proved in the same way.

**Theorem 17.5.1** Let $S$ and $T$ be types of $\mathcal{SOC}$ such that

$$C \vdash S <: T,$$

and let

$$X = \{ t \mid \text{there exists a } T \text{ such that } (t <\# T) \in C \}.$$

Then

$$\mathcal{T}_C^X[S] <: \mathcal{T}_C^X[T].$$

The translation of matching types results in functions from types to types that are in the subtype relation.

**Theorem 17.5.2** Let $S$ and $T$ be types of $\mathcal{SOC}$ such that

$$C \vdash S <\# T,$$

and let

$$X = \{ t \mid \text{there exists a } T \text{ such that } (t <\# T) \in C \}.$$

Then

$$\mathcal{T}_C^X[S] <: \mathcal{T}_C^X[T].$$

**Proof.** We give a proof by induction on the number of steps in the proof that $C \vdash S <\# T$. The proofs for $\text{TopObject}_{\#}$, $\text{Reflex}_{\#}$, $\text{Transitive}_{\#}$, and $\text{Type Abbrev}_{\#}$ are trivial.

Suppose

$$C \vdash \text{ObjectType } M'(\text{MyType}) <\# \text{ObjectType } M(\text{MyType})$$

because

$$C \vdash M' <: M.$$

By Theorem 17.5.1,

$$\mathcal{T}_C^X[M'] <\mathcal{T}_C^X[M].$$

But

$$\mathcal{T}_C^X[\text{ObjectType } M'(\text{MyType})] = \mathcal{T}_C^X[M'].$$
and 
\[ T^X_C[\text{ObjectType } M(\text{MyType})] = T^X_C[M], \]
so we are done.

Now we wish to show that if a type can be assigned to an expression of 
\( \mathcal{S\Omega\ell} \), then the translation of that type can be assigned to the translated 
expression.

**Theorem 17.5.3** Let \( \varepsilon \) be an expression of \( \mathcal{S\Omega\ell} \) that does not involve \( \text{nil} \), let 
\[ X = \{ \mathcal{E} | \text{there exists a } T \text{ such that } (\mathcal{E} \prec T) \in \mathcal{C} \}, \]
and suppose 
\[ \mathcal{C}, \varepsilon \vdash \varepsilon : S. \]
Then 
\[ T^X_C[\mathcal{C}], T^X_C[\varepsilon] \vdash T^X_C[\varepsilon] : T^X_C[S]. \]

**Proof.** We only show the cases for expression identifiers and for match-
bounded polymorphic expressions.

**Identifier** Suppose \( \mathcal{C}, \varepsilon \vdash \text{id}: T \) because \( (\text{id}: T) \in \mathcal{E} \). Then \( (\text{id}: T^X_C[\mathcal{C}]) \in T^X_C[\mathcal{E}], \) and hence 
\[ T^X_C[\mathcal{C}], T^X_C[\varepsilon] \vdash T^X_C[\varepsilon] : T^X_C[T]. \]

**Match-bounded polymorphism** Suppose 
\[ \mathcal{C}, \varepsilon \vdash \text{polyFunc}(\mathcal{E} \overset{\neq}{\prec} T): U \text{ is Block: } \forall \mathcal{C}, \forall \mathcal{E}, \forall \mathcal{C}. \]

because \( \mathcal{C} \cup \{ \mathcal{E} \overset{\neq}{\prec} T \}, \varepsilon \vdash \text{Block: } U. \)

Let \( \mathcal{C}' = \mathcal{C} \cup \{ \mathcal{E} \overset{\neq}{\prec} T \} \) and \( X' = X \cup \{ \mathcal{E} \}. \) By induction,
\[ T^X_C[\mathcal{C}'], T^X_C[\mathcal{E}'] \vdash T^X_C[\text{Block}]: T^X_C[U]. \]

By definition 
\[ T^X_C[\mathcal{C}'] = T^X_C[\mathcal{C}] \cup \{ \mathcal{E} \overset{<}{\prec} T^X_C[T] \}. \]

Thus,
\[ T^X_C[\mathcal{C}] \cup \{ \mathcal{E} \overset{<}{\prec} T^X_C[T] \}, T^X_C[\mathcal{E}] \vdash T^X_C[\text{Block}]: T^X_C[U], \]
and hence by the PolyFunc rule of \( \Lambda^P_C. \)
\[ T^X_C[\mathcal{C}], T^X_C[\varepsilon] \vdash \Lambda(\mathcal{E} \overset{<}{\prec} T^X_C[T]), T^X_C[\text{Block}]: \forall (\mathcal{E} \overset{<}{\prec} T^X_C[T]), T^X_C[U], \]
But
\[ T^X_C[\forall \mathcal{C}, \forall \mathcal{E}, \forall \mathcal{C}], T^X_C[\mathcal{E}] \overset{\lambda}{\vdash} \forall (\mathcal{E} \overset{<}{\prec} T^X_C[T]), T^X_C[U], \]
and
\[ \mathcal{T}_c^X[\text{polyFunc}(t \not\in T): U \text{ is Block}] \triangleq \Lambda(t : \mathcal{T}_c^{\mathcal{M},X}[[t]]).\mathcal{T}_c^X[\text{Block}]. \]

By the equivalent of Lemma 13.1.2, \( \mathcal{T}_c^X[M] = \mathcal{T}_c^X[M] \) for all \( M \). Putting this all together, we get
\[ \mathcal{T}_c^X[C], \mathcal{T}_c^X[E] + \mathcal{T}_c^X[\text{polyFunc}(t \not\in T): U \text{ is Block}]: \mathcal{T}_c^X[\text{ForAll}(t \not\in T).U], \]

**Match-bounded polymorphic application** Suppose
\[ C, E \vdash E[T' / t] : [T' / t] U, \]
because
\[ C, E \vdash E : \text{ForAll}(t \not\in T).U, \]
and
\[ C \vdash T' \not\in [T' / t] T. \]

By induction,
\[ \mathcal{T}_c^X[C], \mathcal{T}_c^X[E] + \mathcal{T}_c^X[E] : \mathcal{T}_c^X[\text{ForAll}(t \not\in T).U] \]
and
\[ \mathcal{T}_c^X[C] + \mathcal{T}_c^{\mathcal{M},X}[T'] \not\in : \mathcal{T}_c^{\mathcal{M},X}[[T' / t] \mathcal{T}]]. \]

The typing judgement translates to
\[ \mathcal{T}_c^X[C], \mathcal{T}_c^X[E] + \mathcal{T}_c^X[E] : \forall(t : \mathcal{T}_c^{\mathcal{M},X}[[t]]).\mathcal{T}_c^X[U] \]
where \( X' = X \cup \{ t \} \).

