A Typed Intermediate Language and Algorithms for Compiling Scala by Successive Rewritings

Philippe Altherr

LABORATOIRE DES MÉTHODES DE PROGRAMMATION
INSTITUT D'INFORMATIQUE FONDAMENTALE
FACULTÉ INFORMATIQUE ET COMMUNICATION
ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Remerciements

J'aimerais tout d'abord remercier mon directeur de thèse Prof. Martin Odersky de m'avoir offert la possibilité de réaliser cette thèse. Travailler au sein de son laboratoire fut particulièrement intéressant et enthousiasment. Je le remercie de m'avoir accorder sa confiance même dans les moments les plus difficiles. Je le remercie en particulier de n'avoir jamais hésiter à faire le nécessaire afin de prolonger les différents délais que je m'entêtais à ne jamais tenir.

Je remercie Vincent Cremet avec qui j'ai beaucoup travaillé, particulièrement durant les dernières années. Beaucoup d'idées présentées dans cette thèse ont en fait été développées en étroite collaboration avec lui. J'aime à penser que dans une certaine mesure nous nous sommes mutuellement supervisés durant nos thèses.

Je remercie aussi Michel Schinz avec qui j'ai partagé mon bureau durant de nombreuses années et avec qui j'ai eu d'innombrables et intéressantes discussions sur des sujets aussi divers que variés.

Je suis très reconnaissant à tous mes collègues qui ont contribué à l'ambiance presque ludique qui régnait au sein du laboratoire et avec qui j'ai partagé de nombreuses heures à la fois durant et en dehors du travail. Je pense tout particulièrement à Sébastien Briais, Daniel Bünzli, Gilles Dubochet, Rachele Fuzzati, Stéphane Micheloud et Christine Röckl.

Je suis également reconnaissant aux autres membres du laboratoire qui contribuaient eux aussi à la bonne ambiance et plus particulièrement à Matthias Zenger pour les discussions très animées que nous avons eues.

J'aimerais encore exprimé ma gratitude aux membres de mon jury de thèse, Prof. Erik Ernst, Prof. Claude Petitpierre, Prof. André Schiper et Prof. Jan Vitek pour avoir pris le temps de lire et évaluer ma thèse.

Finalement, je remercie mes parents de m'avoir supporté durant toutes ces années et mon frère René et mes amis Julien et Alex de m'avoir souvent rappelé qu'il existait aussi autre chose en dehors du travail.

Résumé

Scala est un langage de programmation à usage général développé à l'EPFL. Il combine des concepts provenant des langages orientés-objets à d'autres provenant des langages fonctionnels. Scala est fortement typé est possède un système de types relativement complexe qui incorpore de nombreux concepts avancés.

Le compilateur Scala est consistuté d'une succession de phases qui réécrivent le code source en des versions de plus en plus simples jusqu'au stade où le code peut être trivialement traduit en code objet. Il est attendu que chaque phase produise du code Scala correctement typé.

Cette thèse commence par décrire en détails les constructions les plus importantes de Scala ainsi que son système de types. Elle s'intéresse ensuite à deux phases de réécriture du compilateur dont l'implantation s'est révélée beaucoup plus difficile que prévue. En effet, durant le développement du compilateur, il a été découvert que certains programmes ne peuvent pas être simplifié en les réécrivant si on se restreint à du code Scala correctement typé.

Les deux phases posant problème sont décrites en détails ainsi que les programmes qui ne peuvent pas être correctement réécrits. Un langage intermédiaire typé qui généralise certains aspects de Scala et permet ainsi de réécrire tous les programmes est décrit. Les deux phases de réécriture sont ensuite décrites formellement à l'aide de ce langage intermédiaire.

Mots-clés : langage de programmation, compilation, système de types, langage intermédiaire, réécriture de programmes

Abstract

Scala is a general-purpose programming language developed at EPFL. It combines concepts coming from object-oriented languages with other ones coming from functional languages. Scala is strongly typed and comes with a relatively complex type system, which incorporates several advanced concepts.

The Scala compiler consists of successive phases, which rewrite the source code into ever simpler versions until the code is simple enough such that in can be trivially translated into object code. It is expected that each phase generates well-typed Scala code.

This thesis starts by describing in details the main language constructions of Scala along with its type system. It then focuses on two rewriting phases whose implementation was much more difficult than expected. Indeed, during the development of the compiler, it was discovered that some programs cannot be simplified by rewriting them if the produced code has to be well-typed Scala code.

The two problematic phases are described in details as well as the programs that cannot be correctly rewritten. A typed intermediate language that generalizes some aspects of Scala and thus enables the rewriting of all programs is described. The two rewriting phases are then formally described using this intermediate language.

Keywords: programming language, compilation, type system, intermediate language, program rewriting

Contents

1 Introduction **1**
- 1.1 Scope . . . 1
- 1.2 The Scala Programming Language . . . 2
- 1.3 The Java Virtual Machine . . . 3
- 1.4 The Scala Compiler . . . 3
- 1.5 Typed Rewritings . . . 5
- 1.6 Scala Versions . . . 6
- 1.7 Outline . . . 7

2 The Scala Language **9**
- 2.1 Syntactic Sugar . . . 9
- 2.2 A Java-like Language . . . 10
- 2.3 An Object-Oriented Language . . . 11
  - 2.3.1 Packages, Classes, Traits and Objects . . . 11
  - 2.3.2 Class Hierarchy . . . 12
  - 2.3.3 Class Members . . . 13
  - 2.3.4 Constructors . . . 15
  - 2.3.5 Instance Creations . . . 17
  - 2.3.6 Primitive Values . . . 17
- 2.4 Type System . . . 18
- 2.5 Type Parameters . . . 19
  - 2.5.1 Typing Rules . . . 21
- 2.6 Parameter Variance . . . 24
- 2.7 Singleton Types and Stable Paths . . . 25
  - 2.7.1 Typing Rules . . . 26
- 2.8 Virtual Types . . . 28
  - 2.8.1 Typing Rules . . . 29
- 2.9 Refined Type . . . 30
  - 2.9.1 Typing Rules . . . 31
- 2.10 Class Type Parameters vs. Virtual Types . . . 33
  - 2.10.1 Safety . . . 34

2.10.2 Wildcards . . . 35
2.10.3 Constructors . . . 36
2.11 Inner Classes . . . 37
2.11.1 Enclosing Instances . . . 37
2.11.2 Instance Creations . . . 38
2.11.3 Scala's Inner Classes . . . 40
2.11.4 Typing Rules . . . 42
2.11.5 Qualified Class Types as Refined Types . . . 44
2.12 Mixins . . . 45
2.13 Explicit Self Types . . . 46

3 Lambda Lift **49**
3.1 Introduction . . . 49
3.2 Classical Algorithm . . . 52
3.2.1 Computing Extra Parameters . . . 52
3.2.2 Adding Extra Parameters and Arguments . . . 54
3.2.3 Substituting References to Free Variables . . . 55
3.2.4 Lifting Functions . . . 55
3.3 Generalization to Classes . . . 56
3.3.1 Lifting Local Classes . . . 56
3.3.2 Constructors . . . 57
3.3.3 Inner Classes . . . 58
3.3.4 Local Definitions . . . 58
3.3.5 Generalized Algorithm . . . 60
3.4 Mutable Variables . . . 60
3.5 Typing Issues . . . 62
3.6 Alternative Method . . . 64

4 Explicit Outer **67**
4.1 Introduction . . . 67
4.2 Base Algorithm . . . 69
4.3 Lost Privileges . . . 71
4.4 Lost Qualifiers . . . 72
4.5 Typing Issues . . . 74
4.6 Explicit Self Types . . . 76

5 The Core Language **79**
5.1 Introduction . . . 79
5.2 Syntax . . . 82
5.2.1 Symbols . . . 82
5.2.2 Definitions . . . 85

5.2.3 Expressions . . . 86
5.2.4 Example . . . 88
5.2.5 Syntactic Sugar . . . 89
5.2.6 Stable Expressions . . . 91
5.2.7 Outer Fields vs. Indexed Current Instances . . . 92
5.2.8 Flat Syntax . . . 95
5.3 Encodings . . . 97
5.3.1 Classes and Class Members . . . 97
5.3.2 Singleton Objects . . . 100
5.3.3 Packages . . . 100
5.3.4 Java Classes . . . 101
5.3.5 Types . . . 102
5.3.6 Constructors and Instance Creations . . . 106
5.4 Type System . . . 109
5.4.1 Auxiliary Functions . . . 109
5.4.2 Typing Rules . . . 114
5.4.3 Well-Formedness Rules . . . 117
5.5 Advanced Encodings . . . 120
5.5.1 Covariant Fields and Methods . . . 121
5.5.2 Explicit Self Types . . . 125
5.5.3 Custom Outer Fields . . . 126
5.6 Lambda Lift . . . 127
5.6.1 Introduction . . . 127
5.6.2 Computation of the Extra Sets . . . 130
5.6.3 Program Rewriting . . . 133
5.7 Explicit Outer . . . 136
5.7.1 Introduction . . . 136
5.7.2 Implementation . . . 141

6 Scaletta **143**
6.1 Introduction . . . 143
6.2 Typing Virtual Types with Inner Classes . . . 144
6.3 Untyped Calculus . . . 145
6.3.1 Syntax . . . 146
6.3.2 Semantics . . . 147
6.3.3 Examples . . . 149
6.3.4 Syntactic Sugar . . . 150
6.3.5 Methods . . . 150
6.3.6 Blocks . . . 152
6.3.7 Functions . . . 152
6.3.8 Safety and Confluence . . . 153

6.3.9 Alias Analysis . . . 153
6.4 Type System . . . 154
6.4.1 Types . . . 154
6.4.2 Annotations . . . 155
6.4.3 Abstract Evaluation . . . 155
6.4.4 Well-formedness . . . 157
6.4.5 Alias Analysis . . . 158
6.5 Typing Abstract Inheritance . . . 159
6.5.1 Field Roles . . . 160
6.5.2 Overriding of Type Fields . . . 160
6.5.3 Holes . . . 162
6.5.4 Formalizing Holes . . . 164
6.5.5 Overriding of Outer Fields . . . 164
6.5.6 Formalizing Class Exclusion . . . 166
6.5.7 Group Exclusion . . . 168
6.6 Encodings . . . 169
6.6.1 Methods . . . 169
6.6.2 Class Constructors . . . 171
6.6.3 Interfaces and Mixins . . . 172
6.6.4 Type Abstractions . . . 174
6.7 Undecidability and Typing Strategies . . . 175
6.8 Scaletta vs. Core Language . . . 176
6.9 Related Works . . . 177

7 Conclusion and Future Work 179
7.1 Scala . . . 179
7.2 Core Language . . . 179
7.3 Scaletta . . . 180

Bibliography 181

Index 185

Curriculum Vitæ 189

List of Figures

5.1 Core Syntax (nested version) 83
5.2 Core Syntax (flat version) 94
5.3 Core Constructor Functions 108
5.4 Core Auxiliary Functions 110
5.5 Core Auxiliary Functions Implementation (1) 111
5.6 Core Auxiliary Functions Implementation (2) 112
5.7 Core Auxiliary Functions Implementation (3) 113
5.8 Core Typing Relations 114
5.9 Core Typing Rules (1) 115
5.10 Core Typing Rules (2) 116
5.11 Core Typing Rules (3) 117
5.12 Core Well-Formedness Relations 118
5.13 Core Well-Formedness Rules (1) 119
5.14 Core Well-Formedness Rules (2) 120
5.15 Path Equality of Explicit Self Fields 126
5.16 Lambda Lift Set Computation Functions 130
5.17 Lambda Lift Set Computation Functions Implementation (1) 131
5.18 Lambda Lift Set Computation Functions Implementation (2) 132
5.19 Lambda Lift Rewriting Functions 133
5.20 Lambda Lift Rewriting Functions Implementation (1) 134
5.21 Lambda Lift Rewriting Functions Implementation (2) 135
5.22 Lambda Lift Rewriting Functions Implementation (3) 136
5.23 Explicit Outer Rewriting Functions 140
5.24 Explicit Outer Rewriting Functions Implementation (1) . . . 141
5.25 Explicit Outer Rewriting Functions Implementation (2) . . . 142

6.1 Scaletta Syntax 146
6.2 Scaletta Semantics 147
6.3 Scaletta Annotations 155
6.4 Scaletta Abstract Evaluation 156
6.5 Scaletta Well-Formedness Relations 157

6.6 Scaletta Hole Resolution 165
6.7 Scaletta Class Exclusion 167

Chapter 1

Introduction

1.1 Scope

Scala is a new strongly typed general-purpose programming language with object-oriented and functional aspects. It is a research language developed by Martin Odersky and his team at EPFL. It incorporates several advanced concepts from recent research.

However, Scala is not only a research language, it aims also at being of practical use for large-scale applications. An important library and a compiler supporting separate compilation and targeting the Java Virtual Machine were implemented. They are officially distributed since January 2004 and used in several courses at EPFL.

The Scala compiler was designed as a succession of code rewritings phases. The source code is first parsed into an intermediate language. This intermediate language is more or less a subset of the Scala language or, if one prefers, a desugared version of Scala. It is shared by all rewriting phases. The code is then successively rewritten by each phase, which simplify it by replacing complex constructs by simpler ones. After the last phase, the code is simple enough such that it can finally be trivially translated into code for the Java Virtual Machine.

The intermediate language is a typed one. To our surprise, we discovered during the implementation of the Scala compiler that this posed several unexpected problems for some of our rewriting phases. Indeed, with our typed intermediate language there were programs that could not be rewritten in a type safe way by some of our phases.

This thesis focuses on the problems raised by typed rewritings. First, the problematic phases and the programs that cannot be rewritten are described. The Core language, a typed intermediate language that resolves

the issues is then introduced. Finally, the problematic phases are formally described using the Core language.

1.2 The Scala Programming Language

Scala is a purely object-oriented language; like in Smalltalk, every value is an object (an instance of some class) and every operation is a method call (or a message send in Smalltalk terms). Classes can be defined pretty much everywhere; at the top-level but also within other classes (inner classes), within code blocks (local classes), even within expressions (anonymous classes). There is special syntax for classes with exactly one instance (singleton classes). Classes can inherit from a single class. They can additionally mixin one or more classes. This is similar but more powerful than the implementation of interfaces in Java.

Scala is also a functional language; every function is also a value. However, it is not purely functional as objects can have mutable states and functions can perform side-effects. Like classes, functions can be defined almost everywhere; within classes (methods), within code blocks (local functions), within expressions (anonymous functions) but unlike classes, they cannot be defined at the top-level. Scala supports several other features commonly found in functional languages like function closures, partial function applications, algebraic data-types and pattern matching.

The functional aspects are smoothly integrated with the object-oriented aspects. For example, closures are instances of special classes and algebraic data-types are a special kind of classes.

Scala is a strongly typed programming language. Its type system supports several typing features. These include common features like class types, function and class polymorphism but also less common features like variant class type parameters, virtual types, qualified class types and compound types, which let one specify that a value has to be an instance of a list of classes. There are even more exotic features like singleton types, which denote types including exactly one value, refined types, which are types that express constraints on class members, and explicit self types, which are annotations that specify the type of the current instance of a class.

From the beginning, Scala was designed with the aim of making it a possible successor to Java. It was therefore designed to run on the Java virtual machine and to inter-operate smoothly with existing Java code. For example, every Java class can be used as a normal Scala class. Thus, all Java libraries are directly accessible by Scala programs. Although Scala is

not an extension of Java, it remains very Java-like. Most Java concepts are also present in Scala and often have exactly the same syntax, semantics and/or typing rules.

Chapter 2 gives a much more in depth overview of Scala and its type system. It is however mainly targeted for compiler writers. An overview targeted for advanced programmers is available in [20]. There is also more introductory material available in [28, 21, 19]. Examples relying on advanced features of Scala are described in [26, 27]. A complete description of Scala is detailed in the official language specification [22]. The core aspects of Scala have been formalized in the *νObj* calculus [23].

1.3 The Java Virtual Machine

The Java virtual machine [17] is an abstract computing machine. It consists of a stack, a garbage collected heap and a stack-based processor. Programs consists of a set of `class` files and the name of the class containing the entry point.

Each *`class` file* fully describes a single class. At startup, the Java virtual machine loads the `class` file of the class containing the entry point, initializes the class and invokes the entry point method. If other `class` files are needed, they are loaded on the fly when the first instruction referencing them is executed.

The Java virtual machine guarantees that programs will never corrupt the memory. It achieves this by severely restricting the permitted operations. For example, neither the heap nor the stack can be directly accessed; there is nothing like pointer arithmetic. The Java virtual machine also relies on typed code; every `class` file is first type checked before any of its method is executed.

The Java virtual machine is very Java centric. A `class` files is more or less equivalent to a binary version of a Java 1.0 class with linearized code (labels and jumps instead of block structures).

1.4 The Scala Compiler

The Scala compiler consists of a front-end and a back-end. Both parts share the same typed intermediate language. The front-end reads the source files, checks whether they are syntactically and semantically correct and if so generates a term of the intermediate language representing the content

of the source files. This term is then passed to the back-end, which successively rewrites it in ever simpler forms and finally compiles it into a set of `class` files.

The intermediate language is a striped down version of Scala consisting only of the most fundamental constructs of Scala. It is much simpler than Scala but remains expressive enough to encode any Scala program. Although much simpler than Scala, turning it into `class` files is still not trivial and requires further processing.

The intermediate language is much simpler than Scala mainly for two reasons. First of all, several language constructs can be desugared into more fundamental ones. The second reason has to do with the fact that within the compiler, all definitions (classes, functions, variables, etc) have an identity and are represented by a unique symbol. References to definitions like for example in class types, method calls or variable assignments are represented by references to their unique symbol. This means that references are never ambiguous like in Scala where a name analysis is usually necessary to determine which definition is meant. This reduces considerably the number of required typing rules.

The *front-end* consists of a parser and an analyzer. The *parser* reads the source files, signals syntactic errors and produces a syntactic tree representing the content of the source files. The *analyzer* then verifies that the tree is semantically correct and if so translates it into a term of the intermediate language. In reality the verification and the translation are done in parallel and many semantic errors correspond to situations where the translation is impossible. The translation process mainly involves name analysis, type analysis and type inference. Each one of these three tasks is complex in itself. In addition to that each one relies on the two other ones. Therefore all three tasks have to be performed in parallel. This makes the analyzer one of the most complex part of the compiler.

The *back-end* consists of several rewriting phases and a code generator. Each *rewriting phase* takes a term of the intermediate language and rewrites it into an equivalent but simpler one. By simpler we mean either that the new term uses a smaller subset of the intermediate language or that it is in a form that is easier to translate into `class` files. The term resulting from the application of all phases is simple enough to be trivially translated into a set of `class` files by the code generator. As all phases simply rewrite terms of the intermediate language, which is a subset of Scala, the whole compiler can be viewed as a succession of source to source transformations, which transform the initial Scala program into an equivalent one that can be trivially compiled into `class` files.

There are five main rewriting phases: trans match, lambda lift, explicit

outer, expand mixin and type erasure.

The *trans match* phase translates pattern matching expressions into regular conditional expressions. The techniques used in the Scala compiler to perform this translation are described in [9, 8].

The *lambda lift* phase eliminates local classes and local functions by lifting them into their first enclosing class. Thus, it transforms all local classes into inner classes and all local functions into methods. This phase is described in Chapter 3.

The *explicit outer* phase eliminates inner classes by lifting them to the top-level. The lifted classes are augmented with an additional field containing the current instance of their enclosing class. This field is called the outer field, hence the name of the phase. This phase is described in Chapter 4.

The *expand mixin* phase eliminates mixins. This phase has been described by Schinz in [29].

The *type erasure* phase eliminates all the types that are not supported by the type system of the Java virtual machine. This implies more or less the elimination of all types that are not class types. Unsupported types are not really eliminated but rather replaced by (erased to) class types that approximate the original types. A similar transformation is required to compile GJ to the Java virtual machine. This transformation has been formalized by Igarashi, Pierce and Wadler in [12].

1.5 Typed Rewritings

Using a typed intermediate language has advantages but also some disadvantages. An obvious disadvantage is that it forces all code rewritings to handle not only value expressions but also type expressions. Thus, it complicates the specification and the implementation of these rewritings.

A typed intermediate language is helpful in the implementation of the code generator, which translates the intermediate language into `class` files. As the `class` files consists themselves of typed code, it would be necessary to compute all the required types during the generation of the `class` files if the types were not already present in the intermediate language. This would obviously complicate the implementation of the code generator.

The compiler currently performs almost no optimization but it is envisaged to add new phases to perform code optimizations. Such phases can greatly benefit from a typed intermediate language. Indeed, many optimizations can be performed only in very specific conditions that often

rely on the type of the involved terms. For example, a method call can be inlined only if it is possible to determine at compile-time which implementation will be called at runtime. In several contexts, by determining the type of the receiver object of a call to an abstract method, it is possible to determine that it is an instance of a class that has a final implementation of that method. Thus, although the called method is an abstract one, it can be established which implementation will be called at runtime.