It is easy to show that \( \mathcal{T}_c^{\mathcal{M},X}[[T' / t] \mathcal{T}] = [\mathcal{T}_c^{\mathcal{M},X}[T'] / t] \mathcal{T}_c^{\mathcal{M},X}[\mathcal{T}], \) where \( t \) has kind \( * \Rightarrow * \).

It follows that
\[ \mathcal{T}_c^X[C], \mathcal{T}_c^X[E] + \mathcal{T}_c^X[E] : [\mathcal{T}_c^{\mathcal{M},X}[T'] / t] \mathcal{T}_c^{\mathcal{M},X}[\mathcal{T}], \]
and hence by rule \( \text{BdPolyApp} \) of \( \Lambda^p_c \), that
\[ \mathcal{T}_c^X[C], \mathcal{T}_c^X[E] + \mathcal{T}_c^X[E] : [\mathcal{T}_c^{\mathcal{M},X}[T'] / t] \mathcal{T}_c^{\mathcal{M},X}[U]. \]

As a result,
\[ \mathcal{T}_c^X[C], \mathcal{T}_c^X[E] + \mathcal{T}_c^X[E] : [T' / t] U. \]
17.6 Summary

In this chapter we extended $\mathcal{MOOL}$ to a language $\mathcal{PMOOL}$ that supports match-bounded polymorphism, a form of bounded polymorphism in which the bound on type parameters is expressed in terms of the matching relation rather than subtyping. As the examples in Section 17.3 illustrate, this is often exactly what is needed. While F-bounded polymorphism can also express many of the same examples, match-bounded polymorphism is a simpler construct from the point of view of programmers and fits better with subclasses in languages with MyType.

The language $\mathcal{PMOOL}$ is very rich, including MyType, subtyping and matching, and F-bounded polymorphism where the bounds may be given in terms of either subtyping or matching. In the next chapter we step back to investigate the possibility of simplifying the language while retaining similar expressiveness. In particular, we will consider the possibility of dropping subtyping from the language.
Simplifying: Dropping Subtyping for Matching

“Perfection is finally attained not when there is no longer anything to add, but when there is no longer anything to take away…”

– Antoine de Saint-Exupéry

In this chapter we provide a brief introduction to a simplified version of the language $\mathcal{P}_{MOOL}$ that arises by dropping the notion of subtype and replacing it with a notion of hash types.

While the language $\mathcal{P}_{MOOL}$ is very expressive, it may be too rich a language. The notions of subtyping and matching are very similar, differing on object types only based on whether or not $\textit{MyType}$ occurs as the type of a parameter in a method. Because both subtype and match-bounded polymorphism are available to programmers, they will have to decide which relation is most appropriate whenever defining polymorphic functions.

Moreover, while subclasses will always give rise to object types which match those of the superclass, they will not always be in the subtype relation. However, we have seen that subclasses always result in object types that match those of the superclass. Thus, if we believe the $\textit{MyType}$ construct to be useful (and we do!), it may be worth exploring whether it might be possible to have matching replace subtyping in an object-oriented language.

18.1 Can we drop subtyping?

Because it is not wise to include extraneous features in programming languages, it is worth considering whether or not one of these relations can be omitted. The language $\mathcal{P}_{SOOL}$ omitted $\textit{MyType}$ and matching, but did not have the expressiveness of $\mathcal{P}_{MOOL}$. In particular, reusable binary methods are hard to write safely. Once $\textit{MyType}$ is included in the language, matching
seems forced on the programmer. But what about dropping subtyping from the language?

At first that seems to be a rather extreme position – after all subtyping is one of the key features of object-oriented languages! However, in practice it is usually not necessary for objects of one type to be able to masquerade as another. Instead what is important is to be able to predict which methods (and signatures) are supported by objects. This is exactly what matching guarantees. If $T <\# U$ the objects of type $T$ support all of the same methods as $U$.

Virtually every use of subtype-bounded polymorphism can be replaced by match-bounded polymorphism. Moreover in section 17.3 we saw that match-bounded polymorphism is more compatible with the subclass relation (subclasses generate matching types) than is the F-bounded polymorphism that was necessary to get similar expressiveness.

What about the ability to replace the types of methods by subtypes in subclasses? It turns out that this ability is actually rarely used by programmers (helping to explain why most statically typed object-oriented languages don’t support it). In Chapter 3 we discussed a number of problematic examples that illustrated the lack of expressiveness in object-oriented languages. These included classes with a clone method, and classes with binary methods like equals or the setNext method in the Node class. These examples are handled extremely well with the use of MyType – without needing support for changing method types.

The only example not handled easily by the use of MyType was the example with CircleClass and ColorCircleClass in Section 3.2.3, where it would have been useful to change the type of an instance variable, center. There is no advantage gained by using a language with subtyping, because it would require an illegal covariant change to the type of an instance variable. Instead, this example could be handled by writing it as a class, say PolyCircleClass, that is parameterized by a type parameter that matches PointType (the type of the instance variable center). A programmer would get a usable CircleClass by applying PolyCircleClass to PointType. We could extend PolyCircleClass to a subclass, PolyColorCircleClass, and then apply that to ColorPointType to get a usable ColorCircleClass. Subtype-bounded polymorphism would also work – if PointType does not have any covariant occurrences of MyType – but matching is just as good.

Of course, this is a bit awkward, and is similar to the difficulties in using F-bounded polymorphism. One alternative is to use Beta’s virtual classes to
change the type of the center instance variable, though this would introduce a dynamic type check at run time. Recall that Eiffel allows this sort of covariant type change in instance variables, though this is the source of Eiffel’s type insecurities with existing compilers.

A statically safe alternative is to use a generalization of MyType that allows programmers to change simultaneously a mutually recursive group of types. In the paper [BV99], Bruce and Vanderwaart introduce the concept of a type group and type group extensions to group together several mutually recursive types and allow them all to be extended simultaneously. The type-checking rules for classes implementing these types are similar to those using MyType in that the type checker may only use matching information about the types in the type group when type checking the methods of the class.