The presence of types complicates the implementation of code rewritings but at the same time it also eases their debugging. Indeed, if a rewriting phase contains a bug, chances are great that it will generate ill-typed code. It is therefore possible to localize many bugs simply by type checking the code resulting from each rewriting phase.

Type checking rewritten code was indeed a very effective way to detect bugs. However, many bugs concerned error in the rewriting of types. Thus, in some sense, the types were a great bug detection mechanism to detect bugs in the detection mechanism.

In retrospect, it is unclear whether the presence of types shortened the development of the compiler by easing the detection of bugs or lengthened it by introducing many additional bugs. It is however clear that overall their presence was very positive. Indeed, it made us aware of many interactions between different aspects of the type system that we had not even imagined. It forced us to formalize some poorly understood aspects of Scala and this revealed several under-specifications and even some safety issues that required some changes in the Scala specification.

One thing we discovered during the development of the compiler is that some of its rewriting phases, namely lambda lift and explicit outer, cannot be applied to certain combination of features although they can be applied to each feature considered separately. For example, any Scala program involving inner classes but no virtual types can be rewritten to an equivalent Scala program without inner classes but there are programs involving both inner classes and virtual types that cannot be rewritten to equivalent Scala programs without inner classes. This is only possible if some aspects of Scala are generalized. That is why the Core language described in Chapter 5 generalizes some aspects of Scala.

1.6 Scala Versions

Scala is a young and dynamic language. Its compiler is actively maintained and new versions are regularly distributed. The Scala language itself is also still evolving but more slowly and only through gradual and

smooth steps. Nonetheless, since the work for this document began, a new version of the language called Scala 2 was developed and distributed.

Everything described in this document refers to the version of Scala now known as Scala 1 and the unqualified name Scala always designates Scala 1. However, this does not imply that this document does not pertain to Scala 2. Indeed, the two versions of Scala differ only in small details. Therefore most things described in this document also apply to Scala 2.

One important change that occurred in the compiler of Scala 2 is the reordering of the various rewriting phases. In the new compiler, the type erasure phase comes early and precedes most other phases, including the lambda lift and the explicit outer phases. This has this advantage that all these phases do not have to handle the most complex types of the Scala language and need only to handle very simple types. Thus, the new compiler avoids many problems described in this document. There are however also some disadvantages. For example, some optimizations are possible only if very specific conditions are fulfilled. Determining whether these conditions are fulfilled is sometimes tricky and in many cases relies on the presence of precise typing informations. Thus, erasing types early may prevent a following optimization phase from performing some optimizations by depriving it of sufficiently precise typing informations to let it determine that the required conditions for these optimizations are fulfilled.

1.7 Outline

Chapter 2 gives an overview of the Scala language. It focuses on the main constructs of the language, gives an in depth description of its type system and describes the relationships between the different language constructs and how they interact. Chapter 3 and Chapter 4 give a description mainly based on examples of the lambda lift and the explicit outer phases. They also describe and explain the code examples whose rewriting produce ill-typed code. Chapter 5 describes the syntax and the type system of the Core language and gives a formal description of the lambda lift and the explicit outer phases based on the Core language. Chapter 6 describes Scaletta, a calculus that inspired in many ways the design of the Core language. Finally, Chapter 7 concludes.

Chapter 2

The Scala Language

This chapter gives an overview of the Scala language. It assumes a good knowledge of Java but no preliminary knowledge of Scala. It is not intended as a Scala tutorial but it should be descriptive enough to let the reader understand Scala code.

This chapter aims at giving an overview of what has to be compiled by describing the main language constructs and the type system of Scala. The typing rules associated with each construct are described and related typing issues are discussed. The relationships and the interactions between the different constructs are described and it is explained how some constructs can encode others.

2.1 Syntactic Sugar

In this chapter we show that several Scala constructs are merely nothing more than syntactic sugar. The term syntactic sugar usually describes a language construct that can be translated into another one at parse time. We use here a slightly broader definition that includes also constructs that can be translated into other ones but only during the program analysis.

Let us illustrate that with an example. We consider a Java compiler composed of a parser, an analyzer and a back-end. The parser translates the source code into a syntactic tree. The analyzer verifies the tree and translates it into some intermediate code. Finally, the back-end compiles the intermediate code into `class` files.

A typical Java construct that is pure syntactic sugar is the **`if`** statement with no **`else`** part. The missing **`else`** part is equivalent to an **`else`** part with an empty block. This translation can be done by the parser. Therefore, neither the syntactic tree nor the intermediate code need to represent **`if`**

statements with no **else** part.

To call an instance method, an instance has to be provided. However, it is possible to omit the instance if it is the current instance. For example the method call `f()` is legal in the following code.

```
class C { void f() { f(); } }
```

The call `f()` means of course **this.**`f()`. This looks like syntactic sugar but strictly speaking it is not because the translation cannot be performed by the parser. Indeed, if the method `f` was static the call would mean `C.f()`. It is only during the program analysis when the identifier `f` is resolved that this distinction can be made. The initial call can then be safely translated into **this.**`f()`. Therefore, the intermediate code never needs to represent calls to instance methods with no instance.

The translation of missing instances, like the translation of missing **else** parts, reduces the complexity of the intermediate code by reducing the number of constructs it has to represent. That is why we call both constructs syntactic sugar.

2.2 A Java-like Language

When Scala was designed, one aim was to make it a possible successor to Java or at least a better alternative to Java. Contrary to many other languages that had a similar goal like for example Pizza [24, 25] or GJ [4], Scala was not designed as an extension of Java. Instead, it was designed as a new language that smoothly inter-operates with existing Java libraries and programs. Thus, regular Java source code is not valid Scala source code but once compiled it can be easily used by Scala code.

Although, Scala is not an extension of Java, it remains nonetheless very Java-like. It can certainly be classified in the same language family as Java. Scala, like Java, is a general-purpose, strongly typed, object-oriented language. They both run on the same virtual machine and thus share the same garbage-collected memory model. They have also very similar notions of packages, classes and class members and in fact every Java class is also recognized as a regular Scala class and can be used as such in any context. Most aspects of Scala are either identical to corresponding Java aspects or are extensions or generalizations of such aspects. There are only a few places where Scala behaves differently from Java.

Even the Scala syntax is very Java-like although this is probably not so obvious at first sight. However, most discrepancies are mainly due to three syntactic differences. First of all, contrary to Java where only package and

class definitions start with a keyword, in Scala all definitions start with a keyword; method definitions start with the keyword **`def`** and field and local variable definitions start with either the keyword **`val`** for immutable ones or the keyword **`var`** for mutable ones. The second difference is that in Scala types are written after the parameter name, the field name or the method name and parameter list whereas in Java types are written in front of them. For example, one writes `s: String` instead of `String s`. The third difference is that in Scala there are no statements, only expressions. For example, the Scala **`if`** behaves like the Java ternary operator `?:` and blocks return the value of their last expression. Both can occur within expressions. Apart from these differences, the Scala syntax is more or less an extension of the Java syntax. Thus, most valid Java statements and expressions are also valid in Scala.

At the time when Scala was designed for the first time, the latest version of Java was version 1.4. When Scala was designed to inter-operate smoothly with Java it was done with that version of Java in mind. Since then Java has evolved and Java 5 with its support for generic types has appeared. Those generic types are currently not yet supported by Scala. So when it is stated that Scala smoothly inter-operates with Java, one should understand that is smoothly inter-operates with Java 1.4. Throughout the rest of this document, the term Java without any other indication will always designate Java 1.4 and the term Java 5 will be used to designate the version of Java with generic types.

2.3 An Object-Oriented Language

From the programmer's perspective, Scala is a pure object-oriented language. Like in Smalltalk, every value is an object (an instance of some class) and every operation is a method call (or a message send in Smalltalk terms).

2.3.1 Packages, Classes, Traits and Objects

Classes are defined in packages, which play exactly the same role as in Java. Every Java package is also a Scala package and vice-versa. All classes excepted the one at the root of the class hierarchy extend exactly one other class. Each class may additionally mixin one or more other classes. This is similar but more powerful than the implementation of interfaces in Java. It is further described in Section 2.12. For now, we will consider that classes only inherit from their superclass.

In Scala, there are no interfaces but *traits*, which are stateless abstract classes. In Java terms, traits are interfaces with a superclass and which may have non-abstract methods. Traits enjoy some privileges when they are used as mixin classes but are otherwise equivalent to abstract classes.

Each Java class is seen as a regular Scala class and each Java interface as a regular Scala trait. Their inheritance graph remains the same; superinterfaces are simply seen as mixed-in traits and interfaces as traits extending the class `Object`.

A *singleton class* is a class of which there exists exactly one instance. An *object* definition defines such a class and names its unique instance. Here is an example:

```
object MyObject extends /* parents */ { /* body */ }
```

This definition is equivalent to the following ones excepted that variable definitions are not allowed at the top-level and that with the object definition the underlying class remains inaccessible, which effectively prevents further instantiations. This encoding is also somewhat inaccurate because it does not reflect the fact that objects are created lazy when they are accessed for the first time.

```
val MyObject: MyObject$ = new MyObject$();
class MyObject$ extends /* parents */ { /* body */ }
```

In Scala, the term *object* designates both any instance of some class and the unique instance of a singleton class. The context usually makes it clear which one is intended. If necessary, the term *singleton object* will be used to specifically designate the latter one.

2.3.2 Class Hierarchy

The class `Any` is at the root of the class hierarchy. It has only two predefined subclasses: `AnyVal` and `AnyRef`. It may not be extended by user-defined classes.

The class `AnyVal` is the base class of all *value classes*. Instances of value classes have no identity and are always passed by value. Value classes cannot be defined by the user; there are only nine predefined ones: `Boolean`, `Byte`, `Short`, `Char`, `Int`, `Long`, `Float`, `Double` and `Unit`. The first eight ones correspond to the eight primitive types of Java : **`boolean`**, **`byte`**, **`short`**, **`char`**, **`int`**, **`long`**, **`float`** and **`double`**. The class `Unit` corresponds more or less to the Java return type **`void`**. It is a normal class whose usage is not restricted to return types. It has a unique instance denoted `()` and called *unit*. In Scala, all functions have to return some value. Those that would

be of type **void** in Java are defined with the type `Unit` and return the value `()`. Java methods of type **void** are seen as if they were of type `Unit`.

The class `AnyRef` is the base class of all *reference classes*. Instances of reference classes have an identity and are always passed by reference. All user-defined classes extend directly or indirectly `AnyRef`. The Java class `Object` is seen as if it extended `AnyRef`.

The class `All` is at the bottom of the class hierarchy; it inherits from all other classes. It can neither be extended nor instantiated. The class `AllRef` is just above the class `All`; it inherits from all reference classes. It too cannot be extended but it has a unique instance: the value **null**. There is no class `AllVal`.

2.3.3 Class Members

Class members include methods, fields and inner classes but also virtual types. In Java, class members are either static or instance members. In Scala there are no static members; all members are instance members. Another difference is that in Scala the default visibility of a member is public. The private and protected visibility are available but not Java's package private visibility.

Static members of a Java class `C` are seen as instance members of a pseudo-object `C`. For example, like in Java, `System.out` is a legal expression but `out` is seen as an instance member of the pseudo-object `System`. The pseudo-object `System` can only be used as a prefix in a member selection. It is not a real value and cannot be passed around. Package private members of Java classes are seen as private members.

Methods and fields are described below. Inner classes and virtual types are described in Section 2.11 and Section 2.8.

2.3.3.1 Methods

Methods are like in Java and like in Java 5 they are *covariant* in their return type; their return type can be refined in subclasses. In the code below, the class `A` defines an abstract method `f` with the return type `Any`. The class `B` implements it and refines its return type to `A`. The class `C` overrides it with a new implementation and further refines it return type to `C`.

```
abstract class A { def f(a: A): Any; }
class B extends A { def f(a: A): A = a; }
class C extends B { override def f(a: A): C = this; }
```

Note that abstract methods are defined simply by omitting their implementation and method definitions that override an inherited implementation must be annotated with the keyword **override**.

Methods are not *contravariant* in their parameters; the types of their parameters cannot be widened although this would be perfectly sound. For example, the type of the parameter a could be widened to Any in the class C. This would pose no safety issue. However this conflicts with method overloading. In many cases it would be unclear whether one intends to overload a method or to override it contravariantly. To avoid such ambiguities, contravariant method overriding is not supported.

2.3.3.2 Fields

Fields, like methods, can be abstract and can be overridden in subclasses but their type cannot be refined. In the following example, the class A defines an abstract field v, the class B implements it and the class C overrides it.

```
abstract class A { var v: Int; }
class B extends A { var v: Int = 0; }
class C extends B { override var v: Int = 1; }
```

In an instance of a class that overrides a field, all readings and writings of that field affect the overriding field. The overridden field is however still present and, like an overridden method, it can be accessed through a super call. In our example, each instance of C inherits two fields v; one initialized to 0 and another one initialized to 1. From within the class C, the first one is accessible with the expression **super.**v.

Field definitions are syntactic sugar. Each concrete field defines a private field and a pair of getter and setter methods. Abstract fields define just a pair of abstract getter and setter methods. Field readings and writings are desugared into calls to the corresponding getter and setter methods. Our example is desugared into the following code.

```
abstract class A {
  def v: Int;
  def v_=(v: Int): Unit;
}
class B extends A {
  primitive private var v$: Int = 0;
  def v: Int = v$;
  def v_=(v: Int): Unit = v$ = v;
}
```

```
class C extends B {
  primitive private var v$: Int = 1;
  override def v: Int = v$;
  override def v_=(v: Int): Unit = v$ = v;
}
```

The keyword **primitive** is not a real Scala keyword. In our examples, it is used to indicate constructs that have already been desugared. In the last example, it indicates that the two fields v$ are plain Java-like fields. They will not be further desugared and they are independent fields; none overrides the other.

The desugared version shows that abstract fields and overriding fields are implemented through abstract methods and overriding methods. The real underlying fields are just like in Java.

Immutable fields (defined with **val** instead of **var**) are desugared like mutable ones but with no setter method. Immutable fields are covariant; their type can be refined in subclasses. This translates into covariant getter methods.

2.3.4 Constructors

A class definition defines a new class but also an *implicit constructor* of the new class. Each class has a list of parameters and its superclass a list of arguments[1]. The class parameters and the superclass arguments constitute respectively the parameters and the arguments of the super-constructor call of the implicit constructor. The class parameters can be referenced by the superclass arguments but also from anywhere within the class body. Additional code can be added to the implicit constructor by putting it directly into the body of the class.

The example below defines a class B with a one-parameter constructor and a subclass C with a two-parameter constructor. The C constructor first invokes the B constructor, then initializes the field v and finally prints the string.

```
class B(x: Int) { /* ... */ }
class C(y: Int, z: Int) extends B(y) {
  val v: Int = y + z;
  System.out.println("Initialized instance of C");
  def f(): Int = z;
}
```

[1] Empty lists can be entirely omitted.

Additional constructors can be defined with method definitions with no return type and whose name has been replaced with the keyword **this**. For example, a zero-parameter constructor could be added to the class C with the following definition.

```
def this() = this(0, 0);
```

The body of explicit constructors has to start with a call to another constructor of the current class. Contrary to Java, calls to constructors of the superclass are not allowed. Indeed, this makes no sense in the presence of class parameters. In our example, if the zero-parameter constructor directly invoked the constructor of the superclass B, the parameters y and z would remain uninitialized. The constructor could possibly initialize the field v but it would still be unclear what the method f should return. Initializing the class parameters is not an option because contrary to what was stated above, class parameters are in fact accessible from everywhere within the class excepted from within constructors. And, changing this would cause more problems than it would solve.

We call *primary constructor* a constructor that invokes a constructor of the superclass and *secondary constructor* one that invokes a constructor of the same class. In Scala, each class has exactly one primary constructor, namely the one defined implicitly by the class definition. All explicit constructors are secondary ones.

Although the Scala syntax does not allow explicit primary constructors, the compiler internally desugars class definitions into classes with such a constructor. For example, the class C is desugared into the following code.

```
primitive class C extends B {
  private val z$: Int = _;
  val v: Int = _;
  def this(y: Int, z: Int) {
    this.z$ = z;
    super(y);
    this.v = y + z;
    System.out.println("Initialized instance of C");
  }
  def f(): Int = this.z$;
}
```

The keyword **primitive** preceding the class definition indicates that no implicit constructor is defined. The value _ in the field initializers indicates that the fields are not abstract but remain uninitialized.

The desugared version shows that extends clauses specify two things: a superclass and a call to a constructor of that superclass. In our example, the extends clause `B(y)` specifies the superclass `B` and the call **`super`**`(y)`.

Desugaring a class definition is not completely trivial because some parameters may be referenced by code that does not end up in the primary constructor. For each of these parameters, it is necessary to create an additional field, which stores the value of the parameter, and all the problematic references to the parameter must be replaced with references to this field. In our example, the field `z$` had to be added because the parameter `z` is referenced from within the method `f`. The parameter `y` is only referenced from within the constructor and therefore requires no additional field.

2.3.5 Instance Creations

Instance creations use the same syntax as in Java. It is important to understand that conceptually instance creations, like extends clauses, specify two things. The first one is the class of which a new uninitialized instance has to be created. The second one is a constructor of that class and the arguments with which it has to be invoked to initialize the new instance. For example, the expression **`new`** `C(1, 2)` can be viewed as a shortcut for the following expression:

```
{ val tmp: C = new C; tmp.this(1, 2); tmp }
```

2.3.6 Primitive Values

There are neither primitive values nor primitive types in Scala. For example, the Scala class `Int`, which corresponds to the Java primitive type **`int`**, is a perfectly normal class and integer literals are seen as instances of that class.

There are also no primitive operations. For example the integer addition is seen as a method of the class `Int`. In Scala, identifiers are not restricted to alphanumeric character sequences. Furthermore methods with a single argument can be used as binary operators. Thus, `+` is an identifier and the expression `x + y` is just an alternate way of writing `x.+(y)`.

Although, from the point of view of the programmer, there are neither primitive types nor primitive values, the Scala compiler handles types and values that correspond to Java's primitive types and values in a special way and compiles them to the corresponding primitive types and values of the Java virtual machine.

2.4 Type System

The Scala type system is defined in a very standard way. It relies on types, a reflexive and transitive subtyping relation and a typing relation, which includes the subsumption rule. It distinguishes itself from other type systems by the diversity and the expressiveness of its types.

The *class types* constitute the simplest form of types. To every class `C` corresponds a type `C`, which is called a class type. It is the the most precise type of an instance of the class `C`. There are several other forms of types. These are described in the following sections.

The subtyping of class types is defined by the inheritance graph; if a class `C` extends a class `B`, then the type `C` is a subtype of the type `B`. The class `All` inherits from all classes and all classes inherit from the class `Any`. The type `All` is therefore a subtype of all class types and all class types are subtypes of the type `Any`. These two types constitute therefore the natural bottom and top types of the type system. We will see that this remains true even in the presence of the other forms of types.

In the sole presence of class types, the typing rules for most expressions are trivial. We enumerate here the most important ones. The current instance **`this`** of a class `C` has the type `C`. An instance creation of a class `C` has the type `C` provided all arguments conform to the declared type of the corresponding parameters of the invoked constructor. A value is said to *conform* to some type if it is of that type. A variable has the type declared by its definition. A field selection has the type declared by the definition of the field provided the field is selected on a value that inherits it. A value is said to *inherit* a field or more generally a member if it conforms to the class type of the class that defines the member. An assignment has the type `Unit` provided the expression conforms to the type of the assigned variable. A conditional expression has for type the least upper bound of its two alternatives provided its condition conforms to the type `Boolean`. As long as single inheritance is assumed, the *least upper bound* of two class types is the lowest common supertype of the two types. The presence of a top type guarantees that this type always exists. Finally, a method call has the type declared by the definition of the method provided the method is selected on a value that inherits it and the arguments conform to the declared type of the corresponding parameters.

The above typing rule for method calls is only an approximation of the actual rule. Indeed, expressed like above, it is not completely satisfactory because it does not take into account the fact that methods are covariant in their return type. Let us illustrate this with the two classes defined below where the method `f` is abstractly defined in the class `A` with the return

type A and implemented in the class B with its return type refined to B.

```
abstract class A  { def f(): A; }
class B extends A { def f(): B = this; def g(): B = this; }
```

With the above typing rule, the expression `new B().f()` has the type A because the method f is defined in the class A with the return type A. The expression `new B().f().g()` is therefore ill-typed because the class A does not inherit the method g. However, it would be well-typed if the typing rule for method calls would take into account the fact that g is selected on an instance of B and that the class B refines the return type of f to B. In fact, the actual typing rule for method calls does exactly that.

The type of a method call is determined by the method *lookup*, which takes as an argument the class C of the receiver's type. If the class C refines the type of the called method m, it returns the refined type. Otherwise, if the class C is the one that defines the method m, it returns the declared type of m. And otherwise, it recursively invokes itself with the superclass of C. If the function fails to return a type because it reaches the root of the class hierarchy, it indicates that the method m is illegally called with a receiver that does not inherit it.

2.5 Type Parameters

Like Java 5, Scala supports polymorphic classes and methods. These are defined with a list of type parameters where each parameter has a lower and an upper bound, which may both reference any parameter of the list. One or both bounds may be omitted. They respectively default to All and Any. The code below defines a class C and a function f, both with a single type parameter X whose lower and upper bounds are respectively S and U.

```
class C[X >: S <: U](x: X) { /* ... */ }
def f[X >: S <: U](x: X): Unit = /* ... */;
```

Method calls of polymorphic methods must provide a type argument for each type parameter of the method. For example, f[T](t) is a legal call of f if t is a value of type T and S is a subtype of T and T a subtype of U. When the compiler is able to infer the type parameters, they can be omitted. However, this does not change the fact that conceptually all method calls provide a type argument for each type parameter. For example, the call f(t) is simply desugared (through type inference) into f[T](t).

Similarly, class types of polymorphic classes must provide a type argument for each type parameter of the class. Such class types are called *parameterized class types*. Type arguments must also be provided when a

polymorphic class is used as a superclass and in instance creations of polymorphic classes. The following code illustrates this.

```
class D[Y >: S <: U](y: Y) extends C[Y](y);
val d: D[T] = new D[T](t);
```

The desugared version of this code is given below. Observe how the type parameters and arguments remain associated with the classes while the value parameters and arguments get associated with the constructors. This shows that extend clauses and instance creations specify a class type and a constructor call rather than just a class and a constructor call.

```
primitive class D[Y >: S <: U] extends C[Y] {
  def this(y: Y) = super(y);
}
val d: D[T] = { val tmp = new D[T]; tmp.this(t); tmp }
```

The example also demonstrates that the implicit constructor of polymorphic classes is not polymorphic. However, it seems perfectly reasonable to imagine cases were both the class and its implicit constructor are polymorphic. For example, one could write the following desugared code:

```
primitive class E[Z] {
  val z: Z = _; val i: Int = _;
  def this[V](z: Z, l: List[V]) = {this.z = z; this.i = l.length;}
}
val e: E[String] =
  { val tmp = new E[String]; tmp.this[Int]("", List[Int]()); tmp }
List[String]()
```

If we try to "sugarize" this, it leads to the code below where the class `E` is defined with two distinct lists of type parameters and also instantiated with two distinct lists of type arguments.

```
class E[Z][V](_z: Z, l: List[V]) {
  val z: Z = _z; val i: Int = l.length;
}
val e: E[String] = new E[String][Int]("", List[Int]());
```

Neither class definitions with two type parameter lists nor instance creations with two type argument lists are allowed in Scala. It is therefore impossible to define polymorphic constructors. However, that reflects only a limitation of the current Scala syntax. It is in no way a fundamental limitation of the computing model underlying Scala. The desugared version demonstrates it. Another proof is given by Java 5 where defining such constructors is possible as shown below.

```
class E<Z> {
  final Z z; final int i;
  <V> E(Z z, List<V> l) { this.z = z; this.i = l.size(); }
}
E<String> e = new E<String>("", new LinkedList<Integer>());
```

Conceptually, the instance creation has two type argument lists but because in Java 5 type arguments of method calls are always inferred and never provided in the source code, only one list appears in the code.

2.5.1 Typing Rules

The subtyping rules for type parameters are very simple. If X is a type parameter whose lower and upper bounds are T and U, then T is a subtype of X and X is a subtype of U. There is however a small subtlety illustrated by the two function definitions below. In principle, both should type check because in both definitions it can be shown by applying the two subtyping rules above that `X` is a subtype of `Y` and `Y` is a subtype of `Z`. However, only the first definition does type check.

```
def f[X <: Y, Y, Z >: Y](x: X): Z = x;
def g[X, Y >: X <: Z, Z](x: X): Z = x; // error
```

The problem with the second definition is that it requires an omniscient compiler. Indeed, neither `X` nor `Z` mention `Y` in their bounds. The compiler would therefore have to guess that the bounds of `Y` must be used to prove that `X` is a subtype of `Z`. This does not happen and the second definition does therefore not type check.

To determine whether a type is a subtype of another one, the compiler relies only on the lower and upper bounds of the two types. The *lower* (resp. *upper*) *bounds* of a type consist of that type followed by the lower (resp. upper) bounds of its lower (resp. upper) bound. By definition, class types have neither a lower nor an upper bound. The lower and upper bounds of a class type consist therefore both of this sole class type.

The definition of lower and upper bounds implies that both consist of a possibly empty list of type parameters followed by a class type. Unfortunately, this is not perfectly true because the lists can also be infinite lists of type parameters. For example, in the definition below, the upper bounds of `X` and `Y` are both infinite. To avoid such infinite lists, Scala requires that each type parameter is lower and upper bounded possibly indirectly by some class type. These class types are called the *lower and upper class bounds*. This restriction has the added benefit that it guarantees that `All` is

a subtype of any type parameter and that any type parameter is a subtype of `Any`. Thus, `All` and `Any` keep their status of bottom and top types.

```
def e[X <: Y, Y <: X](x: X): Y = x; // error
```

Determining whether a type T is a subtype of some type U can now be done by computing the upper bounds of T and the lower bounds of U. The type T is a subtype of U if and only if the two lists share a common type parameter or if the final class type of the upper bounds is a subtype of the final class type of the lower bounds. In our initial example, in the definition of `f`, the upper bounds of `X` are `X`, `Y` and `Any` and the lower bounds of `Z` are `Z`, `Y` and `All`. The type `Y` is shared by the two lists, `X` is therefore a subtype of `Z`. In the definition of `g`, the upper bounds of `X` are `X` and `Any` and the lower bounds of `Z` are `Z` and `All`. The two lists share no common type and `Any` is not a subtype of `All`. Therefore, `X` is not a subtype of `Z`.

When looking for a type shared by the two lists, it is important to compare each upper bound to each lower bound. Indeed, the shared types are not necessarily the last ones, not even the last type parameters of each list. It may happen that the two lists "cross" each other. For example, with the definition below, the upper bounds of `X` are `X`, `Y`, `Z2` and `Any` and the lower bounds of `Z` are `Z`, `Y`, `X2` and `All`. The shared type `Y` occurs in the middle of the type parameters of each list.

```
def h[X <: Y, X2, Y >: X2 <: Z2, Z2, Z >: Y](x: X): Z = x;
```

With the introduction of type parameters, testing whether a class type is a subtype of another one becomes a bit more difficult because type arguments have to be taken into account. Parameter variance, which is discussed in Section 2.6, adds even more complications. In the rest of this section we ignore this aspect and consider only classes with invariant type parameters.

A class type T is a subtype of a class type U of the same class if the type arguments of T are equal to those of U. Two types are *equal* if each one is a subtype of the other one. To compare class types of different classes, the function *base-type* is used. This function takes a type R and a class C as arguments and returns a class type S of the class C such that R is a subtype of S or fails if no such class type exists. This function is implemented as follows. If the type R is not a class type, the function is invoked recursively with the upper bound of R. Otherwise, R is a class type of some class D. If D is equal to C, R is returned. Otherwise, the function is invoked recursively with the declared supertype of the class D, in which all type parameters of D have been replaced with the type arguments of R. Finally, if D has no supertype, the function fails. A type T is a subtype of

a class type U of class C if and only if *base-type*(T, C) returns some type S and the type arguments of S are equal to those of U.

The introduction of type parameters also affects the typing rules of field selections and methods calls. The declared types of fields and methods may contain type parameters and these need to be replaced with actual types. For example, with the definitions below, the field selection `d.vs` is not of type `List[X]` but of type `List[List[Int]]`.

```
class C[X]() { var vs: List[X] = Nil; }
class D[Y]() extends C[List[Y]]();
val d: D[Int] = new D[Int]();
```

The function *as-seen-from* is used to type field selections and method calls. This function takes three arguments: a type T, the class C in which the type T occurs and a type U of an instance of the class C. It returns the type T in which all type parameters of the class C have been replaced with the type arguments implied by U. The implied type arguments are those of the class type returned by *base-type*(U, C). The type of a field selection $x.v$ is now defined as the type returned by *as-seen-from*(T, C, U) where T is the declared type of the field v, C the class in which v is defined and U the type of x. For example, the type of `d.vs` is the one returned by *as-seen-from*(`List[X]`, `C`, `D[Int]`). The call *base-type*(`D[Int]`, `C`) evaluates to `C[List[Int]]`. Thus, the parameter `X` of the class `C` must be replaced with `List[Int]` in the type `List[X]`. This yields `List[List[Int]]`, which is the type of `d.vs`.

The typing of method calls still relies on the function *lookup* but this function needs to be modified to return not only a type T but also the class C in which this type was found. This type and this class are then passed to the function *as-seen-from* along with the type U of the receiver. The resulting type may still contain type parameters of the method. These are eliminated by replacing them with the type arguments of the method call. This finally yields the type of the method call.

A last thing affected by the introduction of type parameters is the computation of the least upper bound of two types. This bound is now obtained by computing the upper bounds of the two types and taking the first common bound. If there is no such bound, the least upper bound of their upper class bounds is computed. The least upper bound of two class types is computed as before by determining their first common superclass. However, it is now necessary to check that both types have the same type arguments for that class. If not the next superclass has to be tried. The top type `Any` still guarantees that a least upper bound always exists.

2.6 Parameter Variance

Class type parameters have a variance. A type parameter is covariant if it is preceded by a `+`, contravariant if it is preceded by a `-` and invariant otherwise. Different parameters of the same class may have different variances. The code below declares a class `M` covariant in `X`, a class `N` contravariant in `Y` and a class `O` invariant in `Z`.

```
class M[+X]; class N[-Y]; class O[Z];
```

When two class types of the same class are compared to determine whether one is a subtype of the other, the variance of type parameters determines how corresponding type arguments are compared. Arguments for invariant type parameters have to be equal. For example, the type `O[T]` is a subtype of `O[U]` if and only if `T` and `U` are equal. For covariant type parameters, the argument in the subtype has to be a subtype of the argument in the super-type. So, `M[T]` is a subtype of `M[U]` if and only if `T` is a subtype of `U`. For contravariant type parameters, it is the opposite; the argument in the subtype has to be a super-type of the argument in the super-type. Thus, `N[T]` is a subtype of `N[U]` if and only if `U` is a subtype of `T`.

The usage of covariant and contravariant type parameters is unsafe if some restrictions are not enforced. The code below demonstrates this. In this example, the class `D` correctly implements the method `f`. The new instance of `D` is an instance of `C[Int]`, which is indeed an instance of `C[Any]` because `C` is covariant in `X`. Finally, the call to `f` is also well-typed as the string `"A"` is indeed an instance of `Any`. However, the call results in the evaluation of the expression `"A" - 1`, which is not well-formed. This shows that it is unsafe to use a covariant type parameter as the type of a method argument.

```
abstract class C[+X] { def f(x: X): Int; }
class D extends C[Int] { def f(x: Int): Int = x - 1; }
val c: C[Any] = new D();
c.f("A");
```

To ensure type safety, covariant and contravariant parameters are restricted respectively to covariant and contravariant positions. *Covariant positions* include type lower bounds, function return types and immutable value types. *Contravariant positions* include type upper bounds and function argument types. There are also *invariant positions*, which include mutable variable types. Invariant parameters can be used in invariant positions but also in covariant and contravariant ones.

The definition of covariant and contravariant positions is not yet complete. It says nothing about arguments to class types. Arguments for invariant parameters are always in invariant positions. The status of the other arguments depends on the position of the class type itself. If the class type is in a covariant position, then arguments for covariant parameters are in covariant position and arguments for contravariant parameters are in contravariant position. If the class type is in a contravariant position, then it is just the opposite: arguments for covariant parameters are in contravariant position and arguments for contravariant parameters are in covariant position. Finally if the class type is in an invariant position, all its arguments are also in an invariant position.

All that is fairly complex. Some of this complexity can be removed by noting that two types are equal if and only if each one is a subtype of the other one. An invariant parameter may therefore be interpreted as a parameter that is both covariant and contravariant. Covariant and contravariant positions can be interpreted as positions where a type parameter can be used only if it is at least respectively covariant or contravariant. Finally, invariant positions can be interpreted as positions that are both covariant and contravariant.

2.7 Singleton Types and Stable Paths

A type can be interpreted as a set of values. For example, with this interpretation, the type `String` represents the set of all possible instances of the class `String`. When a value v is of some type T, it simply means that v is an element of the set represented by T and if T is a subtype of U, it means that the set represented by T is a subset of the one represented by U. These sets are usually infinite but in Scala it is possible to define types that contain exactly one value. These types are called *singleton types*.

Given an expression e that evaluates to some value v, the type e`.type` stands for the type containing only the value v. This implies that if some expression has the type e`.type` its evaluation either does not terminate or returns v. An expression whose type is a singleton type is called a *stable value*.

A singleton type e`.type` is sound only if successive evaluations of e always return the same value v. This is not necessarily true because expressions may have side effects. Therefore expressions in singleton types are restricted to *stable paths*, which are expressions for which this property is always true.

A stable path is by definition either the current instance, an immutable

variable or the selection of an immutable field on a stable path. Note that in Scala, class and function parameters are always immutable and form therefore stable paths.

2.7.1 Typing Rules

The introduction of singleton types does not fundamentally change the typing rules, only some local changes are required. A first change is that expressions p that are also stable paths have now the type p`.type` in addition to the type that they would have as a normal expression, which is called their *plain type*.

Like type parameters, singleton types need to be taken care of while typing field selections and method calls. Let us illustrate this with the definitions below and the typing of the expression `v.y`. The field `y` has the declared type `x.type`, which is of course a shortcut for the type `this.x.type`. The type of `v.y` is obtained by replacing in the type of `y` the current instance `this` with the actual instance `v`, which gives `v.x.type`.

```
class C[X](_x: X) { val x: X = _x; val y: x.type = x; }
def f(): C[String] = new C("");
val v: C[String] = new C("");
```

To perform the kind of substitution described above while typing field selections and method calls, we do not need to modify their typing rules. Indeed, these rules already rely on the function *as-seen-from* to replace type parameters with the adequate type arguments in field and method types. We can therefore simply adapt this function to also handle singleton types. We remind that the function *as-seen-from* takes as arguments a type T, the class C in which the type T occurs and a type U of an instance of the class C and returns the type T in which all type parameters of the class C have been replaced with the type arguments implied by U. From now on, it will also replace in T all occurrences of `this` with the instance implied by the type U. If the type U is a singleton type p`.type`, the implied instance is p. Otherwise, the exact identity of the implied instance is unknown. In that case, there is no stable path to replace the occurrences of `this`. Therefore, the function *as-seen-from* fails if there is at least one such occurrence.

The fact that the function *as-seen-from* can fail implies that fields and methods whose type contains a reference to `this` can only be selected on stable values. So, `v.y` is legal because `v` is of type `v.type`, but `f().y` is illegal because `f()` is of type `C[String]`, which is not a singleton type. Note however, that if such a field or method needs to be selected on a non-stable value, it is always possible to store that value in a immutable local

variable and then select the field or method on that variable.

The following code illustrates two restrictions that apply to singleton types. First, the value parameters of a function can neither be used in its return type nor in the types of its parameters. Secondly, the value parameters of a class cannot be used in any type occurring in the definition of any of its non-private members.

```
def g[X](x: X): x.type = x; // error
class D[X](x: X) { val y: x.type = x; } // error
```

It would be possible to lift the restriction on function parameters. That would imply that the type of a function call can depend on some of its arguments. These arguments, like the receiver of methods whose return type reference **this**, would have to be stable values. For example, if `p` is a stable path, `g(p)` would be of type `p.type`.

The restriction on class parameters is more fundamental. Lifting it would require more than just passing stable values as arguments of instance creations. Indeed, given an instance `d` of `D`, the type of `d.y` can be expressed only if the type of `d` includes the identity of the parameter `x`. Thus, the type of an instance of `D` would have to look something like `D[T](p)`.

All singleton types are lower bounded by `All` and upper bounded by their plain type. This implies that the lower bounds of a type may now also contain one singleton type and its upper bounds any number of singleton types. To avoid infinite upper bounds, Scala requires that the type of any variable is upper bounded by some class type. Without that restriction it would be possible to define variables like `val x: x.type` whose type has an infinite number of upper bounds.

When testing whether a type T is a subtype of a type U, it is now necessary to also check whether the upper bounds of T and the lower bounds of U contain equal singleton types. Similarly when computing the least upper bound of two types, it is now necessary to also look for equal singleton types.

Two stable paths are equal if their respective singleton types are equal. Two singleton types are equal if they denote the same value. This is true if one is an upper bound of the other. It is also true if their respective stable paths both select the same field on equal stable paths. To determine whether two singleton types T and U are equal, we compute the upper bounds of each type and take the stable path of the last singleton type of each list. The types T and U are equal if and only if the two paths are the same or if they select the same field on equal paths.

Let us prove that given the definitions below, `b.w1.v1` and `b.w2.v2` are

equal. The upper bounds of `b.w1.v1.type` are itself and `String`. The upper bounds of `b.w2.v2.type` are itself, `b.w2.v1.type` and `String`. The stable paths of the last singleton type of the two lists are `b.w1.v1` and `b.w2.v1`. We have therefore to show that `b.w1.type` is equal to `b.w2.type`, which is indeed the case as the second one is upper bounded by the first one.

```
class A() { val v1: String = ""; val v2: v1.type = v1; }
class B() { val w1: A = new A(); val w2: w1.type = w1; }
val b: B = new B();
```

2.8 Virtual Types

A *virtual type* is a type declared within a class. Like a type parameter, it has a lower and an upper bound, which default respectively to `All` and `Any`. Subclasses can refine one or both of its bounds or assign it a type with a *type refinement*. In the code below, the class `C` declares a virtual type `Xs`, the class `D` refines its upper bound and the classes `E` and `F` assign it a type.

```
abstract class C[X]              { type Xs <: Collection[X]; }
abstract class D[X] extends C[X] { type Xs <: Set[X];        }
         class E[X] extends D[X] { type Xs  = HashSet[X];    }
         class F[X] extends D[X] { type Xs  = ListSet[X];    }
```

When a virtual type is refined, the new lower/upper bound has to be a super-type/subtype of the previous one. An assignment to a virtual type can be interpreted as a simultaneous refinement of its two bounds to the same type. This type has therefore to be both a super-type of the previous lower bound and a subtype of the previous upper bound. A virtual type that has already been assigned a type can neither be refined nor assigned another type. A class can be non-abstract only if all its virtual types have been assigned a type.

Virtual types can be extracted from instances of their declaring class. For example, if p is a stable path of type `C[Int]`, p`.Xs` is a valid type. It is called a *member type*. Its lower and upper bounds are, respectively, `All` and `Collection[Int]`. In general, the bounds depend on the type of the stable path. For example, if p was of type `D[Int]`, the upper bound would be `Set[Int]` and if it was of type `E[Int]`, the type p`.Xs` would be equal to `HashSet[Int]`.

Because of side-effects, successive evaluations of the same expression may return different instances. If these instances have a common virtual

type, it is not necessarily assigned the same type in all instances. For example, an expression *e* of type `C[Int]` may return an instance of `E[Int]` and later an instance of `F[Int]` whose virtual type `Xs` are assigned different types. Thus, *e*`.Xs` does not always represent the same type. This is unsound and for that reason prefixes of member types are restricted to stable paths. The code below shows what could happen if this restriction was lifted.

```
class A { def a(): Unit = (); }
class B { def b(): Unit = (); }
abstract class C {
  type T;     def t(): T;           def f(t: T): Unit;         }
class CA extends C {
  type T = A; def t(): T = new A(); def f(t: T): Unit = t.a(); }
class CB extends C {
  type T = B; def t(): T = new B(); def f(t: T): Unit = t.b(); }
var flip: Boolean = false;
def c(): C = { flip = !flip; if (flip) new CA() else new CB(); }
c().f(c().t())
```

In the last expression, the expected type and the actual argument types of the method `f` are both `c().T`. The call would therefore be legal if prefixes of member types were not restricted to stable paths. However, the two calls to `c` return respectively an instance of `CA` and one of `CB`, which assign different types to the virtual type `T`. The method `f` ends up calling the method `a` on an instance of `B`, which has no such method.

2.8.1 Typing Rules

First of all, it is important to understand that a virtual type is always selected on some instance. So, the type *T* within a class that defines or inherits the virtual type *T* is just a shortcut for the type **`this.`***T*.