However, there are examples in which subtyping seems to play a crucial role. For example, one might want to have an array of graphical objects of different types, as long as they all include at least a certain collection of methods with given signatures. In a language with subtyping, this array can be declared to hold items of a type Graphical, where all of the objects to be placed in the array have types which are subtypes of Graphical. Because of subtyping, elements of subtypes can be assigned to components of the array expecting elements of type Graphical. This would not be possible if we only had matching.

Similarly we might want to define a method draw that takes a parameter that supports at least the methods of Graphical. This would normally be done by declaring:

```
  draw: Graphical -> Void
```

Then draw can be applied to any object with a type that is a subtype of Graphical.

Without the use of subtyping, this is more difficult. One possibility is to change the signature of the method to:

```
  pdraw: ForAll(G<# Graphical). G -> Void
```

This method requires the user to pass the type of the parameter as well as the parameter itself. This is more painful for the programmer. For example, if s is an object of type Square, where Square <# Graphical, the programmer would need to write pdraw[Square](x), rather than just draw(x), to invoke the function. It is clearly easier to use subtyping rather than requiring the programmer to pass both the type and regular parameter.
We do note that if Graphical has a binary method (i.e., a method with a parameter of type MyType), then the more verbose signature is actually more flexible, because Graphical would have no subtypes, but does have many possible matching types.

The problem is worse for the array described above or variables that are intended to hold different types of values, because in the absence of subtyping there is no way of using type parameters to specify a variable that can hold values of different types. Without subtyping, variables can only contain elements with exactly their declared type.

Thus it appears that we cannot just drop subtyping and replace it with matching. We need a way for programmers to specify that an identifier can have a value corresponding to a range of types rather than just a single type.

In the rest of this chapter we provide a brief sketch of a language, N'ool, with the following properties. The language will contain a MyType construct and support match-bounded polymorphism. Subtyping will not be a part of the language, but we will introduce a new type constructor for “hash” types. The combination of the weaker relation of matching and hash types will allow us to emulate the flexibility provided by subtyping.

### 18.2 Introducing hash types

In a traditional object-oriented language, a type declaration, $x : T$, indicates that the value of $x$ may be of type $T$ or any subtype of $T$. The subsumption type-checking rule provides the formal justification for this behavior.

While it is often convenient to have this flexibility, there are times when it is useful to be guaranteed that an expression has a fixed exact type as opposed to knowing the type up to subtyping. This is especially the case with binary methods, where one often wishes to guarantee that the receiver and parameter of the method are of the exact same type. For example, it takes only a little thought to realize it would be disastrous to try to build a linked list where some of the nodes are singly linked and others are doubly linked. This is one reason why it is important that doubly linked nodes are not allowed to be defined as subclasses of singly linked nodes in languages where subclasses always result in subtypes.

In order to provide the programmer with the capability to decide whether or not a type declaration should restrict an identifier to represent only a single type or a range of types, we introduce the following notations in N'ool. We write
18.2  Introducing hash types

\[ x : T \]

to indicate that the value of \( x \) must be of type \( T \), but not a subtype of \( T \). We write

\[ x : #T \]

to indicate that the value of \( x \) must be of some type \( T' \) such that \( T' \not<# T \).

We call types of the form \( #T \), hash types. Because matching is only defined on object types, only object types (including type variables representing object types) can be used in constructing hash types. For example, one may not use hash types with function types.

Here is a simple example of the use of hash types. Suppose we wish to define a node class (similar to that in Figure 16.3) that holds items from any object type matching \( \text{String} \). Then we could declare the instance variable:

\[ \text{value: #String} \]

Because declaring a variable or formal parameter to have a hash type \( T \) allows it to hold values of all types matching \( T \), items with hash types are used in a very similar way as regular variables or formal parameters in languages with subtyping.

While hash types add expressiveness to \( NCOOL \), there are some restrictions on their use that are necessary to preserve type safety. We illustrate the possible problems with a familiar example.

Recall the definitions of \( \text{NodeType} \) and \( \text{DoubleNodeType} \) in Figure 16.5. While \( \text{DoubleNodeType} <# \text{NodeType} \), they are not subtypes. The classes \( \text{Node} \) and \( \text{DoubleNode} \) generate objects of those types.

In anticipation of the discussion below, we repeat method \( \text{setNext} \) from \( \text{DoubleNode} \) below:

\[
\text{function setNext(newNext: MyType): Void is}
\{ \\
    \text{super.setNext(newNext);}
    \text{newNext }\leftarrow\text{setPrevious(close(self))}
\}
\]

Recall that method \( \text{setPrevious} \) was introduced in \( \text{DoubleNodeType} \) but was not contained in \( \text{NodeType} \).

Suppose that \( n1, n2: \#\text{NodeType} \). Then their values could either be of type \( \text{NodeType} \) or of type \( \text{DoubleNodeType} \) (or potentially any other extension of \( \text{NodeType} \)). Suppose we wished to attach \( n2 \) as the “next” element from \( n1 \). That is, we would like to write the statement:
n1 $\leftarrow$ setNext(n2)

While this seems quite plausible, there are problems. Suppose that at run time, n1 is actually an object of type DoubleNodeType, while n2 is of type NodeType. If n1 is generated by class DoubleNode, execution of the above code will result in a type error. In particular, execution of the code for setNext in DoubleNode results in sending the message setPrevious to n2, an object of type NodeType that does not have a method with that name.

This is essentially the same argument we gave back in Section 4.2 to show that extensions of object types with methods having parameters of type MyType (or in Eiffel’s case, like Current), did not result in subtypes. We must prevent this method invocation from being legal if we wish to have a statically type-safe language.

If we think more carefully about what is going on, we can see that the problem is that setNext requires a parameter that has exactly type MyType. However, because we do not know the exact type of n1, we cannot know at type-checking time what MyType is supposed to represent!

It might seem that changing the signature of setNext so that it takes a parameter of type #MyType might help, but changing that and the type of the instance variable next to #MyType do not help, as the same counterexample shows this is not type safe. As before, in order to understand whether a parameter to setNext is legal, we must know the meaning of MyType at the time of type checking. In this example, we definitely do not know the meaning of MyType if all that we know is that the type of n1 is #NodeType.