The lower and upper bounds of a member type are computed in a very similar way to the type of a method call. First of all, like the type of a method call, the exact bounds depend on the type of the instance on which the virtual type is selected. The function *lookup* is therefore overloaded for virtual types. It works exactly like the one for methods but looks for type definitions and type refinements instead of method definitions and method implementations and it returns lower and upper bounds instead of return types. It handles type assignments as if they were type refinements with identical bounds. Like the return types, the bounds returned by the function *lookup* are passed to the function *as-seen-from* to replace

all type parameters by actual types and all occurrences of **this** by actual instances. The resulting bounds are the lower and upper bounds of the member type.

Like for type parameters, it is required that each virtual type is lower and upper bounded, possibly indirectly, by some class type. This is to avoid infinite lower and upper bounds. More precisely, in each class where a virtual type T is defined, refined or assigned, it is necessary to check that **this**.T is lower and upper bounded by some class type.

Lower and upper bounds of types may now also contain any number of type members. Subtyping tests and least upper bound computations must therefore also check whether the two compared lists contain equal member types. Two member types are equal if and only if they select the same virtual type on equal stable paths.

In principle, it would be necessary to adapt the function *as-seen-from* to handle member types. However, as in order to handle singleton types it does already replace occurrences of **this** by actual instances, there is nothing more to do. Fields and methods whose type contain member types, like those whose type contain singleton types, must be selected on stable paths because otherwise their type cannot be expressed as there would be no value with which replace occurrences of **this**.

2.9 Refined Type

Refined types are to virtual types what parameterized class types are to type parameters. They can be used to specify the value of a virtual type or to refine its bounds. Furthermore, they can also refine the return type of a method with a *method refinement* or the type of an immutable field with a *field refinement*. To illustrate this, let us consider the following class.

```
abstract class C { type T; val v: Any; def f[X](x: X): Any; }
```

The refined type `C { type T = Set[Int]; }` is the type of all values that are an instance of the class `C` and whose type member `T` is equal to `Set[Int]`. The type `C { type T <: Set[Int]; }` is slightly less restrictive as it requires only that the type member is a subtype of `Set[Int]`. Finally, `C { val v: String; def f[X](x: X): Set[X]; }` contains a field refinement and a method refinement that specify that the type of the field `v` must be a subtype of `String` and the return type of the method `f` a subtype of `Set[X]`. A field refinement is in fact a disguised method refinement. Indeed, it simply refines the return type of the field's getter method.

Refined types can only be used to specify or refine bounds or types of members inherited by the base class. They cannot be used to require the presence of additional members. For example, one could try to designate with the type `C { val w: String; }` instances of `C` that have an additional field `w` but this type is not legal.

2.9.1 Typing Rules

A refined type can be seen as a class type with some additional constraints. The introduction of refined types requires no modification of the typing rules but we have to take into consideration that the lower and upper class bounds of a type can now be refined types. This affects subtyping tests, the computation of least upper bounds and the function *lookup*.

The function *lookup* takes a class and a method or a virtual type and returns the return type of the method or the bounds of the virtual type in the given class. If a method or a virtual type is selected on a value whose type is upper bounded by a refined type, the function *lookup* should consider its refinements instead of just its class. If this refined type contains a refinement of the selected method or virtual type, the return type or the bounds of the refinement have to be returned. Note that a return type or bounds provided by a refined type do not have to be rewritten by a call to the function *as-seen-from* because, unlike types and bounds coming from a class definition, the refined type is part of the current context.

Testing whether a refined type T is a subtype of another refined type U is done by comparing the underlying class types and additionally testing that each constraint in U is fulfilled by T. Testing whether a virtual type bounds constraint **type** X `>:` R `<:` S is fulfilled is done by testing whether x`.`X is a supertype of R and a subtype of S where x is a fresh value of type T. Testing whether a method return type constraint of a method f is fulfilled is done in a similar way by verifying that the return type of the method f when selected on a value x of type T is a subtype of the one specified by the constraint.

With the introduction of refined types, it may now happen that the least upper bound of two class types is a refined type. This can be illustrated with the definitions below.

```
abstract class C[X]              { type Xs <: Collection[X]; }
         class E[X] extends C[X] { type Xs  = HashSet[X];    }
         class F[X] extends C[X] { type Xs  = ListSet[X];    }
```

Until now, the least upper bound of `E[X]` and `F[X]` was simply `C[X]`. With refined types, we can do better. Indeed, we can compute the least

upper bound of the upper bound of `Xs` in both types. This corresponds to the least upper bound of `HashSet[X]` and `ListSet[X]`, which is equal to `Set[X]`. The least upper bound of `E[X]` and `F[X]` is therefore the refined type `C[X] { type Xs <: Set[X]; }`.

The computation of the least upper bound of two class types or refined types T and U is now done by first computing their least upper class type R as was done until now. Then for each virtual type `X` inherited by R, the least upper bound of its bounds in T and U is computed. If this bound is a subtype of its upper bound in R, it is used to constrain the upper bound of X in R. Methods inherited by R and their return types are handled in a similar way. Lower bounds of virtual types inherited by R need also to be handled in the same way but for those a *greatest lower bound* has to be computed instead of a least upper bound. The computation of greatest lower bounds is similar to the computation of least upper bounds but with exchanged roles for upper and lower bounds.

An unfortunate consequence of the introduction of refined types is that the computation of least upper bounds may now never terminate.

```
abstract class C           { type X; }
abstract class E extends C { type X <: E; }
abstract class F extends C { type X <: F; }
```

For example, with the definitions above, the least upper bound of `E` and `F` is the infinite type:

```
C { type X <: C { type X <: C { type X <: /* ... */; }; }; }
```

The same problem occurs also with covariant type parameters. For example, if the class `C` had a covariant type parameter instead of its virtual type `X` and the classes `E` and `F` had respectively `C[E]` and `C[F]` for superclass instead of `C`, then the least upper bound of `E` and `F` would be the infinite type `C[C[C[...]]]`.

This implies that the typing of some if-expressions or the inference of some type arguments, which sometimes requires the computation of least upper bounds may not terminate. For this reason, the Scala compiler detects computations of least upper bounds that exceed a predefined nesting level and signals an error. These errors can however always be overcome by providing an explicit less precise upper bound. An if-expression can be annotated with an expected type such that the compiler can simply check that each alternative is of the right type instead of computing their least upper bound. Type inference of type arguments can be avoided by providing explicit type arguments.

2.10 Class Type Parameters vs. Virtual Types

Class type parameters and virtual types can be used to solve the same kind of problems. A solution based on type parameters is often less verbose than one based on virtual types. But, typing rules for type parameters are more complex than those for virtual types. Both sets of rules have to handle lower and upper bounds but rules for type parameters have to handle all the additional complexity that arises from parameter variance.

We show here that everything that can be expressed with class type parameters can also be expressed with virtual types and thus demonstrate that virtual types are indeed as expressive as type parameters. The basic idea is that each class type parameter becomes a virtual type with the same bounds. Type arguments of the superclass become type assignments to the corresponding virtual types and parameterized class types become refined types. Let us illustrate this with an example.

```
class C[T] { def f(): T = f(); }
class D[U] extends C[U];
val v: C[Int] = new D[Int]();
```

By applying our transformation to this example, we obtain the code below.

```
abstract class C { type T; def f(): this.T = f(); }
abstract class D extends C { type U; type T = this.U; }
val v: C { type T = Int; } = new D() { type U = Int; };
```

The two classes are now abstract because of the virtual types and the instance creation is now an anonymous class instance creation. The need for these two modifications could be avoided with two very simple changes to Scala. First of all, if instance creations were really considered as two step operations that first create an instance of some type and then initialize it by invoking one of its constructor, one could write `new D{type U = Int;}()`, which creates a new instance of the type `D{type U = Int;}`. This avoids the need of an anonymous class, but there is still the problem that we are creating an instance of an abstract class. This is solved by removing the obligation to declare a class abstract if it declares or inherits a non-assigned virtual type and instead check in each instance creation that all virtual types inherited by the new instance are assigned some type. With these two changes, one could write the following code:

```
class C { type T; def f(): this.T = f(); }
class D extends C { type U; type T = this.U; }
val v: C { type T = Int; } = new D{ type U = Int; }();
```

We still need to describe what to do with variant type parameters. The variance of a type parameter affects only the transformation of the arguments of parameterized class types. Arguments for invariant parameters are transformed into type assignments of the corresponding virtual type. Arguments for covariant (resp. contravariant) parameters are instead transformed into refinements of the upper (resp. lower) bound of the virtual type. If in our example both classes were covariant in their parameter, the transformation would yield the code below. It differs from the code above only in the type of v.

```
class C { type T; def f(): this.T = f(); }
class D extends C { type U; type T = this.U; }
val v: C { type T <: Int; } = new D{ type U = Int; }();
```

The relationship between type parameters and virtual types described in this section was first described by Thorup and Torgerson in [32].

2.10.1 Safety

In Section 2.6 we have seen that variant type parameters are safe only if some restrictions are placed on their usage. There are no similar restrictions on the usage of virtual types but we are able to present a way of transforming any type parameter into a virtual type. Does this mean that virtual types are unsafe? The answer is no and the reason is that the restrictions put on the usage of variant type parameters are somewhat too strong. Let us illustrate this with the transformation of the example of unsafe code given in Section 2.6.

```
abstract class C { type X; def f(x: this.X): Int; }
class D extends C { type X = Int; def f(x: Int): Int = x - 1; }
val c: C { type X <: Any; } = new D();
c.f("A");
```

The definition of the class `C` is here perfectly well-formed because virtual parameters are restricted to neither covariant nor contravariant positions. They can be used everywhere where a type is expected. The definitions of the class `D` and of the value `v` are also well-formed but the method call is not. The type of the argument `"A"` is `String`, which is a subtype of `Any`. The expected type of the argument is `c.X`, which is also a subtype of `Any`. So, both types are subtypes of `Any`, but they are otherwise unrelated. Nothing lets us deduce that `String` is a subtype of `c.X`. The method call is therefore ill-typed and thus there is no safety issue.

A consequence of the absence of restrictions on the usage of virtual types is that virtual types are more expressive than type parameters. In

the previous example, we could declare a method that is illegal with type parameters. One could object that the definition of this method in the class `C` is not very useful. Indeed, given an instance of type `C`, it will never be possible to invoke the method because it is impossible to obtain an argument of the right type. This is true for this example but it is not true in general. For example, if the class `C` was augmented with a method `x` that returns a value of type **`this`**`.X`, its result could be passed as an argument to the method `f`. The expression `c.f(c.x())` in the code below is well-formed.

```
abstract class C {
  type X;
  def f(x: this.X): Int;
  def x(): this.X;
}
class D extends C {
  type X = Int;
  def f(x: Int): Int = x - 1;
  def x(): Int = 0;
}
val c: C { type X <: Any; } = new D();
c.f(c.x());
```

All that demonstrates that the restrictions on the usage of variant type parameters are too strong. These restrictions are needed because when a type is provided as an argument for some type parameter, it is always assumed that the parameter is equal to the provided type. If instead that was assumed only for invariant parameters and it was otherwise assumed that the parameter is lower-bounded by the provided type for contravariant parameters and upper-bounded for covariant parameters, then the restrictions could be lifted.

2.10.2 Wildcards

In Java 5, there is no notion of parameter variance; all class type parameters are invariant. Instead, there are wildcards[35], which can replace type arguments in class types. They indicate that we are not interested by the exact type of the corresponding type parameters.

Let us consider a class `C` with a single type parameter `X`. The type `C<T>` is the type of all instances of `C` whose parameter `X` is equal to `T`. If we are not interested by the type `T` we can replace it by a wildcard: `C<?>` is the type of all instances of `C`. Wildcards can have either an upper or a lower

bound: `C<? extends T>` is the type of all instances of `C` whose parameter `X` is a subtype of `T` and `C<? super T>` is the type of all instances of `C` whose parameter `X` is a super-type of `T`.

All Java's wildcards can be expressed with Scala's virtual types. For instance, the types `C<? extends T>` and `C<? super T>` translate respectively to `C {type X <: T}` and `C {type X >: T}` and the type `C<?>` translates to `C`.

Java's wildcards are a bit more expressive than Scala's type parameters with variance and usually also a bit more verbose. But, they are less expressive than Scala's virtual types. For example, here is what we get if we try to rewrite the last example of the previous section in Java.

```
abstract class C<X> { int f(X x); X x(); }
class D extends C<Integer> {
  int f(Integer x) { /* ... */ };
  Integer x() { /* ... */ }
}
C<?> c = new D();
c.f(c.x()); // error
```

The two class definitions and the value definition are well-typed, but not the method call. Both, the type of the argument and the expected type are `C<?>`, but that is not enough. Indeed, the call is safe only if the type parameter `X` of the two types are equal and this is not true for two arbitrary values of type `C<?>`. In Scala, the call is possible because the compiler is able to determine that in both types, the parameter `X` is equal to `c.X`.

2.10.3 Constructors

Although we stated that virtual types were more expressive than type parameters, there is one place where type parameters are more expressive than virtual types: constructors. Here is an example:

```
class C[X](_x: X) { val x: X = _x; }
```

If we try to rewrite this with a virtual type, we obtain something like this:

```
class C(_x: this.X) { type X; val x: this.X = _x; }
```

This is not legal because the current instance is not available within the parameter of the class. On one hand, this makes sense because syntactically the class parameters are not within the class body. On the other hand, if we desugar the class definition, we obtain the code below, which seems perfectly legitimate. One could therefore argue that the code above should

be accepted. This would make virtual types strictly more expressive than type parameters.

```
primitive class C {
  type X;
  val x: this.X = _;
  def this(_x: this.X) = { super(); this.x = _x; }
}
```

2.11 Inner Classes

A class can be defined directly within another one. Such a class is called a *nested class*. Compared to other top-level classes, nested classes have privileged rights to access the members of their enclosing class; because they are defined within it, they may access all its private and protected members even if they are otherwise unrelated to it. If in addition to that, the nested class may also access the current instance of its enclosing class, it is called an *inner class*. Nested classes that are restricted from that are called *static nested classes*. In Scala, all nested classes can access the current instance of their enclosing class. Therefore all nested classes are also inner classes. This contrasts with Java where both kinds of nested classes can be defined.

Within an inner class the current instance of its enclosing class is called the *current enclosing instance* or simply the *enclosing instance*. Given an instance x of an inner class, it is called the *enclosing instance of x*.

The remaining of this section reminds some facts about inner classes and describes the specificities of Scala's inner classes. As the different aspects of inner classes are slightly more entangled in Scala than in Java, we first start by reminding some facts about Java's inner classes.

2.11.1 Enclosing Instances

In Java, within an inner class `I` nested in a class `C`, the expression `C.this` denotes the current instance of the enclosing class `C`. The code below defines an inner class `I` nested in a class `C`. The inner class makes explicit the presence of the enclosing instance by defining the field `outerI` and initializing it with that instance.

```
class C { class I { final C outerI = C.this; } }
```

The field `outerI` is here explicitly defined but every inner class really has a hidden field that holds its enclosing instance. The syntax `C.this` is

just a way of accessing this hidden field. In fact, an inner class is nothing more than a static nested class with an additional hidden field holding its enclosing instance. We call this field the *outer field* of the inner class.

An inner class defined within another inner class can access the current instance of both of its enclosing classes. For example, within the class `M` defined below, the expression `C.this` denotes the current instance of its (indirectly) enclosing class `C`. By definition, this instance is equal to the enclosing instance `C.this` of its (directly) enclosing class `I`. In other words, within the class `M`, `C.this` is equal to `this.outerM.outerI`.

```
class C {
  class I {
    final C outerI = C.this;
    class M {
      final I outerM = I.this;
    }
  }
}
```

Although the nesting of inner classes may be arbitrarily deep and an inner class may reference the current instance of any of its enclosing classes, a single outer field per class is always sufficient to access all these instances. For example, let us consider a top-level class C_0 and n inner classes $C_1, \ldots, C_n$ where each inner class C_i is defined in the class C_{i-1}. Let us also assume that each inner classes C_i has an outer field `outer`C_i. The enclosing instance C_i`.this` within the class C_n is then equal to the value returned by the expression `this.outer`C_n`.`$,\ldots,$`.outer`C_{i+1}. This demonstrates that with explicit outer fields, any enclosing instance expression C`.this` can be desugared into a succession of field selections.

2.11.2 Instance Creations

To create a new instance of an inner class `I`, an instance of its enclosing class `C` has to be provided. In Java, the syntax *expr*`.new I(`*args*`)` is used to that effect. It creates a new instance of the class `I` whose enclosing instance is the result of the evaluation of the expression *expr*. The enclosing instance may be omitted if the instance creation is enclosed, possibly indirectly, in a subclass `D` of `C`. In that case, the expression `new I(`*args*`)` is desugared into `D.this.new I(`*args*`)`. We illustrate this by augmenting the class `C` above with the two fields defined below. The value `i` is an instance of the class `I` whose enclosing instance is the current instance of the class `C` (the expression `new I()` is syntactic sugar for `this.new I()`) and

the value `m` is an instance of the class `M` whose enclosing instance is the value `i`.

```
final I   i = new I();
final I.M m = i.new M();
```

Every inner class introduces a single outer field, but an instance of an inner class may have several outer fields because it inherits one from each inner class it is an instance of. For example, every instance of the class `J` defined below has two outer fields, namely `outerI` and `outerJ`, and every instance of the class `N` has the two outer fields `outerM` and `outerN`.

```
class C {
  class I {
    final C outerI = C.this;
    class M {
      final I outerM = I.this;
    }
    class N extends M {
      final I outerN = I.this;
      N(I i) { i.super(); }
    }
  }
  class J extends I {
    final C outerJ = C.this;
  }
  final I   i = new I();
  final J   j = new J();
  final I.N n = i.new N(j);
}
```

For the same reason instance creations of inner classes require an enclosing instance, super constructor calls of inner classes also require an enclosing instance. The syntax *expr*`.super(`*args*`)` is used for these super constructor calls where *expr* must evaluate to an instance of the enclosing class `C` of the super class. Like with instance creations, the enclosing instance may be omitted if there is an enclosing class `D` that is a subclass of `C`. In that case, the expression `super(`*args*`)` is equivalent to `D.this.super(`*args*`)`.

In the code above, the class `J` has no explicit constructor; it gets a default constructor containing a call to the super constructor `super()`, which is here equivalent to `C.this.super()`. This implies that for any instance of class `J`, its two outer fields hold exactly the same value. The constructor of class `N` specifies that the enclosing instance of its superclass `M` is its argu-

ment `i`. Therefore, the two outer fields of a given instance of class `N` may hold different values. For example, for the value `n`, `n.outerN` is equal to `i` while `n.outerM` is equal to `j`. In fact, the two values are even of different types.

2.11.3 Scala's Inner Classes

In Scala, an inner class I whether it occurs in a type, in an extends clause or an instance creation must always be qualified by a type T. The syntax of *qualified class types* is the following: T`#`I. The *class qualifier* T specifies the type of the enclosing instance of the class I. It has to be at least a subtype of the enclosing class of I. As class qualifiers are often singleton types, there is a less verbose syntax for these cases: p`.`I can be used instead of p`.type#`I. Furthermore, if p is simply an enclosing instance C`.this` it can be entirely omitted. Here are some examples:

```
class C()             { class I(); }
class D() extends C() { class J() extends D.this.I(); }
val d: D = new D();
class K extends d.J();
val j: d.J = new d.J();
```

In an extends clause, the qualifier of the superclass plays two roles. It indicates the type of the enclosing instance inherited from the superclass and it specifies also that instance. For that reason, it has to be a singleton type. For the same reasons, the qualifier of the instantiated class in instance creations has also to be a singleton type. This can be observed in the following desugared version of the code above. The field `outerJ` denotes the implicit outer field of the inner class `J`.

```
primitive class C {
  def this() = super();
  primitive class I {
    def this() = super();
  }
}
primitive class D extends C {
  def this() = super();
  primitive class J extends D.this.I {
    def this() = D.this.super();
  }
}
val d: D = { val tmp = new D; tmp.this(); tmp }
```

```
primitive class K extends d.J {
  def this() = d.super();
}
val j: d.J =
  { val tmp = new d.J; tmp.outerJ = d; tmp.this(); tmp }
```

One can see that the `d` of the instance creation `new d.J()` is used in both the type of the new instance and the initialization of the outer field of that instance. Similarly, the `d` in the extends clause of the class `K` appears in the supertype of `K` and in the call to the constructor of the superclass. To be more in line with the desugared version of inner class instance creations, this call could also be desugared as follows:

```
  def this() = { this.outerJ = d; super(); }
```

With the usage of singleton types as class qualifiers it is possible to specify the exact value of inherited and instantiated inner classes. For example, in the previous example the type system knows that for any instance of `K`, its inherited outer fields from `I` and `J` are both equal to `d`. Being able to know the exact value of an outer field and thus prove that different outer fields necessarily contain the same value is very useful, if not vital, in the presence of virtual types. For example, in the code below, the method call `g(f())` is legal only because it can be proven that for any instance of `J`, the outer field it inherits from the class `I` is equal to the one it inherits from the class `J`. Thus, within the class `J`, the return type of the method `f` and the type of the parameter of the method `g` are both equal to `D.this.T` and the method call is therefore legal.

```
abstract class C() {
  type T;
  abstract class I() { def f(): T; }
}
abstract class D() extends C() {
  def g(t: T): Unit = ();
  abstract class J() extends I() { def test(): Unit = g(f()); }
}
```

If like in Java 5 it would only be possible to specify that the outer field inherited from `I` is of type `D`, the call would not be possible.

The usage of singleton types makes it possible to give a much more precise type to inherited outer fields in Scala than in Java. But strangely, in Scala, it is impossible to be just as precise as in Java. Or, in other words, in Scala, it is impossible to be as imprecise as in Java. For example, it is impossible to write valid Scala code whose desugared version would be the

following one although this code seems perfectly legitimate. This impossibility is however only due to the Scala syntax, which forces us to use one class qualifier to specify two conceptually different things: the compile-time qualifier of the superclass and the runtime enclosing instance of the superclass.

```
primitive class C {
  primitive class I;
}
primitive class D extends C {
  def this() = super();
  primitive class J extends D#I {
    def this(d: D) = d.super();
  }
}
```

One could try the code below but it is illegal because in extends clauses, class parameters are available only in the arguments of the superclass, not in its qualifier. If this code was legal it would still be more precise than the code above as the type system would still know the exact identity of the enclosing instance of the superclass. It would also be very problematic as the supertype of the class `J` would depend on one of its value argument. This poses similar problems to field and method types referencing class value parameters discussed in Section 2.7.1.

```
class C()             { class I(); }
class D() extends C() { class J(d: D) extends d.I(); }
```

2.11.4 Typing Rules

Class qualifiers are covariant; $T\#C$ is a subtype of $U\#C$ if and only if T is a subtype of U. This tells us how to compare two class types of the same inner class. For class types of different classes, the comparison still relies on the function *base-type* but this function must now not only compute the type arguments of a given class implied by a given type but also its qualifier if the class is an inner class.

Until now the supertype of a class type T of some class C was computed simply by replacing in the declared supertype of the class C all type parameters of C with the type arguments provided by the type T. The problem with inner classes is that their declared supertype may contain type parameters of enclosing classes and also references to the current instance of these classes. These parameters and current instances must all be replaced with the types and instances implied by the type T. This looks

suspiciously like what the function *as-seen-from* does. And, a call to that function is also exactly what is required here. Indeed, if C is an inner class, its declared supertype can be seen as a type that occurs in the enclosing class B of C. It is therefore normal that it is handled like the declared type of fields and methods of the class B. If S is the declared supertype of C and U is the qualifier of the class C in T, then the supertype of T is obtained by computing *as-seen-from*(S, B, U) and replacing in the resulting type all the type parameters of C by the type arguments of T. If C is a top-level class, then the supertype of T is computed as before.

The function *as-seen-from* needs also to be updated. Indeed, types occurring in an inner class may not only reference the instance and the type parameters of the current class but also those of all its enclosing classes. All these references need also to be replaced with some actual instances and types. Assuming the function is invoked with the arguments T, C and U where T is the type to rewrite, C the class where it occurs and U the type of an instance of C the function works as follows. First, if T contains references to the current instance of C, U has to be a singleton type p`.type` and all those references are replaced with p. Then, the type $S = \mathit{base\text{-}type}(U, C)$ is computed and all type parameters of C in T are replaced with the type arguments of S. This is just the same as before. It yields a type T'. If C is a top-level class, the type T' is returned. Otherwise, C is an inner class defined within some class B and the type S has a class qualifier R that is a subtype of B. The result of the function *as-seen-from* is the result of the recursive call *as-seen-from*(T', B, R).

Let us apply all this to the typing of the expression `j.f()` in the context of the definitions below.

```
class C[V]() {
  class I[W]() {
    def f(): Tuple4[V, C.this.type, W, I.this.type] = f();
  }
}
class D[X]() extends C[X]() {
  val c: C[List[X]] = new C[List[X]]();
  class J[Y]() extends c.I[Y]();
}
val d: D[Int] = new D[Int]();
val j: d.J[String] = new d.J[String]();
```

The type of `j.f()` is computed with the call

as-seen-from(`Tuple4[V, C.this.type, W, I.this.type]`, `I`, `j.type`)

The instance `I.this` is of course replaced with `j`. For the other arguments

of `Tuple4`, we have to compute *base-type*(`j.type`, `I`). The type `j.type` is upper bounded by `d.J[String]` whose supertype is obtained by computing

as-seen-from(`c.I[Y]`, `D`, `d.type`)

and replacing `Y` with `String`. The type `c.I[Y]` is in fact a shortcut for `D.this.c.I[Y]` and the actual instance of `D.this` is `d`. Thus, the supertype of `d.J[String]` is `d.c.I[String]`. This lets us replace the third argument of the `Tuple4` with `String`. It tells us also that the actual instance for `C.this` is `d.c`. To compute the actual type for `V`, we have to compute *base-type*(`d.c.type`, `C`). This yields `C[List[Int]]`, which lets us conclude that the type of `j.f()` is

```
Tuple4[List[Int], d.c.type, String, j.type]
```

We have seen that fields and methods whose type contains singleton types or member types can only be selected on stable paths because otherwise their type cannot be expressed. With inner classes, this gets even worse. Indeed, there are situations where a field or a method cannot be used at all. For example, in the function `test` below, the type of the method call `i.f()` cannot be expressed. This call is therefore forbidden although it seems perfectly legitimate.

```
abstract class C() {
  type T;
  class I() { def f(): C.this.T = f(); }
}
def test(i: C#I): Unit = { i.f(); () }
```

This situation is a bit frustrating because, in principle, we know what the type of the method call is: it is the type `T` of the enclosing instance of `i`. With explicit outer fields, it would be `i.outerI.T`. So, the only problem here is that without explicit outer fields, this type cannot be named.

2.11.5 Qualified Class Types as Refined Types

If outer fields were explicit, it would be possible to replace qualified class types with refined types. Indeed, the type *T#I* could be replaced with the refined type *I* `{ val outer`*I*`: `*T*`; }`. With this scheme, subclasses of inner classes would implicitly refine the outer field of their superclass. For example, the definition

```
class J() extends p.I() { /* ... */ }
```

would be translated into the following one

```
class J() extends I() { val outerI: p.type = p; /* ... */ }
```

2.12 Mixins

In order to overcome the limitations of single inheritance, Scala supports mixin composition. This is similar to the implementation of interfaces in Java but more powerful because unlike interfaces mixed-in classes can contain implemented methods.

The code below defines an abstract class `Map` and two implementations `HashMap` and `TreeMap`. It defines also a trait `SynchronizedMap`, which contains the additional behavior (code) needed to transform any `Map` into a synchronized `Map`. A synchronized `HashMap` is obtained by defining a subclass of `HashMap` that mixes-in the trait `SynchronizedMap`. The same trait can be reused in the same way to define a synchronized version of `TreeMap`.

```
abstract class Map[K,V]() {
  def insert(k: K, v: V): Unit;
  def lookup(k: K): Option[V];
}
class HashMap[K,V]() extends Map[K,V]() { /* ... */ }
class TreeMap[K,V]() extends Map[K,V]() { /* ... */ }

trait class SynchronizedMap[K,V]() extends Map[K,V]() {
  override def insert(k: K, v: V): Unit =
    synchronized (this) { super.insert(k, v); }
  override def lookup(k: K): Option[V] =
    synchronized (this) { super.lookup(k); }
}
class SynchronizedHashMap[K,V]()
  extends HashMap[K,V]() with SynchronizedMap[K,V]();
class SynchronizedTreeMap[K,V]()
  extends TreeMap[K,V]() with SynchronizedMap[K,V]();
```

The traits and the mixin composition mechanism of Scala are loosely modeled on the description of traits as composable units of behavior by Schärli [31, 30]. In Scala, normal classes can also be used as mixins but in a more limited way. While traits can be inherited multiple times through different inheritance paths, mixed-in classes can only be inherited once by a given class. This restriction is needed to avoid ambiguities that could arise if the class has a state. For example, it would be necessary to deter-

mine whether the state should be duplicated or not. And, if the state is not duplicated, it must be decided which inheritance path will initialize it.

With mixins, it may happen that a value should be a subtype of two different class types. For example, one could define a trait `LoggedMap`, which logs all operations of the class `Map`. It could then happen that in some context, a value has to be an instance of both traits `SynchronizedMap` and `LoggedMap`. This can be expressed with *compound types*. For example, the type of a `Map` from `Int` to `String` that is both synchronized and logged is the following one:

```
SynchronizedMap[Int,String] with LoggedMap[Int,String]
```

With the introduction of mixins and compound types a class can be inherited through different paths. This needs to be taken into account in the implementation of the function *base-type*. It raises also some new typing issues. For example, a trait with a type parameters that is inherited more than once through different inheritance paths could be inherited with different type arguments. This must be forbidden. Similarly, a virtual type could be refined in different ways through different inheritance paths. In that case, there must be a refinement that refines all others. There are also some semantics issue. For example, a class can inherit multiple implementations for a same method. In that case there must be a way to determine which one will be used.

2.13 Explicit Self Types

Class definitions can specify an *explicit self type*. This type specifies the type of the current instance **`this`**. In the code below, the class `BaseNode` is defined with the explicit self type `Node`. It implies that within the class `BaseNode`, the current instance **`this`** has the type `Node` instead of the type `BaseNode` that it would have in the absence of explicit self type. The implementation of the method `self` is therefore legal.

```
abstract class Graph() {
  type Node <: BaseNode;
  abstract class BaseNode(): Node { def self(): Node = this; }
}
class  LabeledGraph() extends Graph() {
  type Node = LabeledNode;
  class LabeledNode() extends BaseNode();
}
```

Without explicit self type, it would be impossible to implement the method `self` within the class `BaseNode`. Instead, it would be necessary to implement it in the subclass `LabeledNode`. If there were other subclasses, it would be necessary to repeat the implementation in each subclass.

Any subclass of a class with an explicit self type must be defined such that the type of its current instance **`this`** is a subtype of the explicit self type of its superclass. This may imply that it needs itself an explicit self type. Any class with an explicit self type has also to be abstract.

It is important to understand that the explicit self type of a class applies only to its current instance. For example, the current instance of the class `BaseNode` can be passed where an instance of `Node` is expected but not any arbitrary value of type `BaseNode`. The explicit self type of a class is not a supertype of that class.

The example above demonstrates that with explicit self types it is possible to avoid some code duplication but it remains rather academic. An more realistic example and probably also a more convincing one of the usefulness of explicit self types can be found in [27].

Chapter 3

Lambda Lift

Local function and class definitions are not supported by the Java virtual machine. This chapter describes how they are transformed into method and inner class definitions. The technique used to perform this transformation is similar to the well-known lambda lifting technique [15] used to lift local function definitions to the top-level in functional languages.

3.1 Introduction

Functions are usually implemented with a stack. Each time a function is invoked, a new portion of the stack is allocated to the function. This stack portion is called the function's activation record. It constitutes the working memory of the function and contains, among other things, the function's arguments, local variables and return address. It is deallocated when the function returns.

For global functions, all the values accessed by their code are either global values or can be found in their activation record. That's different for local functions, at least if like in Scala local functions may reference the arguments and the local variables of their enclosing functions. Those values will not be found in the function's own activation record but in the activation record of the corresponding enclosing function. Let us illustrate this with the following functions.

```
def fold(ls: List[Int], z: Int, f: (Int, Int) => Int): Int = {
  def loop(x: Int, ys: List[Int]): Int =
    if (ys.isEmpty) x else loop(f(x, ys.head), ys.tail);
  loop(z, ls)
}
```

The local function `loop` contains the expression `f(x, ys.head)`, which

references three variables: `f`, `x` and `ys`. Two of those, `x` and `ys`, are of its own arguments and can therefore be found in its own activation record, but `f` is an argument of the enclosing `fold` function. It has to be retrieved from an activation record of that function. This implies that each time the `loop` function is invoked, it must be passed a reference to the activation record of its enclosing `fold` function.

One technique to implement local functions is to transform them into global functions but with an additional argument containing a reference to the activation record of their enclosing function. Local functions with more than one enclosing function can retrieve the activation record of non-directly enclosing functions by starting with the one received as an argument and extracting from it the one of the next enclosing function and so on, until the desired one is reached.

A major drawback of this technique is that it does not work for local classes. Local classes could be augmented with an additional field containing a reference to the activation record of their enclosing function. The problem is that instances of local classes may still exist after the enclosing function has returned. Such instances would have a reference to a deallocated activation record. The same problem arises if functions are first class values and references to local functions outlive the invocation of their enclosing function. Those problems could be solved by allocating activation records on the heap and letting the garbage collector take care of them instead of automatically deallocating them on function returns, but that is a rather radical change that is also prone to introduce space leaks.

Another drawback of this technique is that it is usually not expressible in the source language because activation records are usually not directly accessible by the programmer. This prevents the compiler from performing a source to source transformation to eliminate all local functions and thus simplify the work of the rest of the compiler.

Lambda lifting is a technique that avoids these problems. This technique transforms a functional program with local function definitions into a program consisting only of global, possibly recursive, function definitions. It was first described by Augustsson in [3] and later formalized by Johnsson in [15]. It is similar to the technique described above but instead of passing down to the local functions the activation record of their enclosing function, it passes them down the value of all variables of their enclosing functions that they access.

In our example, the `loop` function accesses the argument `f` of the enclosing `fold` function. The value of this argument can be passed down to the `loop` function by augmenting it with an additional argument `f$`. The problematic reference to `f` can then be replaced with a reference to this

new f$ argument. The whole loop function definition can then be moved to the top-level. This yields the following code.

```
def loop(f$: (Int, Int) => Int, x: Int, ys: List[Int]): Int =
  if (ys.isEmpty) x else loop(f$, f$(x, ys.head), ys.tail);
def fold(ls: List[Int], z: Int, f: (Int, Int) => Int): Int =
  loop(f, z, ls);
```

The lambda lifting technique works also for local classes; local classes are augmented with fields containing the values of the accessed variables of the enclosing functions. If some instances of a local class outlive the invocation of their enclosing function, they can still refer the variables of that function through their local copy in the additional fields.

Another advantage of this technique is that it can be expressed as a source to source transformation.

The lambda lifting technique has however also some disadvantages in comparison to the technique based on explicit activation records. First of all, it does not work for mutable variables. As variables of enclosing functions are accessed through local copies, modification of those variables would only have a local effect. Another disadvantage of this technique is that it usually requires more stack space. In our example, the loop function was augmented with a single argument, but a local function that accesses several variables of its enclosing functions, needs an extra argument for each of it whereas with the explicit activation records technique only one extra argument is needed. This may significantly increase the used stack space, especially for recursive functions, because each recursive call requires its own copy of those variables.

On the Java virtual machine, the programmer has no direct access to the stack. For example, it is impossible to create a reference to some stack position. This prevents the application of the technique based on explicit activation records. Therefore, the Scala compiler systematically uses the lambda lifting technique to eliminate local functions and local classes even when the usage of an explicit activation record would be more appropriate.

Scala functions do not appear on the top-level but within classes; therefore, local functions and local classes are not moved to the top-level but into their next enclosing class. In some sense, each class successively acts as the top-level for its methods and thus local functions are turned into new methods and local classes into new inner classes.

Section 3.2 describes the classical lambda lifting algorithm for functional languages. Section 3.3 describes how the technique is generalized to classes and Section 3.4 how mutable variables are treated. Types and sev-

eral typing issues are discussed in Section 3.5. Finally, Section 3.6 presents an alternative method, which reduces the stack usage and can also solve some typing issues.

3.2 Classical Algorithm

The classical lambda lifting algorithm transforms a set of nested function definitions into a set of global function definitions. It assumes that all variables are immutable. The algorithm consists of four successive steps. First, the set of variables for which an extra parameter is needed is computed for each local function. Local function definitions are then augmented with those extra parameters and calls to those functions are accordingly augmented with extra arguments. Then, in the body of each local function, all references to variables for which the function got an extra parameter are replaced with references to those parameters. Finally, all local function definitions are moved to the top-level. The first step is the most complex one. The three others are rather straightforward and are usually performed all at once in real implementations.

3.2.1 Computing Extra Parameters

The set of variables for which a function needs an extra parameter in order to be lifted to the top-level is called its *extra set*. In our example, the extra set of the function `loop` is just its unique free variable `f`. The extra set of a function contains always all its free variables but it is generally not limited to those variables. It may also contain variables that would be free within the function but do not actually occur in its body. This is illustrated by the following example.

```
def f(u: Int, v: Int, w: Int): Int = {
  def g(x: Int): Int = x + v;
  def h(y: Int): Int = g(y) + i(y);
  def i(z: Int): Int = { def j(): Int = h(z) + w; j() }
  i(u)
}
```

The function `h` contains no free variables but it calls the function `g`, which contains the free variable `v`. If `g` is augmented with an extra parameter for the variable `v` in order to lift it to the top-level, its call in `h` has to be augmented with the extra argument `v`. Thus, `v` becomes free in `h` and has to be included in its extra set. This demonstrates that the extra set of a function depends on the extra set of the functions it calls.

The extra set of a function f has to contain all the free variables that occur in the body of f. Furthermore, it must also contain all the variables that occur in the extra set of the functions called by f and that are free in f. The constraint that such variables are free in f is not necessarily satisfied for extra sets of nested functions. For example, the function `i` calls the function `j` whose extra set contains `z` but `z` is not free in `i` and is therefore not added to its extra set.

The *call set* of a function consists of all the functions called in its body. Together, the call sets of all functions form the *call graph* of the program. The extra sets are computed by first initializing the extra set of each function with its free variables and then visiting each node of the call graph. For each visited function f and each function g in its call set, the variables of the extra set of g that are also free in f are added to the extra set of f. If the call graph is cyclic, the graph traversal needs to be repeated until a fixed point is reached, which necessarily happens as there is a fixed number of variables in the program. If the graph is acyclic only one traversal is needed, provided called functions are always visited before calling functions.

The array below gives the call sets and the successive extra sets of the four local functions of our example. The successive extra sets are computed here by visiting the functions in alphabetical order.

function	call set	extra set 0	extra set 1	extra set 2	extra set 3
`g`	ϵ	`v`	`v`	`v`	`v`
`h`	`g,i`	ϵ	`v`	`v,w`	`v,w`
`i`	`j`	ϵ	`w`	`v,w`	`v,w`
`j`	`h`	`w,z`	`v,w,z`	`v,w,z`	`v,w,z`

Danvy and Schultz observed in [7] that functions that are defined in the same scope and that are strongly connected[1] in the call graph necessarily have identical extra sets. They show that this implies that it is always possible to simplify the call graph in order to remove all cycles and thus enable the computation of the call sets in a single graph traversal.

The call graph is simplified in two steps. First all calls to nested functions are removed from the call graph such that the call set of any function f contains only functions defined in the same scope as f or in an enclosing scope. In counterpart, the call set and free variables of each function are augmented with those of its nested functions.

[1] Nodes of a directed graph are strongly connected if there is a path from each node to every other node.

In our example, the call from `i` to its nested function `j` is removed. In counterpart, the call set and the free variables of `i` are augmented with the call to `h` and the free variable `w` of `j`.

function	call set	free variables
g	ϵ	v
h	g,i	ϵ
i	h	w
j	h	w,z

Strongly connected functions in the new call graph are necessarily defined in the same scope. Such functions always have identical extra sets. It is therefore possible to treat them as a single function whose call set and free variables are the union of those of the individual functions. This removes all the cycles from the call graph. All extra sets can then be computed with a single traversal of the new graph.

In our example, the functions `h` and `i` are strongly connected. If they are merged, all the extra sets can be computed with a single traversal of the call graph by first visiting, `g`, then the merged `h` and `i` and finally `j`.

function	call set	free variables	extra set
g	ϵ	v	v
h,i	g	w	v,w
j	h	w,z	v,w,z

3.2.2 Adding Extra Parameters and Arguments

Once the extra sets of all functions are known, the function definitions are augmented with a new parameter for each variable in their extra set. At the same time, all function calls are also augmented with a new argument for each variable in the extra set of the called function. The new arguments are nothing else than the variables in the extra set themselves. The transformation of our example yields the code below.

```
def f(u: Int, v: Int, w: Int): Int = {
  def g(v$g: Int, x: Int): Int = x + v;
  def h(v$h: Int, w$h: Int, y: Int): Int = g(v, y) + i(v, w, y);
  def i(v$i: Int, w$i: Int, z: Int): Int = {
    def j(v$j: Int, w$j: Int, z$j: Int): Int = h(v, w, z) + w;
    j(v, w, z)
  }
  i(v, w, u)
}
```

The transformed program is valid only if the variables added in the function calls are indeed accessible at the call sites. This is necessarily true. Indeed, the added variables are in the extra set of the called functions. They are therefore free in those functions and thus defined in the same scope or an enclosing scope of where the functions are defined. As functions can only be called from the same scope or one nested in the scope of its definition, the variables are also accessible at the call sites.