There is no difficulty in determining what is legal if we know the exact type of the receiver, n1, even in the case where we make the parameter type of setNext and the type of instance variable next be #MyType.1 Suppose the parameter type is #MyType. If n1:NodeType then actual parameters for n1 $\leftarrow$ setnext of type #NodeType, NodeType, and DoubleNodeType all work without type problems. If n1 has type (exactly) DoubleNodeType, then actual parameters of setNext with type #DoubleNodeType or DoubleNodeType both work without difficulty. Of course, the use of an actual parameter of type NodeType will not be type safe. The use of an actual parameter of type #NodeType also should not type check because it may not be safe.

---

1. Actually, while there are no type problems, we will find the class is not very usable if the type of next and the associated method parameters and return types are declared to have type #MyType. If we get a node as a result of sending getNext to another node, we lose the exact type and hence can’t send setNext messages to it.
18.2 Introducing hash types

As a general rule, we will find the use of MyType very handy whenever we have a homogeneous collection of values to be dealt with. However, MyType is rarely appropriate when we have a heterogeneous collection of values. Thus MyType is very handy in the definition of singly or doubly linked nodes, because lists are composed of either homogeneous collections of singly linked nodes or homogeneous collections of doubly linked nodes.\(^2\)

We can define a class for binary search trees as follows:

```java
class BinarySearchTree(T <: OrderableMT) {
    function insert(T): Void is ...;
    function find(T): Boolean is ...;
    function remove(T): Boolean is ...
}
```

An object generated from `BinarySearchTree(U)` will represent a binary search tree in which all elements have exactly type \( U \). Thus an object generated from `BinarySearchTree(U)` contains a homogeneous collection of elements.

However, suppose we wished to define a parameterized binary search tree, all of whose elements were of any type matching a type \( U \). We would proceed as follows.

First recall type constructor `OrderableF` from Section 15.1. We modify its methods to take parameters of type \( #T \):

```java
OrderableM(T) = ObjectType {  
    equal: #T -> Boolean;  
    greaterThan: #T -> Boolean;  
    lessThan: #T -> Boolean  
}
```

Objects of type `OrderableM(T)` are able to compare themselves with objects of any type matching \( T \).

Then we can use F-bounded matching to write:

```java
class HeteroBinarySearchTree(T <: OrderableM(T)) {  
    function insert(#T): Void is ...;
    function find(#T): Boolean is ...;
    function remove(#T): Boolean is ...
}
```

\(^2\) Of course the values stored in the list can be specified to be either homogeneous or heterogeneous. It is the nodes that we are concerned with here!
If $U < \# \text{OrderableM}(U)$, objects generated from class HeteroBinarySearchTree$(U)$ represent binary search trees containing items of any type matching $U$. It is important that $U < \# \text{OrderableM}(U)$ because this way an object with type $V$ matching $U$ can be compared with an object with any other type $W$ that matches $T$. Notice that it makes no sense to use $\text{MyType}$ in any of the types associated with these heterogeneous data structures because we would then be unable to compare elements of type $V$ with elements of type $W$.

Thus while objects with parameters of type $\text{MyType}$ do not work well in heterogeneous collections, $\text{MyType}$ is also typically not the correct modeling choice in expressing what must be done. Instead, $\text{MyType}$ is most useful in working with homogeneous collections. However, matching continues to be useful in both situations.

### 18.3 Type-checking rules

We make no changes to the syntax of the language from $\text{PMOOL}$ aside from dropping subtype-bounded polymorphic expressions. We also make no changes to the definition of static type assignments, $\mathcal{E}$. We simplify the definition of type constraints from Definition 17.2.1 by no longer including subtype constraints in $C$.

**Definition 18.3.1** Relations of the form $x^c = G$ and $t < \# T$, where $x^c$ is a type constructor variable, $G$ is a type constructor expression of the same kind, $t$ is a type variable, and $T$ is a type expression are said to be generalized type constructor constraints. Because matching is only defined for types, rather than constructors of higher kind, constraints of the form $t < \# T$ will be restricted to those in which $t$ and $T$ are both types, and $T$ represents an object type. A generalized type constructor constraint system for $\text{PMOOL}$ is defined as follows:

1. The empty set, $\emptyset$, is a generalized type constructor constraint system.

2. If $C$ is a generalized type constructor constraint system, $x^c$ is an identifier that does not appear in $C$ or $G$, and $G$ is a constructor expression of the same kind, then $C \cup \{x^c = G\}$ is a generalized type constructor constraint system.

3. Let $C$ be a generalized type constructor constraint system, let $t$ be a type identifier that does not appear in $C$, and let $T$ either be of the form $\text{ObjectType} M$, for some $M$, or be a type identifier representing an object type with respect to $C$. Then $C \cup \{t < \# T\}$ is a generalized type constructor constraint system.
We also simplify the definition of when a type identifier represents an object type by no longer considering bounds involving subtypes.

**Definition 18.3.2** A type identifier $t$ represents an object type with respect to $C$ if $(t = T) \in C$ or $(t <\# T) \in C$, where $T$ is $\text{TopObject}$, is of the form $\text{ObjectType} M$ for some $M$, or is a type identifier representing an object type with respect to $C$ with the constraint on $t$ removed.

Because we will no longer allow the subtyping of method signatures in matching types, we can get along with the much simpler notion of extension.

**Definition 18.3.3**

1. Let $M$ and $N$ be record types. Then $M$ extends $N$, written $M \prec N$, if and only if every label $l$ of $N$ is also included in $M$ and the types associated with corresponding labels in $M$ and $N$ are exactly the same.

2. Let $F$ and $G$ be type constructors with kind $\star \prec \star$. Then $F \prec G$ iff for all types $t$, $F(t)$ and $G(t)$ are record types such that $F(t) \prec G(t)$.

As usual, the matching relation $(<\#)$ is only defined on type expressions that represent object types. The matching rules are given in Figure 18.1. The main difference between these rules and the rules for $\text{MOOL}$ given in Figure 16.7 is in the hypothesis for rule $\text{Object} _{\#}$. Now the types of corresponding methods in matching object types must be identical (up to the implied change of meaning in $\text{MyType}$ in different contexts).

Type-checking rules for $\text{MOOL}$ are almost identical to those for $\mathcal{P} \text{MOOL}$. However, occurrences of subtyping in the type-checking rules are replaced by extension. We include the revised rules for classes and subclasses in Figure 18.2 as an example.

The rules in Figure 18.3 for message sending are the same as before when the receiver is an exact type, but are restricted for hash types.