3.2.3 Substituting References to Free Variables

With the addition of extra parameters and arguments the references to free variables have not suddenly vanished. On the contrary, the extra arguments have probably increased their number. However, all functions have now an extra parameter for each reference to a free variable occurring in their body. All references to free variables can therefore be eliminated simply by replacing them with references to the corresponding extra parameter. This results in the code below for our example.

```
def f(u: Int, v: Int, w: Int): Int = {
  def g(v$g: Int, x: Int): Int = x + v$g;
  def h(v$h: Int, w$h: Int, y: Int): Int =
    g(v$h, y) + i(v$h, w$h, y);
  def i(v$i: Int, w$i: Int, z: Int): Int = {
    def j(v$j: Int, w$j: Int, z$j: Int): Int =
      h(v$j, w$j, z$j) + w$j;
    j(v$i, w$i, z)
  }
  i(v, w, u)
}
```

3.2.4 Lifting Functions

Function bodies reference now exclusively parameters and local variables of their function. Nesting is therefore no longer relevant. All nested functions can be moved to the top-level. This gives the following code for our example.

```
def f(u: Int, v: Int, w: Int): Int = i(v, w, u);
def g(v$g: Int, x: Int): Int = x + v$g;
def h(v$h: Int, w$h: Int, y: Int): Int = g(v$h, y)+i(v$h, w$h, y);
def i(v$i: Int, w$i: Int, z: Int): Int = j(v$i, w$i, z);
def j(v$j: Int, w$j: Int, z$j: Int): Int = h(v$j, w$j, z$j) + w$j;
```

3.3 Generalization to Classes

In Scala, there are not only local functions but also local classes. This section describes how the lambda lifting technique is generalized to local classes. It is important to keep in mind that lambda lifting aims only at eliminating local definitions, not inner classes, which remain in place. In fact, lambda lifting preserves the nesting of classes; if a class is nested within some class it will still be after lambda lifting. Thus references to enclosing instances are unaffected by lambda lifting.

3.3.1 Lifting Local Classes

Like local functions, methods of local classes can reference free variables. For example, in the code below, the method `m` of the local class `D` references the free variable `x`.

```
abstract class C() { def m(): Int; }
def f(x: Int): C = {
  class D() extends C() { def m(): Int = x; }
  new D()
}
```

In order to lift a local class, all references to free variables in its methods must be eliminated. Treating methods like normal functions and augmenting them with extra parameters obviously does not work. Our example would be transformed into the code below where the class `D` is no longer a correct implementation of `C`.

```
abstract class C() { def m(): Int; }
class D() extends C() { def m(x$D: Int): Int = x$D; }
def f(x: Int): C = new D();
```

When a local class is lifted, the signature of its methods cannot be changed. References to free variables are instead eliminated by adding a new field to the local class for each referenced free variable. To initialize these new fields, constructors of the class are augmented with extra parameters and instance creations with extra arguments. This produces the following code for our example.

```
abstract class C() { def m(): Int; }
class D(val _x$D: Int) extends C() {
  val x$D: Int = _x$D;
  def m(): Int = this.x$D;
}
def f(x: Int): C = new D(x);
```

3.3.2 Constructors

Although constructors look like special methods, they are treated very differently. First of all, unlike methods, their signature changes when their class is lifted; they get extra parameters for the extra fields of their class. The second difference is that references to free variables within constructors do not contribute to the extra fields of their class. They contribute only to the extra parameters of the constructor in which they occur. In fact, constructors are treated as if they were functions defined in the same scope as their class. The following example illustrates this.

```
class C(m: Int);
def f(x: Int, y: Int): C = {
  class D(n: Int) extends C(n: Int) {
    val v: Int = x;
    def this() = this(y);
  }
  new D()
}
```

When lifting a class, it is important to consider its desugared version. The desugared version of our example is given below. It clearly shows that the references to the free variables x and y occur both in constructors of D.

```
primitive class C { def this(m: Int) = {} }
def f(x: Int, y: Int): C = {
  primitive class D extends C {
    val v: Int = _;
    def this(n: Int) = { super(n); this.v = x; }
    def this() = this(y);
  }
  new D()
}
```

As the references to x and y occur in constructors, they should only contribute to the extra parameters of these constructors and not to the extra fields of D. This is indeed the case, as demonstrated by the code below were D has been lifted to the top-level.

```
primitive class C { def this(m: Int) = {} }
primitive class D extends C {
  val v: Int = _;
  def this(x$D: Int, n: Int) = { super(n); this.v = x$D; }
  def this(x$D: Int, y$D: Int) = this(x$D, y$D);
```

```
}
def f(x: Int, y: Int): C = new D(x, y);
```

3.3.3 Inner Classes

Inner classes are not lifted out of their enclosing class. They remain in place but possible references to free variables contribute to the extra fields of their enclosing class and are replaced with references to these fields.

```
def f(x: Int): Any = {
  class D() { class E() { def m(): Int = x; } }
  new D()
}
```

In the code above, the inner class `E` references the free variable `x`. In the lambda lifted code below, this reference is replaced with a reference to the extra field `x$D` of the enclosing class `D`.

```
class D(_x$D: Int) {
  val x$D:Int = _x$D;
  class E() { def m(): Int = D.this.x$D; }
}
def f(x: Int): Any = new D(x);
```

3.3.4 Local Definitions

Methods and inner classes of a local class can themselves contain local definitions. In that case the local definitions are first lifted within the local class and only then the local class is lifted. The lifting of the local definitions turns the functions and the classes constituting them into methods and inner classes of the local class. During this lifting, all references to free variables defined in an enclosing scope of the local class are treated as references to global variables and are thus left unmodified.

In the code below, the method `m` of the local class `D` contains a local function `g` and a local class `E`. The function `g` references the variables `x` and `y`, which are both free. The first one is defined in an enclosing scope of the local class `D` while the second one is defined within `D`.

```
def f(x: Int): Any = {
  class D() {
    def m(y: Int): Any = {
      def g(): Int = x + y;
      class E() { def n(): Int = g(); }
```

```
      new E()
    }
  }
  new D()
}
```

Before the class `D` is lifted out of the function `f`, the function `g` and the class `E` are first lifted out of the method `m`. This turns them into a method and an inner class of `D`. During this operation, the reference to the variable `x` in the body of the function `g` is treated as if it was a reference to a global variable and is left unmodified. The result of this first lifting is given below.

```
def f(x: Int): Any = {
  class D() {
    def g(y$g: Int): Int = x + y$g;
    class E(_y$E: Int) {
      val y$E: Int = _y$E;
      def n(): Int = D.this.g(E.this.y$E);
    }
    def m(y: Int): Any = new E(y);
  }
  new D()
}
```

At this point, the class `D` can be normally lifted out of the function `f`. This leads to the following code.

```
class D(_x$D: Int) {
  val x$D: Int = _x$D;
  def g(y$g: Int): Int = D.this.x$D + y$g;
  class E(_y$E: Int) {
    val y$E: Int = _y$E;
    def n(): Int = D.this.g(E.this.y$E);
  }
  def m(y: Int): Any = new E(y);
}
def f(x: Int): Any = new D(x);
```

It is of course possible to perform the two steps described here in a single one. In that case, it is important to recognize that the initial extra set to which a reference to a free variable contributes depends not only on the location of the reference but also on the place where the referenced variable is defined. If the reference is separated from the definition by at least one enclosing class then it contributes to the initial extra set of

the outermost of those enclosing classes. Otherwise, it contributes to the initial extra set of the function in which it occurs.

In our example, the variables `x` and `y` are both referenced in the body of the function `g` but only `y` contributes to the initial extra set of `g`. The reference to the variable `x` contributes to the initial extra set of the class `D`, which encloses the reference and separates it from the definition of `x` in the function `f`.

3.3.5 Generalized Algorithm

When classes are taken into account, the lambda lifting algorithm stays the same except that extra sets are also computed for local classes and local classes are augmented with an extra field for each variable in their extra set. Like local functions, local classes are attributed a call set and the extra sets of local classes are computed the same way as the extra sets of local functions.

Constructors of local classes are treated like local functions defined in the same scope as their class. While computing extra sets, it is important to remember that a constructor is implicitly called in each instance creation. Furthermore, the extra set of each primary constructor has to be augmented with its class. Indeed, in the step "parameter and argument addition", the bodies of primary constructors have to be augmented with code to initialize the extra fields of their class.

Methods and inner classes are not treated like local functions and local classes but are considered as a part of their enclosing class. They remain in place and contribute to the free variables and the call set of their enclosing class. So the free variables of a local class are all the free variables that occur in its methods and in methods of its inner classes. Similarly, the call set of a local class is the union of the call sets of all its methods and the methods of its inner classes.

The presence of classes does not change the fact that strongly connected definitions defined in the same scope have identical extra sets. It is therefore still possible to compute all the extra sets with a single graph traversal.

3.4 Mutable Variables

In Scala, unlike in Java, references to free variables in local definitions are not restricted to be immutable. This is problematic because in lifted definitions references to free variables are replaced with references to copies of the originally referenced variables. Thus, after lambda lifting there are

several instances of the same variables and modifications to one instance are not reflected by the other ones.

In the example below, the function `loop1` updates the variable `sum`. If it was lifted to the top-level, it would only have access to a copy of the original variable. Modifications to that copy would have no effect on the original variable and the function `loop2` would always be called with `sum` equal to `0.0`.

```
def normalize(list: List[Float]): List[Float] = {
  var sum: Float = 0.0;
  def loop1(ls: List[Float]): Unit =
    if (!ls.isEmpty) {sum = sum + ls.head; loop1(ls.tail)}
  def loop2(ls: List[Float]): List[Float] =
    if (ls.isEmpty) Nil else (ls.head / sum)::loop2(ls.tail);
  loop1(list); loop2(list)
}
```

Interestingly, mutable fields pose no problem because they are always accessed through a reference to their enclosing object. It may happen that this reference gets duplicated but never the field itself. This provides a solution for the mutable variables that are modified in local definitions. These variables are replaced with mutable fields `elem` of instances of the class `Ref` defined below.

```
class Ref[Elem](_elem: Elem) { var elem: Elem = _elem; }
```

More precisely, before lambda lifting, all variables v of type X initialized with x that are referenced in at least one local definition are replaced with immutable variables v of type `Ref[`X`]` and initialized with **new** `Ref[`X`](`x`)`. Furthermore all references to v are replaced with references to v`.elem`. After that, local definitions can be lambda lifted as usual.

```
def normalize(list: List[Float]): List[Float] = {
  val sum: Ref[Float] = new Ref[Float](0.0);
  def loop1(ls: List[Float]): Unit =
    if (!ls.isEmpty)
      {sum.elem = sum.elem + ls.head; loop1(ls.tail)};
  def loop2(ls: List[Float]): List[Float] =
    if (ls.isEmpty) Nil else (ls.head / sum.elem)::loop2(ls.tail);
  loop1(list); loop2(list)
}
```

3.5 Typing Issues

Until now we considered only value variables but functions and classes can have type parameters that can be referenced by any nested definition. Local definitions may therefore contain references to free type variables. These are eliminated just like references to free value variables by taking them into account while computing the extra sets of the local definitions. These are then augmented with an extra type parameter for each type variable in their extra set. And, function calls and class types are accordingly augmented with extra type arguments.

Because types can contain references to free type variables, they must be taken into account in the computation of the extra sets. Singleton types and member types contain both a stable path and can therefore reference free value variables. These references must also be taken into account in the computation of the extra sets. Indeed, it may happen that the only reference to a free value variable in a local definition, like the reference to `x` in the class `D` below, occurs in a type of that definition. In that case, the variable still must be added to the extra set of the definition because otherwise the type in which it occurs could not be correctly expressed in the lifted definition.

```
abstract class C() { type T; def t(): T; }
def f(x: C): String = {
  class D() { def m(t: x.T): String = t.toString(); }
  new D().m(x.t())
}
```

The lifted class `D` is given below. The type of the parameter `t` clearly relies on the added field and could not be correctly expressed without it.

```
class D(_x$D: C) {
  val x$D: C = _x$D;
  def m(t: this.x$D.T): String = t.toString();
}
```

It is worth flagging the extra parameters and fields that are added only because they occur in some type. Indeed, when types are erased in the type erasure phase of the compiler, all singleton types and all member types are eliminated. Thus, after type erasure, the flagged extra parameters and fields are no longer referenced and can safely be removed.

The fact that some value parameters or fields are added only because of some types is annoying. Even more annoying is the fact that there are several situations where the lifted type is invalid even with the extra parameters and fields. The different kinds of problems are illustrated by the

local definitions of the following example.

```
abstract class C() { type T; }
abstract class D[X]();
def f(x: C): Any = {
  def g[T <: x.T](t: x.T): x.T = t;
  class E() extends D[x.T]();
  new E()
}
```

The lifted version of the function g is given below. It shows that the bound of T, the type of t and the return type all three reference the argument x$g. All this is illegal because function arguments can only be referenced from within the function body. However, lifting this restriction seems not completely unreasonable and the fact that valid Scala code produces such functions gives credit to this. For the function g, it would imply that in all its calls, the first value argument has to be a stable path p of type C. Its second value argument would have to be of type p.T. The type argument would have to be a subtype of that type. The result of the call would be of type p.T.

```
def g[T <: x$g.T](x$g: C, t: x$g.T): x$g.T = t;
```

It might seem that the lifted version of the class E is the following one, but this is not correct.

```
class E(x$E: C) extends D[x$E.T](x$E);
```

Indeed, we know that the value parameters of a class correspond to the value parameters of its primary constructor. Referencing them in the supertype of the class just makes no sense. The correct lifting of the class E produces the following definition:

```
class E(_x$E: C) extends D[E.this.x$E.T](_x$E) {
  val x$E: C = _x$E;
}
```

In this definition the supertype of the class E depends on its current instance. This is illegal in Scala, because the supertype of a class is not considered as a type that occurs within the class. It may therefore not reference the current instance of the class. However, here again, it is not completely unreasonable to change this. It would imply that the base type of class D of a type T (*base-type*(T, D)) can only be computed if T is a singleton type p`.type`. This base type would be D[p`.x$E.T`], which is a perfectly reasonable type.

It is interesting to note that if the class `D` had a virtual type `X` instead of its type parameter `X`, there would be no problem at all. The lifted definition would be the following one.

```
class E(_x$E: C) extends D(_x$E) {
  val x$E: C = _x$E;
  type X = E.this.x$E.T;
}
```

Note that the type assigned to the virtual type `X` is exactly the same as the one passed to the type parameter `X` in the previous definition.

3.6 Alternative Method

When several local definitions are defined in the same block, it may be profitable to introduce a local class in the block and move all the local definitions into the new class. This transforms the local definitions into fields, methods and inner classes of the new class. Here is the result of this operation for the example of Section 3.4.

```
def normalize(list: List[Float]): List[Float] = {
  class L() {
    var sum: Float = 0.0;
    def loop1(ls: List[Float]): Unit =
      if (!ls.isEmpty) {sum = sum + ls.head; loop1(ls.tail)};
    def loop2(ls: List[Float]): List[Float] =
      if (ls.isEmpty) Nil else (ls.head / sum) :: loop2(ls.tail);
  }
  val l: L = new L();
  l.loop1(list); l.loop2(list)
}
```

The new class `L` contains no free variables and can be moved without modification to the top-level. This produces better code than what would have been obtained without the class `L`. Both versions use an instance of an auxiliary class; the new instance of `L` replaces here the new instance of `Ref`. But, in this version, the functions `loop1` and `loop2` get no extra parameter whereas in the other version both get an extra parameter for the variable `sum`. One could argue that in this version the two functions get an extra implicit parameter containing the current instance of `L`. That is true, but in the other version, the functions would also end up in some class and thus have an implicit parameter containing the current instance of that class.

This example shows that the addition of extra parameters to lifted functions (and also extra fields to lifted classes) can be avoided by introducing an auxiliary class. The extra parameters are then replaced with fields of the auxiliary class. This has a little cost for immutable fields, which must be dereferenced whereas with the extra parameters they are directly accessible. The cost for mutable variables remains the same as they would anyway be dereferenced from an instance of `Ref`. Furthermore, if there are several mutable variables, multiple instances of `Ref` are replaced with a single instance of the auxiliary class.

There are several factors that are in favor of the usage of an auxiliary class. If the local functions are recursive, the cost of the instance creation of the auxiliary class is spread over all the recursive calls. If there are many free variables, the auxiliary class can significantly reduce the stack usage. If the free variables are accessed rarely, for example only in leaf calls, then the additional cost of dereferencing them is largely compensated by the avoided cost of passing the extra arguments in each function call.

The usage of an auxiliary class can also help when there are some typing issues with the lifting of the local definitions. The code below shows the last example of the previous section where the two problematic local definitions have been moved into an auxiliary class `L`.

```
abstract class C() { type T; }
abstract class D[X]();
def f(x: C): Any = {
  class L() {
    def g[T <: x.T](t: x.T): x.T = t;
    class E() extends D[x.T]();
  }
  val l: L = new L();
  new l.E()
}
```

The class below is the result of the lifting of the class `L`. The method `g` and the class `E` no longer present a typing issue.

```
class L(_x$L: C) {
  val x$L: C = _x$L;
  def g[T <: L.this.x$L.T](t: L.this.x$L.T): L.this.x$L.T = t;
  class E() extends D[L.this.x$L.T]();
}
```

This example shows that the usage of an auxiliary class can indeed avoid certain typing issues. However, introducing an auxiliary class and thus some real runtime costs just to avoid typing issues is somewhat dubi-

ous. Unlike the extra parameters introduced only because of some singleton type or member type, these auxiliary classes cannot be easily removed during type erasure.

Although systematically introducing an auxiliary class just to avoid typing issues would be foolish, there are cases where the usage of an auxiliary class would translate into faster code. However determining when this would be the case is not an easy task and requires some good heuristic. The Scala compiler does currently never introduce auxiliary classes.

Chapter 4

Explicit Outer

Although Java supports inner classes since version 1.1, the Java virtual machine still supports only top-level classes. This chapter describes how inner classes are eliminated by augmenting them with an explicit outer field and lifting them to the top-level.

4.1 Introduction

After lambda lifting all classes are either top-level classes or inner classes. As the Java virtual machine provides no support for inner classes, it is necessary to lift all of them to the top-level. One could try to somehow lambda lift inner classes to the top-level. There are however several differences between the lambda lifting of local classes and the lifting of inner classes to the top-level.

First of all, an inner class can have several enclosing classes and it may reference the current instance of each of them. These references to enclosing instances are like references to free variables. So, lambda lifting an inner class would add as many extra fields as there are referenced enclosing instances. However, this is unnecessary and may waste a lot of memory. Indeed, adding to each inner class a single explicit outer field containing the instance of its directly enclosing class is enough. Indirect enclosing instances can still be reached through a chain of outer field selections.

Like lambda lifted local classes, lifted inner classes need an extra type parameter for each free type variable X they reference. For example, if the class `D`, defined below, is lifted to the top-level, it needs an extra type parameter for the type parameter `T`.

```
class C[T]() { class D() { def f(): T = f(); } }
val c: C[String] = new C[String]();
```

```
val d: c.D = new D();
```

Class types of lifted classes must of course provide corresponding extra type arguments. In our example, the type of `d` must be augmented with an extra type argument for the extra type parameter of the class `D`. For local classes, the extra argument for an extra parameter corresponding to a free variable X is simply X. This is different for inner classes. Indeed, unlike class types of local classes, class types of inner classes can occur anywhere. They are not restricted to the scope or nested scopes of the one where the class is defined. It may therefore happen that X is not even a valid type in the scope where the class type is. In our example, it would clearly be illegal to augment the type of `d` with the type argument `T`. For class types of inner classes, the extra type arguments have to be computed from their class qualifier. The adequate extra argument for the type of `d` is `String` because the extra argument corresponds to the type parameter `T` of the enclosing class of `D` and this parameter is equal to `String` in the type of the prefix `c` of `D` in the type of `d`.

When a class is lambda lifted, the number of extra fields and also the number of extra parameters of its constructors depend on the number of free variables referenced in its body. An implementation change that removes or adds references to free variables can change these numbers. This is problematic for inner classes because an inner class can be referenced from anywhere and not only from the same scope or from a nested scope of the one where it is defined. For example, it may happen that an inner class is defined in a file and extended or instantiated in another one. With separate compilation, these two files can be compiled at different times. In principle, only interface changes should force the recompilation of dependent files, but if an implementation change modifies the number of extra parameters of the constructor of the lifted inner class, it will force the recompilation of all files that extend or instantiate this class. To avoid this, inner classes are systematically augmented with an explicit outer field and their constructors with an extra parameter to initialize it even if they never reference any of their enclosing instances.