Messages corresponding to binary methods may not be sent to expressions with hash types. When (non-binary) messages are sent to expressions with hash types, the resulting type is found by replacing all occurrences of $\text{MyType}$ by the hash type. Thus, if the $\text{clone}$ method of type $T$ has signature $\text{Void} \rightarrow \text{MyType}$ and $E$ is an expression of type $\#T$, then the type of $E \Leftarrow \text{clone}()$ is $\#T$, as expected.

There are no changes to the type-checking rules for accessing instance variables or sending messages to visible objects because neither involves object types.

Of course the subsumption rule for subtypes goes away, but it is replaced by rules involving hash types. The $\#\text{Subsumption}$ rule allows one to treat an
expression of type $\#T$ as though it had the type $\#U$ if $T <\# U$. The weakening rule allows one to treat an expression with an exact type as though it had the corresponding hash type.

The linkedlist program from Figures 17.2, 17.3, and 17.4 of the last chapter is an example of a legal $\mathcal{NOOL}$ program. In fact nearly all of the $\mathcal{SOOL}, \mathcal{PSOOL}, \mathcal{MOOL},$ and $\mathcal{PMOOL}$ programs exhibited in this text are legal $\mathcal{NOOL}$ programs – or would be so by simply converting certain object types to hash types. As mentioned earlier, programmers tend not to use the full strength of subtyping as much as might be expected, even when subclasses are being defined. As a result matching and hash types can be used to give similar expressiveness.

In fact, if the type system of a language like $\mathcal{SOOL}$ were to use an invariant type system like Java, then there would be a simple translation from type-safe programs of $\mathcal{SOOL}$ to type-safe programs of $\mathcal{NOOL}$, where the translation simply replaces occurrences of object types with their hashed versions. Thus the use of hash types actually gives a language which properly extends languages with type systems like Java’s.
18.3 Type-checking rules

\[
\begin{align*}
\text{Class} & \quad C', \mathcal{E} \vdash \text{inst}[\text{MT}], \\
& \quad C', \mathcal{E} \vdash \text{meth}[\text{MT}][\text{SelfType}] : M(\text{MT}) \\
& \quad C', \mathcal{E} \vdash \text{class}(\text{inst}[\text{MyType}], \text{meth}[\text{MyType}][\text{SelfType}]) : \\
& \quad \quad \quad \quad \text{ClassType}(\text{IV}(\text{MyType}), M(\text{MyType}))
\end{align*}
\]

where

- \( C' = C \cup \{ M' \prec M, \text{IVR}' \prec \text{IV}^{ref}, \text{MT} = \text{ObjectType} M'(\text{MyType}), \text{SelfType} = \text{VisObjectType} (\text{IVR}'(\text{MyType}), M'(\text{MyType})) \}, \)
- \( \mathcal{E}' = \mathcal{E} \cup \{ \text{self:}\text{SelfType}, \text{close:}\text{SelfType} \to \text{MT} \}. \)
- None of \( M', \text{IVR}', \text{MT}, \text{or SelfType} \) occurs free in \( \text{inst}, \text{meth}, \text{IV}, M, C, \) or \( \mathcal{E} \).

\[
\begin{align*}
\text{Subclass} & \quad C, \mathcal{E} \vdash \text{E : ClassType}(\text{IV}_{\text{sup}}(\text{MyType}), M_{\text{sup}}(\text{MyType})), \\
& \quad C', \mathcal{E} \vdash \text{inst}[\text{MT}], \\
& \quad C', \mathcal{E} \vdash \text{meth}[\text{MT}][\text{SelfType}] : M_{\text{sub}}(\text{MyType}), \\
& \quad C' \vdash T_{ij} \prec T_{ij}, \text{for} 1 \leq j \leq m \\
& \quad C, \mathcal{E} \vdash \text{class inherits E modifies} l_{i_1}, ..., l_{i_m} : \\
& \quad \quad \quad \quad \text{ClassType}(\text{IV}(\text{MyType}), M(\text{MyType}))
\end{align*}
\]

where

- \( \text{IV} = \text{IV}_{\text{sup}} \oplus \text{IV}_{\text{sub}} \) and \( M = M_{\text{sup}} \oplus M_{\text{sub}}, \)
- There is no overlap in the labels occurring in \( \text{IV}_{\text{sup}} \) and \( \text{IV}_{\text{sub}}, \) and
- The overlapping labels in \( M_{\text{sup}} \) and \( M_{\text{sub}} \) are exactly \( l_{i_1}, ..., l_{i_m}, \)
- The type of \( l_{i_j} \) in \( M_{\text{sub}}(\text{MT}) \) is \( T_{ij}', \) while the corresponding type in \( M_{\text{sup}}(\text{MT}) \) is \( T_{ij}. \)
- \( C' = C \cup \{ M' \prec M, \text{IVR}' \prec \text{IV}^{ref}, \text{MT} = \text{ObjectType} M'(\text{MyType}), \text{SelfType} = \text{VisObjectType} (\text{IVR}'(\text{MyType}), M'(\text{MyType})) \}, \)
- \( \mathcal{E}' = \mathcal{E} \cup \{ \text{self:}\text{SelfType}, \text{close:}\text{SelfType} \to \text{MT} \}. \)
- None of \( M', \text{IVR}', \text{MT}, \text{or SelfType} \) occurs free in \( \text{inst}, \text{meth}, \text{IV}, M, C, \) or \( \mathcal{E} \).

Figure 18.2 Typing rules for classes and subclasses in \( \mathcal{N} \mathcal{O} \mathcal{O} \mathcal{L}. \)
18.4 An informal semantics of hash types

Because \( \mathcal{N}OOL \) does not support subtyping, it would make more sense to provide a translation to \( \lambda^p_{U} \) than to \( \lambda^p_{U^c} \). However, this would require more space than we would like to devote here. (See [BFP97] for an alternative approach to the semantics of a language, \( COOM \), that is very similar to \( \mathcal{N}OOL \).) However, we do wish to explain hash types in terms of existential types.

We can understand the type \#T as an abbreviation for \( \exists t (t \triangleleft\# T).t \). In words, an expression has type \#T iff it has a type \( t \), where \( t \triangleleft\# T \). While we did not include existential types, let alone match-bound existential types, in any of our object-oriented languages, the reader should be able to understand their use in explaining the meaning of hash types.