Finally, when an inner class is lifted to the top-level, it loses the privileges it enjoyed as a member of its enclosing class. For example, within an inner class, any private member of any of its enclosing classes can be referenced. When the inner class is moved to the top-level, it loses this privilege and the references are no longer valid. This is resolved by adding access methods to the enclosing classes and by replacing the references with calls to these methods.

Section 4.2 describes the base algorithm used to lift inner classes to the

top-level. Section 4.3 describes how the the problems posed by the loss of privileges are solved. Section 4.4 describes how qualifiers of inner class types are replaced with additional type arguments. Section 4.5 discusses some typing issues that remain and Section 4.6 some specific issues related to explicit self types.

4.2 Base Algorithm

Conceptually, inner classes are lifted to the top-level in three steps but these can easily be performed all at once in an implementation. During the first step, all inner classes are augmented with an explicit outer field and an extra type parameter for each type parameter of all their enclosing classes. And, all constructors of inner classes get an extra value parameter, which is used to initialize the explicit outer field. At the same time, class types and constructor calls of inner classes are augmented with corresponding arguments.

The second steps affects only the body of inner classes. It replaces all references to type parameters of enclosing classes with references to the corresponding extra type parameter of the inner class. Furthermore, references to the directly enclosing instance are replaced with references to the explicit outer field and references to indirectly enclosing instances are replaced with chains of outer field selections.

During the third step, all inner classes are moved to the top-level. At the same time, the qualifiers of class types and the prefixes of constructor calls are dropped.

Let us first illustrate the lifting of inner classes with the three following classes.

```
class C0[X0]() {
  class C1[X1]() {
    class C2[X2]() {
      def print(): Unit = {
        System.out.println(C2.this);
        System.out.println(C1.this);
        System.out.println(C0.this);
}}}}
```

The lifted definitions are given below. Observe that the two lifted classes `C1` and `C2` get both a single explicit outer field but an extra type parameter for each type parameter of all their enclosing classes. Observe also how the references to enclosing classes have been replaced with selections of outer fields on the current instance.

```
class C0[X0]() {}
class C1[X0$C0,X1](_outer$C1: C0[X0$C0]) {
  val outer$C1: C0[X0$C0] = _outer$C1;
}
class C2[X0$C0,X1$C1,X2](_outer$C2: C1[X0$C0,X1$C1]) {
  val outer$C2: C1[X0$C0,X1$C1] = _outer$C2;
  def print(): Unit = {
    System.out.println(C2.this);
    System.out.println(C2.this.outer$C2);
    System.out.println(C2.this.outer$C2.outer$C1);
  }
}
```

Let us now illustrate with the code below how class types, class extensions and class instantiations of inner classes are affected.

```
class D0[Y0]() extends C0[List[Y0]]();
val d0: D0[Int] = new D0[Int]();
class D1[Y1]() extends d0.C1[List[Y1]]();
val d1: D1[Long] = new D1[Long]();
val c2: d1.C2[Float] = new d1.C2[Float]();
```

The same code after inner class lifting is given below. Observe that the prefix of the constructor call in the extends clause of `D1` and in the instance creation of `C2` and the qualifier of the `C2` class type have all been dropped.

```
class D0[Y0]() extends C0[List[Y0]]();
val d0: D0[Int] = new D0[Int]();
class D1[Y1]() extends C1[List[Int],List[Y1]](d0);
val d1: D1[Long] = new D1[Long]();
val c2: C2[List[Int], List[Long], Float] =
  new C2[List[Int], List[Long], Float](d1);
```

The extra value argument of the inner class constructor calls are simply the dropped constructor prefixes. The computation of the extra type arguments of inner classes is less trivial. The extra type arguments of superclasses and instance creations are computed from the constructor prefix and those of class types from the class qualifier. The extra arguments are the arguments of the enclosing classes implied by the constructor prefix or the class qualifier. For example, the extra argument of the superclass of `D1` is obtained by computing *base-type*(`d0.type`, `C0`), which evaluates to `C0[List[Int]]`. Thus, the extra type argument of `C1` is `List[Int]`.

The extra type arguments of the class type and the instance creation of `C2` are obtained by computing *base-type*(`d1.type`, `C1`). Note that this computation is performed with not yet lifted definitions. It evaluates to

`d0.C1[List[Long]]`. Thus, the argument corresponding to the parameter of the class `C1` is `List[Long]`. For the argument corresponding to the parameter of `C0` an additional computation of *base-type*(`d0.`**`type`**, `C0`) is required. More generally, the computation of the extra arguments for an inner class with *n* enclosing classes requires *n* successive *base-type* computations.

4.3 Lost Privileges

Inner classes enjoy the same privileges when they access members of their enclosing classes as these enclosing classes themselves. There are three of these privileges. First of all, inner classes can access private member of their enclosing classes. They can also access protected members defined in a different package. Finally, they can call method implementations inherited and possibly overridden by their enclosing classes (calls like `C.`**`super`**`.f(...)`). All these privileges are lost when the inner classes are lifted to the top-level.

In the code below, the method `i` of the inner class `I` uses the three privileges. First, it calls the implementation of the method `f` inherited and overridden by its enclosing class `B` . Then, it calls the private method `h` of `B`. Finally, it calls the protected method `g` inherited by `B`.

```
package foo;
class A() {
  def f(x: Int, y: Int): Int = x + y;
  protected def g(): Int = 0;
}
package bar;
class B() extends foo.A() {
  override def f(x: Int, y: Int): Int = x - y;
  private def h(): Int = 1;
  class I() { def i(): Int = B.super.f(h(), g()); }
}
```

The definition of the lifted class `I` is given below. The call to `h` is clearly illegal as `h` is a private method of `outer$I`. The same is true for `g`, which is a protected method defined in a different package. Things are even worse for the super-call to `f`. There is not even a syntax to express it. The syntax `f@A` is not legal Scala. It is used here to indicate that the lookup for the implementation of `f` should start in the class `A`.

```
class I(_outer$I: B) {
```

```
  val outer$I: B = _outer$I;
  def i(): Int = outer$I.f@A(outer$I.h(), outer$I.g());
}
```

One way to eliminate these illegal references would be to simply declare them legal. The analyzer would check that the source code contains no illegal reference but later phases would be allowed to generate such references. In fact, that is exactly what is done for virtual types. Unfortunately, the same is not possible for methods because the Java virtual machine verifies at runtime that no private or protected methods are illegally called. Furthermore, the Java virtual machine supports only method super-calls on the current instance. For these reasons, calls to private and protected methods on enclosing instances and also method super-calls on enclosing instances are replaced in lifted classes with calls to access methods added to the corresponding enclosing classes. The code below shows the definitions of `B` and `I` resulting from the lifting of `I`.

```
class B() extends foo.A() {
  override def f(x: Int, y: Int): Int = x - y;
  private def h(): Int = 1;
  def access$B$super$f(x: Int, y: Int): Int = super.f(x, y);
  def access$B$g(): Int = g();
  def access$B$h(): Int = h();
}
class I(_outer$I: B) {
  val outer$I: B = _outer$I;
  def i(): Int = outer$I.access$B$super$f(
    outer$I.access$B$h(), outer$I.access$B$g());
}
```

Note that field references are never problematic because fields are always accessed through their getter and setter methods. It is therefore never necessary to generate access methods for fields.

4.4 Lost Qualifiers

When inner classes are lifted to the top-level, the qualifiers of their class types are dropped. The qualifiers are not entirely lost as the extra type arguments of class types are computed from their qualifier but the exact type and identity of the enclosing instances are lost. This is sometimes problematic. For example, let us consider the following code.

```
class O() {
```

```
  type T;
  class I() { def f(): O.this.T = this.f(); }
  def g(i: this.I): this.T = i.f();
}
```

The type of `i.f()` is `this.T` (obtained by replacing `O.this` with `this` in `O.this.T`). It corresponds to the declared type of `g`. The expression `i.f()` is therefore a legal body of `g`. Consider now the definitions of `I` and `O` resulting from inner class lifting.

```
class I(_outer$I: O) {
  val outer$I: O = _outer$I;
  def f(): this.outer$I.T;
}
class O() {
  type T;
  def g(i: I): this.T = i.f();
}
```

The type of `i.f()` is here `i.outer$I.T` (`this.outer$I.T` with `this` replaced with `i`). It would be a subtype of `this.T`, the declared type of `g`, if it could be established that `i.outer$I` is equal to `this`. Unfortunately, the declared type `O` of `outer$I` does not let us establish that. The expression `i.f()` is therefore no longer a legal body of `g`.

The problem comes from the fact that class qualifiers are dropped during the class lifting. Before lifting, the parameter `i` is declared as an instance of `I` whose enclosing instance is of type `this.type`. After lifting, it is still an instance of `I` but its enclosing instance is just some instance of `O`.

The class qualifiers should be conserved by the class lifting. This is done by adding to inner classes an extra type parameter corresponding to the type of their enclosing class. This produces the following definitions for our example.

```
class I[O$this <: O](_outer$I: O$this) {
  val outer$I: O$this = _outer$I;
  def f(): this.outer$I.T;
}
class O() {
  type T;
  def g(i: I[this.type]): this.T = i.f();
}
```

The type of `i.outer$I` is now `this.type`. The path `i.outer$I` is therefore equal to `this` and `i.f()` is again a legal body of `g`.

Inner classes with multiple enclosing classes get an extra type parameter for each of their enclosing classes. Indeed, the parameters corresponding to the indirectly enclosing classes are needed to express the bound of the parameter corresponding to the directly enclosing class, as illustrated in the following example.

```
class A() { class B() { class C(); } }
```

The lifted versions of the above classes are given below. Observe that in the lifted version of `C`, the type parameter `A$this` is needed to express the upper bound of its second type parameter `B$this`.

```
class A() {}
class B[A$this <: A](_outer$B: A$this) {
  val outer$B: A$this = _outer$B;
}
class C[A$this <: A, B$this <: B[A$this]](_outer$C: B$this) {
  val outer$C: B$this = _outer$C;
}
```

4.5 Typing Issues

Unfortunately the additional type parameters do not solve all problems. Indeed, the lifting of classes whose parameter bounds or types reference an enclosing instance produces illegal types. Let us illustrate this with the following code where the type `T`, a shorthand for `P.this.T`, of the parameter `_t` of the class `Q` references the enclosing instance of `Q`.

```
abstract class P() { type T; class Q(_t: T) { val t: T = _t; } }
```

The definitions resulting from the class lifting are given below. The type of the parameter `_t` now illegally references the current instance of `Q`.

```
abstract class P() { type T; }
class Q[P$this <: P](_outer$Q: P$this, _t: this.outer$Q.T) {
  val outer$Q: P$this = _outer$Q;
  val t: this.outer$Q.T = _t;
}
```

The reference to the current instance could be avoided by replacing the type of the parameter `_t` with `_outer$Q.T`. This is however even more problematic because class value parameters are accessible only in the body of the class and not in the bounds and the types of its parameters. Furthermore, the value `_t` would no longer be a legal initializer of the field `t`. Indeed, the type `_outer$Q.T` is not a subtype of `this.outer$Q.T` because

there are no typing rules that can be applied to establish that the paths `_outer$Q` and **`this`**`.outer$Q` are equal.

Another way to avoid the reference to the current instance is to replace the type of the parameter `_t` with `P$this.T`. The reference to the type parameter `P$this` is legal but not the selection of the virtual type `T` because virtual types must always be selected on stable values. However, if the type parameter `P$this` is marked as being of a special kind that can only be instantiated with a singleton type, the selection can be allowed. The type `P$this.T` becomes then a legal one and the value `_t` remains a legal initializer of the field `t`. Indeed, as `P$this` represents a singleton type and `outer$Q` is of that type, the paths `P$this` and **`this`**`.outer$Q` necessarily denote the same value and are therefore equal. Thus, the type `P$this.T` is a subtype of **`this`**`.outer$Q.T` and `_t` a legal initializer of `t`. So, with this solution, the lifted definition of the class `Q` is well-formed. Unfortunately, this is not true for its class types. Indeed, there are class types for which there is not singleton type for the parameter `P$this`. For example, the type `P#Q` becomes after class lifting `Q[P]` where `P` is not a singleton type.

The best solution to solve the problem posed by the reference to the current instance seems to be to simply allow references to the current instance in the bound and the types of class parameters as was already argued in Section 2.10.3.

The lifting of inner classes can also generate class definitions whose supertype references the current instance, which is illegal. This happens when an inner class references an enclosing instance in its supertype. The class `N` defined below is such a class. Its fully desugared supertype is `L.`**`this`**`.M[L.`**`this`**`.T]`.

```
abstract class L() {
  type T;
  class M[X]();
  class N() extends M[T]();
}
```

The result of class lifting is given below. The first type argument in the supertype of the class `N` could possibly be replace with `L$this` but that is not the case for the path **`this`**`.outer$N` in its second type argument.

```
abstract class L() { type T; }
class M[L$this <: L, X](_outer$M: L$this) {
  val outer$M: L$this = _outer$M;
}
class N[L$this <: L](_outer$N: L$this)
  extends M[this.outer$N.type, this.outer$N.T](_outer$N)
```

```
{
  val outer$N: L$this = _outer$N;
}
```

Here again, it seems that the best solution is to simply allow references to the current instance in class supertypes as was already argued in Section 3.5.

4.6 Explicit Self Types

The explicit self type of a class applies only to the current instance of that class. This is problematic because class lifting replaces references to the current instance of enclosing classes by references to outer fields. Let us illustrate the problem with the definitions below. Note that the call of the method `g` in the body of the method `f` is legal only because the class `O` has the explicit self type `P`.

```
class O(): P { class I() { def f(): Int = O.this.g(); } }
class P() extends O() { def g(): Int = 0; }
val o: O = new P();
val i: o.I = new o.I();
```

Class lifting produces the code below where the call of the method `g` is no longer legal. Indeed, the field `outer$I` is of type `O$this`, which is upper bounded by `O` but `O` is not a subtype of `P` as the explicit self type `P` of the class `O` only applies to its current instance.

```
class O(): P {}
class P() extends O() { def g(): Int = 0; }
class I[O$this <: O](_outer$I: O$this) {
  val outer$I: O$this = _outer$I;
  def f(): Int = this.outer$I.g();
}
val o: O = new P();
val i: I[o.type] = new I[o.type](o);
```

Replacing the upper bound of the parameter `O$this` by the explicit self type of the class `O` is not a solution. It would make the call to the method `g` legal. However, at the same time, the type `I[o.type]` of the variable `i` would become illegal because the type `O` of the value `o` is not a subtype of `P`.

Changing the type system to consider the explicit self type of a class as a supertype would make the call to the method `g` legal but such a change raises many other problems because it introduces cycles in the subtyping

relationship. In our example, the explicit self type `P` of the class `O` would become a supertype of `O` but `O` is already a supertype of `P` as the class `P` is a subclass of `O`.

One solution is to use access methods like for private and protected members (Section 4.3). However, unlike for private and protected members, these methods do not perform something that could not be done elsewhere. Indeed, each of these methods would end up, after the type erasure phase, simply casting the current instance to the explicit self type and calling the accessed method. For example, the access method for the method `g` in the class `O` would simply cast the current instance of `O` to `P` and call the method `g`. This cast could also be done elsewhere; in the class `I`, the outer field `outer$I` could be cast to `P` and thus the method `g` could be called directly. In this case, access methods introduce unneeded runtime overhead for the sole benefit of delaying the insertion of type casts. So it seems more advantageous to directly insert these casts and avoid the overhead of the access methods.

Another solution that avoids early introductions of casts and introduces no runtime overhead exists but it requires some small changes to the analyzer, the type system and the back-end. It assumes that all classes inherit two special members: a virtual type `Self` and an implicit field `self`. These two members are used to encode explicit self types. The upper bound of the virtual type `Self` is refined in every class to the type of the current instance. This type is equal to the explicit self type if the class has one. The field `self` is an immutable field of type `Self` initialized with the current instance. Thus, for any expression *e*, the expression *e*`.self` returns the same value as *e* and is of type *e*`.Self`. The analyzer is modified to add to each class definition the adequate refinement of its virtual type `Self`, to remove all explicit self types and to replace all current instances of the form *C*`.this` that occur in a context that relies on the explicit self type of the class *C* by the member selection *C*`.this.self`. Thus, the analyzer would transform the definition above of the class `O` to the following one:

```
class O() {
  type Self <: P;
  class I() {
    type Self <: I;
    def f(): Int = O.this.self.g();
  }
}
```

This definition contains no explicit self type and lifting the class `I` to the top-level poses no problem. The type system needs to be modified such

that it understands that for any stable path p, the paths p and p`.self` are equal. Finally, the back-end needs to be modified such that it compiles any expression e`.self` like it would compile the expression e with possibly an additional cast if needed.

Chapter 5

The Core Language

Transformations like lambda lift and explicit outer cannot be expressed in a fully type safe way as Scala to Scala transformations because the transformation of some programs yields ill-typed Scala code. The Core language is a typed intermediate language for Scala compilers. It generalizes some aspects of Scala in such a way that all transformations needed to compile Scala code can be expressed as Core to Core transformations. This chapter describes the syntax and the type system of the Core language, the encodings used to translate Scala code into Core code and a Core version of the lambda lift and the explicit outer transformations.

5.1 Introduction

The Core language is a typed intermediate language that can be used to express all code transformations needed to compile Scala code as well-typed Core to Core transformations. The design of the Core language was driven by two rather contradictory constraints: the language had to be both simple, in order to simplify the implementation of the code transformations, and expressive enough to encode Scala programs.

There are several ways in which the Core language is simpler than Scala. Some simplifications are possible because the Core language is only intended as an intermediate language and not as a programming language. Compared to Scala, the Core language consists of a very limited number of constructs. For example, the Core language has no notion of type parameters; those are encoded with virtual types. This tends to increase the size of programs, as virtual types are much more verbose than type parameters. This encoding would be very cumbersome in a programming language but it poses no problem at all in an intermediate language

because the encoding is done by the compiler. On the contrary, it simplifies all parts of the compiler that handle intermediate code because all those parts do not have to handle type parameters, they only need to handle virtual types.

Many simplifications are also possible because the Core language is only intended to be used after the program analysis. For example, the program analysis tries to infer all type arguments that were omitted in function calls. It is therefore possible to make type arguments of function calls mandatory in the Core language. Indeed, if some type arguments of a function call are missing in the Scala source code, either the analyzer succeeds in inferring them and can thus create a corresponding Core function call with no missing type arguments, or it fails and reports a compilation error. In that case, the compilation stops after the program analysis and no Core function call needs to be created.

In Scala, the names of entities, like classes, functions or variables, have a unique role: they are a means of designating the entity of which they are the name. Different entities may have the same name if they are defined in different contexts. A same name may therefore designate different entities in different contexts. For example, two packages `foo` and `bar` can both contain a class named `C`. In that case, the class name `C` is ambiguous; it can designate both classes. The meaning of the class name `C` depends on the context in which it occurs, namely the current package and the list of imported class names. The determination of the entity designated by each name is the task of the name analysis, which is a part of the program analysis. During the program analysis, the compiler creates a *symbol* for each entity defined in the program. The task of the name analysis is to replace each name by a reference to the symbol of the entity it designates. It is a compilation error if the name analysis cannot determine a unique entity for each name. Thus, after the program analysis, either the name analysis succeeded and each name could be resolved to a unique entity and replaced by a reference to the corresponding symbol or it failed and the compilation stopped with one or more error messages. This implies that in programs that pass the program analysis entity names are no longer relevant; each one has been replaced by a reference to the symbol of the unique entity it designates. In the Core language, identifiers represent the symbol of the different entities rather than their name. It is therefore assumed that all entities have a globally unique identifier. The usage of symbols instead of names has the advantage that any entity, in any context, can be designated unambiguously by its symbol. This leads to much simpler typing rules because they do not have to incorporate name resolution aspects.

The definition of a new member, the refinement of an inherited member and the implementation or overriding of an inherited member are conceptually three different notions but Scala uses the same syntax for all three. Let us illustrate this with the code below.

```
abstract class A           { type T <: Any;    def f(): Int; }
abstract class B extends A { type T <: Number; def f(): Int = 0; }
```

The class `A` defines a new type `T` and a new method `f`. Although it uses the same syntax, the class `B` does neither define a new type `T` nor a new method `f`. Instead, it refines the upper bound of the type `T` inherited from `A` and implements the method `f` also inherited from `A`. Thus, `T` and `f` are bindings in `B` and fresh binders in `A`. In Scala, the exact meaning of a member declaration can only be determined during the program analysis by examining the inherited members of the class in which it occurs. The Core language avoids these difficulties by using three different constructs for the three different notions.