With this understanding of the translation of hash types, we can explain why the \#Subsumption and Weakening rules are true. If the value of \( \mathcal{E} \) has type \( \exists t (t \triangleleft\# T).t \), then it is easy to unpack the value and repack it with type \( \exists (t \triangleleft\# U).t \) as long as \( T \triangleleft\# U \), validating the \#Subsumption rule. Similarly, if \( \mathcal{E} \) has type \( T \), then it is easy to pack it to have type \( \exists (t \triangleleft\# T).t \).

The interesting thing about this understanding of these rules is that use of these two rules has a semantic effect on the values of the expressions. However the original value is still available unchanged.
18.5 Summary

In this section we have given a brief description of a language $\mathcal{N}OOL$ which results (roughly) from replacing uses of subtyping in $\mathcal{P,MOOL}$ by matching and newly introduced hash types. The purpose of the design of this new language was to provide a simpler language. In particular, we desired to avoid the confusion of supporting two very similar relations on object types, subtyping and matching. More details, including a proof of type safety based on an operational semantics, may be found in the paper [BFP97].

Though we did not have space to include many examples, we assert that the use of $\mathcal{N}OOL$ allows the programmer more expressiveness than working in a language without $\text{MyType}$. At this point, that is only a claim rather than a statement of fact. Further experience with languages like $\mathcal{N}OOL$ will be necessary in order to evaluate this assertion. The language $\text{LOOJ}$ (referred to as Rupiah in [Bur98, Fos01]), is an extension of Java using similar ideas.
Historical Notes and References for Section IV

As we noted earlier, bounded polymorphism was introduced by Cardelli and Wegner. F-bounded polymorphism was introduced when it was realized that regular bounded polymorphism was not adequate to express many of the restrictions on polymorphism required in object-oriented languages, particularly when the bounds on the parameter identifiers involved binary methods. Eiffel appears to be the first commercial language to provide formal support for bounded polymorphism.

The notion of an explicit type identifier, MyType, representing the type of self appears to have first arisen in Trellis/Owl [SCB+86]. As we saw earlier, Eiffel’s use of anchored types provides an even more general construct. One of the first theoretical papers to discuss MyType, and the one that most influenced the material presented here, was the paper, “Inheritance is not subtyping” [CHC90], by Cook et al.

Mitchell’s object-based calculus, presented in [Mit90], also includes a way of referring to the type of self, though a bound variable rather than a keyword is used. Later work by Fisher et al. [FHM94] continued development of this calculus.

The language Emerald [BH91, BHJ+87] also allows references to the type of self in method signatures via a bound local name for the type. Emerald provides constrained parametric polymorphism where the constraint on type variables is given via a “such that” clause similar to the “where” clauses in Ada that allow the programmer to specify the names and signatures of methods that the type variable must support. However, Emerald also allows the use of named types in these clauses with a syntax similar to F-bounded polymorphism.
The languages Theta [DGLM94, DGLM95] and PolyJ [MBL97] use “where” clauses to constrain type variables. Constraints of this sort are essentially equivalent to match-bounded polymorphism. The main differences involve whether one has to write out all of the methods one wishes to require or just has to write out the name of a type containing those methods.

The discussion of \texttt{MyType} in Chapters 16 and 17 is based on the development and analysis of the languages TOOPLE [Bru94, BCD+93, BCK94], TOIL [vG93], and PolyTOIL [BSvG95, BFSvG01]. These languages were designed by Bruce and his students, extending the ideas of [CHC90]. The language \texttt{MOOL} in this book is very similar to TOIL, while \texttt{PMOOL} is similar to PolyTOIL.

Eifrig et al. [ESTZ94b, ESTZ94a] also presented a semantics for an imperative object-oriented language containing a \texttt{MyType} construct. As noted earlier, the language Strongtalk [BG93], a typed version of Smalltalk, also involves a \texttt{MyType} construct (called \texttt{Self} there). Its type system is based on the type rules for TOOPLE.

The definition of matching as originally given in [Bru94] was based on F-bounded polymorphism. Abadi and Cardelli, in “On subtyping and matching” [AC95b], argue that higher-order subtyping is more general than F-bounded polymorphism and serves as a better interpretation for the matching relation introduced in Chapter 16. We have followed their lead in our presentation.

The language Objective ML also supports the notion of MyType by allowing the use of a local bound type variable to stand for the type of \texttt{self}. The use of row variables in object types provides a mechanism for the expression of a matching relation.

Bono and Bugliesi, in their paper, “Matching for the lambda calculus of objects”, discuss replacing polymorphism over row variables in Mitchell’s object calculus by matching and implicit match-bound polymorphism, carefully examining the connections between these schemes.

After completing the design of PolyTOIL, Bruce was concerned that it was confusing to have such similar relations as subtyping and matching coexisting in a language. As a result he and his coworkers began looking at the design of a language, \texttt{LOOM} [BFP97], with matching only. Gawek and Matthes [GM96] reported an interesting study that showed that when programmers worked with a language supporting both subtype-bounded polymorphism and match-bounded polymorphism, they had difficulty determining which to use. While PolyTOIL only supported match-bound polymorphism at the source program level, these results strengthened the appeal
More recent work on $\mathcal{L}O\mathcal{O}\mathcal{M}$ has included the addition of a module system [Pet96, BPV98] to the language in order to provide more refined information hiding facilities. Work in another direction has included the design and implementation of an extension to Java, now called $\mathcal{L}O\mathcal{O}J$ [Bur98, Fos01], that adds $\texttt{MyType}$ (called $\texttt{ThisType}$ in $\mathcal{L}O\mathcal{O}J$), match-bounded polymorphism, and exact types to Java. Finally, Bruce and Vanderwaart [BV99] present the design and translational semantics of a mutually recursive generalization of $\texttt{MyType}$ that supports the simultaneous specializations of mutually recursive types, adding expressiveness similar to that of virtual classes.
Bibliography


[Joy98] Bill Joy. Re: Do parametric types beat virtual types? Message to Java-
genericity electronic mail list, October, 1998.


[KLM94] Dinesh Katiyar, David Luckham, and John Mitchell. A type system for pro-
totyping languages. In 21st ACM Symp. Principles of Programming Languages, pages

[KMMPN87] Bent Bruun Kristensen, Ole Lehrmann Madsen, Birger Moller-Pedersen,
and Kristen Nygaard. The Beta programming language. In Bruce Shriver and
Peter Wegner, editors, Research Directions in Object-Oriented Programming, pages 7–

[KPS93] Dexter Kozen, Jens Palsberg, and Michael I. Schwartzbach. Efficient recur-

1978.

[KR94] Samuel N. Kamin and Uday S. Reddy. Two semantic models of object-oriented
languages. In Carl A. Gunter and John C. Mitchell, editors, Theoretical Aspects of


[Lan65] P.J. Landin. A correspondence between algol 60 and church’s lambda nota-


[Lou93] Kenneth C. Louden. Programming Languages: Principles and Practice. PWS,
1993.

[LP98] Harry R. Lewis and Christos H. Papadimitriou. Elements of the Theory of Com-


[MBL97] Andrew C. Myers, Joseph A. Bank, and Barbara Liskov. Parameterized types
for Java. In ACM Symposium on the Principles of Programming Languages, pages 132–

[McC79] N. McCracken. An Investigation of a Programming Language with a Polymorphic


Index

Bono, Viviana, 364
Bruce, Kim, 115, 116, 284–286, 364
Bugliesi, Michele, 364

C, 4
C++, 3, 6, 9, 12–14, 17–20, 23, 24,
   26–28, 31, 34, 35, 37, 39, 48–52, 63,
   64, 68, 84, 86, 104, 108, 109, 111,
   114, 115, 135, 178, 194, 201, 222,
   268, 272, 273, 277, 280, 286, 299,
   300
calculus
   lambda, 98, 119, 167, 316
      bounded polymorphic, 156, 168,
      286
      higher-order polymorphic, 284
      polymorphic, 141–166, 168, 285
      second-order, 168
      typed, 15, 120–132, 167, 222
      untyped, 167, 168
object, 13, 98, 285, 286
Cardelli, Luca, 13, 58, 72, 98, 116, 168,
   169, 258, 285, 363, 364
cast, 34, 36, 37, 39, 41, 44, 45, 47, 55
Catagna, Giuseppe, 102, 115
Cecil, 12, 100, 102
Chambers, Craig, 115
Church, Alonzo, 167
class, 6, 18, 20, 95, 113, 173, 176, 179,
   180, 212, 225
abstract, 6
defered, 6
derived, 22
parameterized, 50
type-checking, 190
virtual, 104, 114, 115, 365
clone, 36
CLOS, 100, 102, 114
close, 181, 322
closeobj, 206, 275, 322
Clu, 5, 52
costraint
type constructor, 311
costraint system
generalized type constructor, 333, 356
type, 185, 226, 240
type constructor, 311, 319
costraint system, 158
costraints
generalized type constructor, 333, 356
simple type, 158
costructor, 51
covariant, 27, 158, 160
Cook, William, 110, 114, 116, 363
covariant, 26, 158, 160, 161
Curien, Pierre-Louis, 286
delegate, 96
Drossopoulou, Sophia, 287
Eiffel, 3, 6, 10, 12, 14, 34–36, 46, 49, 50, 52, 60, 61, 63, 67–70, 84, 87, 89, 101, 110, 111, 114, 162, 237, 277, 281, 297, 299, 303, 306, 351, 354, 363
Emerald, 12, 114, 116, 168, 303, 363
environment
static type, 125, 184
errors
forbidden, 258
trapped and untrapped, 258
expression
null, 134
case, 133
open, 154
pack, 153
type constructor, 143
type, of lambda calculus, 123
typed lambda calculus, 125
val, 134
expressions
let
type, 147
extend, 96
extends, 357
extension, 25
Featherweight Java, 287
field, 17
Fisher, Kathleen, 110, 285, 286, 363
Fix, 150
fixed point, 148
fixed-point operators, 147
FORTRAN, 4
function
polymorphic, 145
recursive, 147
generator, 148
Ghelli, Giorgio, 286
Girard, J.-Y., 168
GJ, 14, 49, 51, 56, 58–60, 68–70, 104, 115
Gunter, Carl, 284
Haskell, 4, 259
hiding
information, 6, 7, 15, 16, 86, 92, 101, 104, 113, 120, 140, 141, 151, 153, 168, 201, 263, 272, 275, 279, 280, 285, 365
identifier
bound, 125
<table>
<thead>
<tr>
<th>Index</th>
<th>381</th>
</tr>
</thead>
<tbody>
<tr>
<td>free, 124</td>
<td></td>
</tr>
<tr>
<td>implementation, 5</td>
<td></td>
</tr>
</tbody>
</table>
nil, 18, 179, 180, 188, 199, 200, 234, 261, 263, 266–271, 279
NOOL, 101, 352, 353, 356–361
normal form, 132
Oberon, 114
object, 6, 17
Object Pascal, 9, 12, 34, 35, 37, 114, 300
Objective C, 114
Objective CAML, 114, 287
Objective ML, 287, 364
Palsberg, Jens, 285
parameterized class, 52
partial functions, 149
Pascal, 4
Petersen, Leaf, 286
Pierce, Benjamin, 169, 284, 285
Pizza, 115
Plotkin, Gordon, 168
PolyJ, 364
polymorphism
bounded, 57, 61, 168, 342, 350
parametric, 50, 141
subtype, 25, 50
PolyTOIL, 285, 364
pre-expression, 123, 135, 157
constructor, 142
private, 272
protected, 272
PSOOL, 104, 226, 291–297, 299, 300, 310, 311, 331, 335, 349, 358
public, 272
r-value, 77, 78, 80, 134
Reddy, Uday, 283
reduction rules, 130
references, 36
Reppy, John, 286, 287
represents an object type, 333, 357
Reynolds, John, 116, 168
Riecke, Jon, 287
rule
α, 130
β, 130
δ, 131
η, 131
congruence, 143, 146, 151, 292
subsumption, 71
rules
computation, 128
subtyping, 158, 194, 293
type-checking, 101, 124–126, 135, 146, 155, 158, 162, 176, 184, 187, 191, 193, 226, 263, 294, 300, 314, 334, 357
Sather, 14, 110, 111, 114
Scheme, 4, 8, 121, 259
Schwartzbach, Michael, 285
Scott, Dana, 168
Self language, 12, 115
SelfType, 303
semantics
axiomatic, 14
denotational, 14
operational, 14
sharing, 17
translational, 203, 207, 229, 296, 338
sharing, 36
signature, 5, 6, 20, 22, 27, 28, 31, 86,
<table>
<thead>
<tr>
<th>Index</th>
<th>383</th>
</tr>
</thead>
<tbody>
<tr>
<td>96, 99, 100, 102, 108, 176, 258, 278, 300, 301, 309, 310, 317, 332, 341, 350, 351, 357, 363</td>
<td></td>
</tr>
<tr>
<td>Simula 67, 3, 12, 14, 34, 104, 111, 113, 114</td>
<td></td>
</tr>
<tr>
<td>Smalltalk, 3, 9, 12, 14, 17–19, 34, 36, 63, 68, 110, 111, 113, 116, 258, 259, 364</td>
<td></td>
</tr>
<tr>
<td>sound translation, 234, 239, 243, 257, 342</td>
<td></td>
</tr>
<tr>
<td>specification, 5</td>
<td></td>
</tr>
<tr>
<td>Strachey, Christopher, 167</td>
<td></td>
</tr>
<tr>
<td>strongly checked, 258</td>
<td></td>
</tr>
<tr>
<td>Strongtalk, 364</td>
<td></td>
</tr>
<tr>
<td>structural subtyping, 25</td>
<td></td>
</tr>
<tr>
<td>subclass, 22, 179, 180, 218</td>
<td></td>
</tr>
<tr>
<td>subject reduction, 131</td>
<td></td>
</tr>
<tr>
<td>substitution, 128</td>
<td></td>
</tr>
<tr>
<td>subsumption, 163, 195, 352, 357, 360</td>
<td></td>
</tr>
<tr>
<td>subtype, 24, 71, 156, 306</td>
<td></td>
</tr>
<tr>
<td>breadth, 74</td>
<td></td>
</tr>
<tr>
<td>depth, 74</td>
<td></td>
</tr>
<tr>
<td>subtyping, 25, 33, 72, 116, 156, 267, 350, 364</td>
<td></td>
</tr>
<tr>
<td>array, 80</td>
<td></td>
</tr>
<tr>
<td>bounded polymorphic function, 160</td>
<td></td>
</tr>
<tr>
<td>breadth, 160</td>
<td></td>
</tr>
<tr>
<td>class, 85</td>
<td></td>
</tr>
<tr>
<td>decidability, 169</td>
<td></td>
</tr>
<tr>
<td>function, 75, 158</td>
<td></td>
</tr>
<tr>
<td>higher-order, 338, 364</td>
<td></td>
</tr>
<tr>
<td>Java, 108</td>
<td></td>
</tr>
<tr>
<td>Java array, 81</td>
<td></td>
</tr>
<tr>
<td>loss of with MyType, 308</td>
<td></td>
</tr>
<tr>
<td>object, 83, 311</td>
<td></td>
</tr>
<tr>
<td>polymorphic function, 160</td>
<td></td>
</tr>
<tr>
<td>polymorphic lambda calculus, 156</td>
<td></td>
</tr>
<tr>
<td>product, 160</td>
<td></td>
</tr>
<tr>
<td>record, 74, 158</td>
<td></td>
</tr>
<tr>
<td>recursive type, 161</td>
<td></td>
</tr>
<tr>
<td>reference, 78</td>
<td></td>
</tr>
<tr>
<td>sum, 160</td>
<td></td>
</tr>
<tr>
<td>type constructor, 161, 311</td>
<td></td>
</tr>
<tr>
<td>updatable record, 80</td>
<td></td>
</tr>
<tr>
<td>width, 160</td>
<td></td>
</tr>
<tr>
<td>super, 24, 263</td>
<td></td>
</tr>
<tr>
<td>superclass, 23, 24, 26, 28, 34, 35, 32, 32, 64, 89, 100, 191, 228, 233, 263, 304</td>
<td></td>
</tr>
<tr>
<td>supertype, 24</td>
<td></td>
</tr>
<tr>
<td>templates, 5</td>
<td></td>
</tr>
<tr>
<td>Theta, 364</td>
<td></td>
</tr>
<tr>
<td>Thorup, Kresten, 115</td>
<td></td>
</tr>
<tr>
<td>TOIL, 364</td>
<td></td>
</tr>
<tr>
<td>TOOPLE, 284, 364</td>
<td></td>
</tr>
<tr>
<td>Torgersen, Mads, 115</td>
<td></td>
</tr>
<tr>
<td>Trellis/Owl, 114, 363</td>
<td></td>
</tr>
<tr>
<td>Turing, Alan, 167</td>
<td></td>
</tr>
<tr>
<td>Turner, David, 284</td>
<td></td>
</tr>
<tr>
<td>type</td>
<td></td>
</tr>
<tr>
<td>Command, 134</td>
<td></td>
</tr>
<tr>
<td>anchor, 111</td>
<td></td>
</tr>
<tr>
<td>anchored, 363</td>
<td></td>
</tr>
<tr>
<td>array, 80</td>
<td></td>
</tr>
<tr>
<td>bounded existential, 157, 158</td>
<td></td>
</tr>
<tr>
<td>bounded polymorphic, 157</td>
<td></td>
</tr>
<tr>
<td>class, 84, 178</td>
<td></td>
</tr>
<tr>
<td>existential, 152, 153, 155, 156, 168, 206, 360</td>
<td></td>
</tr>
<tr>
<td>function, 75, 123</td>
<td></td>
</tr>
<tr>
<td>hash, 349, 353, 357, 358, 360</td>
<td></td>
</tr>
<tr>
<td>object, 20, 83</td>
<td></td>
</tr>
<tr>
<td>parameterized, 49, 141</td>
<td></td>
</tr>
<tr>
<td>polymorphic, 145, 292</td>
<td></td>
</tr>
<tr>
<td>record, 72, 133</td>
<td></td>
</tr>
<tr>
<td>recursive, 105, 150, 151, 161</td>
<td></td>
</tr>
</tbody>
</table>
Index

reference, 134, 174
run-time, 99
static, 99–101
sum, 133, 160
tuple, 132
universal, 145
updatable record, 80
variable, 77
virtual, 104, 107
type checking
   dynamic, 105
   static, 105
type constructors, 142
type error, 258
type system, 6
   invariant, 35, 36, 41, 45–49, 64, 108,
      194, 299, 358
   polymorphic, 50
   sound, 8, 10, 15, 16, 110, 111, 132,
      173, 287, 316, 322
typed lambda calculus, 120

Ungar, David, 115

variable
   instance, 13, 17–20, 35, 48, 84–86,
      91–93, 114, 151, 168, 180, 181,
      201–204, 212, 213, 218, 228, 229,
      272, 284
   virtual, 24
Viswanathan, Ramesh, 285
void, 18

Wegner, Peter, 115, 116, 168, 363