Other difficulties are avoided with additional annotations. For example, the type of a conditional expression is the least upper bound of the types of its two branches but the least upper bound of two types does not always exist (Section 2.9.1). Conditional expressions of the Core language are therefore annotated with their type. For similar reasons, block expressions are also annotated with their type. Indeed, the type of a block is determined by the type of its last expression. This type may contain references to local members of the block. These references must be eliminated because the type of the block must be valid in the scope enclosing the block where the block members are not available. This is done by *widening* the type of the last expression of the block, which consists in computing the least upper bound of the type that does not reference any members of the block. This least upper bound, like the least upper bound of two types, does not always exist. Block expressions are therefore annotated with their type in order to avoid its computation.

Finally, the Core language is also simpler than Scala because it is less constrained than Scala. The well-formedness of a Core program guarantees more or less only that all operations performed by the program are legal; it guarantees that members are selected on values that inherit them, that arguments in function calls are of the right type and number, that variables are assigned with values of the right type, etc. The well-formedness of a Scala program guarantees much more; it guarantees that variables are initialized before they are read, that new instances are initialized by a constructor before they are used, that only non-abstract classes are instantiated, that private and protected members are never accessed from scopes

that are not entitled to do so, etc. All these constraints are not enforced by the Core language. In fact, the Core language does not even have a notion of uninitialized variables, constructors, abstract members and private or protected members.

There are also some aspects on which the Core language is more general or more expressive than Scala. For example, the Core language has a notion of record, which unifies some aspects of the global scope, classes, functions and blocks. There are also the value parameters of functions, which can be referenced by the bounds and the types of the parameters and by the return type, while in Scala only type parameters can be referenced.

5.2 Syntax

The Core language relies on six different kinds of *entities*: a unique root context, classes, functions, blocks, value variables and type variables. The syntax of the language is given in Figure 5.1. It consists of symbols, definitions and expressions; symbols are a means to identify and reference the different entities, definitions define the entities by specifying their symbol and attributes and expressions describe the types and the values of the language.

Overlined metavariables denote lists of metavariables. For example, $\overline{C}$ is a shorthand for $C_1, \ldots, C_n$ with $n \geq 0$. The length of a list is obtained with the syntax $|\overline{C}|$ and empty lists are denoted by ϵ. Note that C and $\overline{C}$ can be used simultaneously and denote $n + 1$ different metavariables.

5.2.1 Symbols

All entities, including blocks, are identified by a globally unique symbol. The identifying symbol of an entity is specified in the definition of the entity. The root context, which has no explicit definition, is identified by the predefined symbol **`Root`**. Thanks to these globally unique symbols, any entity can be referenced by its symbol in any expression in an unambiguous way and independently from the context in which the expression occurs.

A *record* is an entity consisting of a list of members. It can be instantiated into values through which its members can be accessed. Every member definition has access to an instance of the record, known as the *current instance* of the record, and has thus access to all other members of its record. There are four different kinds of records: the root context,

Symbols				
Record	P, Q, R	$=$	**Root** $\mid C \mid f \mid b$	
Class	A, B, C	$=$	$\ldots$	
Function	f, g	$=$	$\ldots$	
Block	b	$=$	$\ldots$	
Value	u, v, w	$=$	**outer@**$R \mid \ldots$	
Type	U, V, W	$=$	**Outer@**$R \mid \ldots$	
Member	m, n, o	$=$	$f \mid v \mid V$	
Definitions				
Program	$\mathcal{P}$	$=$	$\overline{\mathcal{C}}; \overline{\mathcal{F}}; \overline{\mathcal{V}}$	
Class	$\mathcal{C}$	$=$	**class** C **extends** $\overline{C}\,\{\overline{\mathcal{C}}; \overline{\mathcal{F}}; \overline{\mathcal{V}}; \overline{\mathcal{T}}; \overline{O}\}$	
Function	$\mathcal{F}$	$=$	**def** $f\{\overline{f}\}[\overline{\mathcal{T}}](\overline{\mathcal{V}})$**:** $X = x$	
Value	$\mathcal{V}$	$=$	$[$**mutable**$]$ **val** v**:** X	
Type	$\mathcal{T}$	$=$	**type** V **>:** X **<:** Y	
Expressions				
Value	$x, y, z,$	$=$	**this**(k)	enclosing context
	p, q, r	$\mid$	p**.**v	variable evaluation
		$\mid$	p**.**$v = x$	variable assignment
		$\mid$	p**.**$f[\overline{X}](\overline{p})$	function application
		$\mid$	**new** X	instance creation
		$\mid$	**if** $[X](x)$ y **else** z	conditional
		$\mid$	$\{b$**:** $X \mid \overline{\mathcal{C}}; \overline{\mathcal{F}}; \overline{\mathcal{V}}; x\}$	block
		$\mid$	$x; y$	sequence
Type	X, Y, Z	$=$	p**.type**	singleton type
		$\mid$	p**.**V	abstract type
		$\mid$	$\overline{R}\,\{\overline{O}\}$	record type
		$\mid$	**bottom**	bottom type
Signature	S, T	$=$	**:** X	value signature
		$\mid$	**>:** X **<:** Y	type signature
Refinement	O	$=$	**override** m S	refinement
Miscellaneous				
Integer	i, j, k, l	$=$	$0, 1, 2, \ldots$	

Figure 5.1: Core Syntax (nested version)

classes, functions and blocks. Class instances are explicitly created by instance creation expressions. Function and block instances are implicitly created when they are called or entered. The root context is instantiated only once before the program evaluation starts. In implementation terms,

class instances correspond to the chunks of memory allocated on the heap to store their state, function and block instances correspond to their activation records on the stack and the root context instance corresponds to the static data of the program. Instances are always passed by reference.

Record members are of three different kinds: functions, value variables and type variables. For a class, they correspond to its methods, fields and virtual types. For a function, the value and type variables correspond to its value and type parameters. The members of a function never include functions. For a block, function and variables correspond to its local functions and local variables and for the root context they correspond to the global functions and the global variables of the program.

The unification into records of the root context and all classes, functions and blocks has for consequence that all functions and variables become methods and fields of some record. Thus, functions and variables must always be selected on an instance of some record, even if they correspond to function parameters or to local or global functions or variables. This means that there are no free bindings. Or, more precisely, there is only one free binding, namely the current instance of the innermost enclosing record. This instance is the only value that is directly accessible by any expression. Everything else has to be accessed by successive selections from that instance.

Every entity is owned by some record. This record is called the *owner* of the entity. For all entities, excepted the root context, the owner of the entity is the record in which the entity is defined. For functions and variables, the owner is also the record of which the entity is a member. The root context, which is the outermost record and thus has no enclosing record, is by definition its own owner.

Every record R has two implicit members: an *outer value field* identified by the symbol **`outer`**`@`R and an *outer type field* identified by the symbol **`Outer`**`@`R. The outer value field holds a reference to the current instance of the enclosing record of R and the outer type field holds the type of that instance. In the case of the root context, which has no enclosing record, the outer value field contains a self reference to the root context. Thus, the outer value field of a record R always contains an instance of the owner of R and the outer type field the type of that instance.

Thanks to the implicit outer fields, the current instance of any enclosing record is always accessible through successive outer field selections on the current instance of the innermost enclosing record. The Core language does therefore not need a special syntax, like Java's syntax C`.`**`this`**, to access the current instance of indirectly enclosing records.

Classes, like functions and variables, are always defined within some

record but classes are not regarded as members of their enclosing record because, unlike functions and variables, classes do not necessarily have to be used in conjunction with an instance of their enclosing record. Indeed, calling or accessing functions and variables makes sense only if an instance of their record is provided because their definitions have access to such an instance (the current instance) but it is, for example, perfectly legitimate to express the type of all instances of some class without specify anything about their enclosing instance. Therefore, instead of considering classes as members of their enclosing record, classes are seen as entities that hold a reference to an instance of their enclosing record, namely in their outer field.

5.2.2 Definitions

A program consists of a list of global classes $\overline{\mathcal{C}}$, a list of global functions $\overline{\mathcal{F}}$ and a list of global variables $\overline{\mathcal{V}}$. Global functions and global variables cannot be directly defined in Scala but they may be produced by the desugaring of some constructs. For example, the desugaring of each top-level object produces a global variable. Furthermore, all static methods and all static fields of Java classes are mapped to global functions and global variables. Finally, literals are also mapped to global variables. For example, the integer literal `0` is mapped to the global variable `int$0`.

A class is defined by a symbol C and a list of superclasses $\overline{C}$. It contains a list of inner classes $\overline{\mathcal{C}}$, a list of methods $\overline{\mathcal{F}}$, a list of fields $\overline{\mathcal{V}}$, a list of virtual types $\overline{\mathcal{T}}$ and a list of refinements $\overline{O}$. The member definitions $\overline{\mathcal{F}}$, $\overline{\mathcal{V}}$ and $\overline{\mathcal{T}}$ all define new members while the refinements $\overline{O}$ refine inherited members. A Scala member declaration, for example a type declaration **`type`** V `>:` X `<:` Y, depending on whether it defines a new member or refines an inherited one, is translated into a member definition or a refinement of the overridden member.

A function is defined by a symbol f, a list of overridden functions $\overline{f}$, a list of type parameters $\overline{\mathcal{T}}$, a list of value parameters $\overline{v}$, a return type X and a body x. An important difference with Scala functions is that in a Core function not only its type parameters but also all its value parameters can be referenced in the bounds of the type parameters, in the types of the value parameters and in the return type of the function. Another difference is that there are no abstract functions as functions always have a body. Abstract Scala methods are translated into functions that recursively call themselves with the same arguments. The list of overridden functions $\overline{f}$ specifies which functions are overridden by the defined function f. It may

be non-empty only if f is a method of some class C. Furthermore, all functions in $\overline{f}$ must be inherited by C. Whenever a function in $\overline{f}$ is called on an instance of C, the body of f is evaluated instead of its own one. Thus, the implementation of f overrides the implementation of the functions $\overline{f}$.

A value variable is defined by a symbol v and a type X. The variable is mutable if and only if its definition includes the modifier **`mutable`**. Unlike in Scala, variable definitions do not include an initializer expression. Thus, global and local variables and class fields must always be explicitly initialized before they are read. Function parameters are automatically initialized by function calls.

A type variable is defined by a symbol V, a lower bound X and an upper bound Y. Scala type variables defined with a type value instead of type bounds are translated into type variables with equal bounds.

5.2.3 Expressions

Expressions describe values and types but also refinements and signatures. Refinements are used in class definitions and in record types to specify which and how inherited members are refined. Signatures are used in refinements to specify the new attributes of the refined member.

The type system of the Core language requires that some value expressions are such that successive evaluations always return the same value. These expressions are called *stable value expressions* or simply *stable expressions*. In the syntax, the typing rules and all other formal descriptions, value expressions that need to be stable are denoted by the metavariables p, q and r while plain value expressions are denoted by x, y and z.

5.2.3.1 Value Expressions

The current instance is denoted by the keyword **`this`**. Unlike in Scala, it does not necessarily designate an instance of a class. It designates an instance of the innermost enclosing record. So, within a method, it does not designate an instance of the class of the method but an instance of the method itself. The class instance may however still be accessed through the outer field of the method.

Strictly speaking, the syntax **`this`** is syntactic sugar because the Core syntax requires that the keyword **`this`** is suffixed with a non-negative integer index. This index only plays a role within record types; elsewhere it has to be equal to zero. The exact meaning of the index and the desugaring of **`this`** are explained in Section 5.2.7.

The syntax for variable evaluations and variable assignments requires that the variable is selected on some value. This results from the fact that not only classes but also functions, blocks and the root context are regarded as different kinds of records. Thus, fields but also function parameters, local variables and global variables must always be selected on an instance of their respective record.

Variable assignments are used both to initialize variables, including immutable ones, and to update the value of initialized mutable variables. Some code transformations need to handle variable assignments differently depending on whether they initialize a variable or update its value. It is therefore assumed that for any given variable assignment it is possible to determine whether it corresponds to a variable initialization or not.

The syntax for function applications is similar to Scala's syntax for method applications. Like variables, functions must always be selected on some value because not only methods but also local and global functions are record members. One difference with Scala's syntax is that type arguments are mandatory; there is no type inference. Another difference is that value arguments have to be stable expressions. This is due to the fact that the value parameters of a function can be referenced in the parameters and the return type of the function.

An instance creation expression creates a new instance of a type X. It does not call any constructor. All fields of the new instance, including its outer fields, are therefore uninitialized and must be explicitly initialized before they are evaluated.

A conditional expression consists of a condition x, two alternatives y and z and a type X, which specifies the type of the whole expression. Its presence avoids the need to compute the least upper bound of the type of the two alternatives in order to determine the type of the conditional expression.

A block is defined by a symbol b and a type X. It contains a list of local classes $\overline{\mathcal{C}}$, a list of local functions $\overline{\mathcal{F}}$, a list of local variables $\overline{\mathcal{V}}$ and a value x. The evaluation of a block expression evaluates x and returns the resulting value. The symbol b is needed to express the type of the block's current instance within its members $\overline{\mathcal{F}}$ and $\overline{\mathcal{V}}$ and its value x. It is also needed in the syntax of the symbols of its outer value and type fields. The type X specifies the type of the whole expression. It is expressed in the context of the block's owner; within X, **`this`** denotes an instance of the block's owner. Its presence avoids the need to widen the type of the value x in order to compute the type of the block expression.

The sequence expression evaluates two expressions x and y one after the other and returns the value returned by the second one.

5.2.3.2 Type Expressions

Singleton types are exactly like in Scala and *abstract types* are like Scala's member types. As all type variables, whether they are virtual types or function type parameters, are record members, abstract types are not only used to reference virtual types but also function type parameters.

A *record type* consists of a list of record symbols $\overline{R}$ and a list of refinements $\overline{O}$. It combines in a single construct Scala's notions of class types, compound types and refined types. A value of the record type is such that it is an instance of all the records $\overline{R}$. Furthermore, its members are such that the constraints specified by the refinements $\overline{O}$ are fulfilled.

In Scala, compound types specify a list of inherited classes. Core record types are more general and specify a list of inherited records. This is needed because record types must also be able to describe instances of records that are not classes. For example, within a function f, the current instance **`this`** has the type f `{}`.

The Core language has no special classes, like `Any` and `All`, that play the role of top and bottom types. Instead, the type `{}` (a record type with an empty list of records and an empty list of refinements) is a natural top type and there is an explicit bottom type **`bottom`**.

5.2.3.3 Refinements and Signatures

A refinement consists of a member symbol m and a signature S. It specifies that in the class or the record type in which the refinement occurs the attributes of the inherited member m are superseded by those specified in the signature S.

There are two kinds of signatures: *value signatures*, which are used in conjunction with value variables and functions, and *type signatures*, which are used in conjunction with type variables. The type X of a value signature refines the type or the return type of the refined value variable or function. The types X and Y of a type signature refine the lower and upper bounds of the refined type variable.

5.2.4 Example

To illustrate the syntax of the Core language, we consider the translation of the following Scala code.

```
class Math {
  def factorial(n: Int): Int = {
    def loop(a: Int, i: Int): Int =
```

```
      if (i > n) a else loop(a * i, i + 1);
    loop(1, 1);
  }
}
```

The result of the translation is given below. It is very similar to the original Scala code. In fact, the declarations of the class `Math`, the method `factorial` and the local function `loop` remain unchanged. The conditional expression is just augmented with the type of its branches while the block expression is augmented with an identifying symbol `local` and the type of its result.

```
class Math {
  def factorial(n: Int): Int = {local: Int |
    def loop(a: Int, i: Int): Int =
      if [Int](this.i > this.outer@loop.outer@local.n) this.a
      else this.outer@loop.loop(this.a * this.i, this.i + 1);
    this.loop(1, 1);
  }
}
```

Most differences occur in parameter references and function calls. They are mainly due to the fact that not only classes but also blocks and functions are regarded as a kind of record with a current instance and that their parameters, local variables and local functions are accessed through that instance like fields and methods of classes. For example, within the function `loop`, the expression **`this`** denotes the current instance or activation record of that function and not the current instance of the enclosing class `Math`. Its parameters `a` and `i` are accessed by selecting them on its current instance. Similarly, the local function `loop` is called like a method by selecting it on the current instance of its enclosing block. In the recursive call, this instance is accessed through the outer field **`outer`**`@loop` of the function `loop`. Outer fields are also used in the reference to the parameter `n` to access the current instance of the function `factorial`.

5.2.5 Syntactic Sugar

The Core code of the previous section is not perfectly conform to the syntax of the Core language. Indeed, it uses some syntactic sugar. First of all, the indexes that should suffix every occurrence of the keyword **`this`** are omitted. It happens that here all these indexes are equal to zero. Then, several empty clauses and empty lists are omitted: empty extends clauses, empty lists of overridden functions, empty lists of type parameters and

type arguments and empty refinement clauses (`Int` instead of `Int {}`) are all omitted. Furthermore, the keywords **`val`** and **`type`** are omitted in parameter definitions. Finally, the Core syntax contains neither infix binary operators nor literals. Like in Scala, the expression `x + y` is syntactic sugar for the method call `x.+(y)` where `+` is the symbol of the called method. And, literals replace references to predefined global variables to which they are mapped. For example, the literal `1` replaces a reference to the predefined global variable `int$1`. The Core code of the previous section with all syntactic sugar removed is given below.

```
class Math extends /* epsilon */ {
  def factorial{}[](val n: Int {}): Int {} = {local: Int {} |
    def loop{}[](val a: Int {}, val i: Int {}): Int {} =
      if [Int {}](this(0).i.>[](this(0).outer@loop.outer@local.n))
        this(0).a
      else
        this(0).outer@loop.loop[](
          this(0).a.*[](this(0).i),
          this(0).i.+[](
            this(0).outer@loop.outer@local.outer@factorial
              .outer@Math.int$1));
    this(0).loop[](
      this(0).outer@local.outer@factorial.outer@Math.int$1,
      this(0).outer@local.outer@factorial.outer@Math.int$1);
  }
}
```

This code is syntactically correct but rather hard to understand; it is obfuscated by all the details required by the Core syntax. These details are very useful to describe formally the type system and code transformations but in code examples they are more a nuisance as they can be easily inferred by the reader and do otherwise only obfuscate the code. The syntactic sugar described above is therefore used in code examples. In addition to that, the syntax *R*`.this` is used. It designates the current instance of the enclosing record *R*. For example, the expression `factorial.this.n` could replace the reference to the parameter `n`. Furthermore, type arguments of function calls and the type of conditional expressions and block expressions that can easily be inferred are omitted. Block symbols that are never referenced are also omitted. Finally, current instances that can be easily inferred are also omitted. Thus, the reference to the parameter `n` could be as simple as `n`.

Thanks to all this syntactic sugar, most examples of Core code look like Scala code. The syntactic sugar is however not systematically used but

only in code parts that are not directly related to the current matter.

5.2.6 Stable Expressions

With the syntactic sugar described in the previous section, the Core code of Section 5.2.4 becomes syntactically correct but it would not type check because the arguments of the recursive call to the function `loop` are not stable. Indeed, the type system of the Core language requires that all member qualifiers but also all value arguments of function calls are stable expressions.

In principle, a stable expression is any value expression such that successive evaluations always return the same value. This is however undecidable for an arbitrary expression. The Core type system does therefore only recognize a subset of all those expressions, namely the current instances and the selections of immutable variables on stable expressions. Thus, the expressions `a * i` and `i + 1` are not recognized as stable expressions and can therefore not be directly used as arguments of a function call. To make call to the function `loop` legal, an auxiliary block with two auxiliary local variables must be used. This is demonstrated in the code below.

```
class Math {
  def factorial(n: Int): Int = {local: Int |
    def loop(a: Int, i: Int): Int =
      if [Int](this.i > this.outer@loop.outer@local.n) this.a
      else {b0: Int |
        val v0: Int;
        val v1: Int;
        v0 = this.outer@b0.a * this.outer@b0.i;
        v1 = this.outer@b0.i + 1;
        this.outer@b0.outer@loop.loop(this.v0, this.v1);
      }
    this.loop(1, 1);
  }
}
```

Transforming non-stable member qualifiers and function arguments into stable expressions is trivial but it makes the code much more verbose. Non-stable member qualifiers and function arguments are therefore used in some code examples as a kind of syntactic sugar.

5.2.7 Outer Fields vs. Indexed Current Instances

Every record R has an implicit outer type field **Outer**@R and an implicit outer value field **outer**@R. The implicit definitions of these fields are the following ones where the record P is the owner of the record R.

```
type Outer@R >: bottom <: P {};
val outer@R: this.Outer@R;
```

These definitions imply that the outer field **outer**@R contains an instance of the owner P of the record R. Within the record R, this instance is, by definition, the current instance of the enclosing record P. Thus, the expression P**.this** is syntactic sugar for the expression **this.outer**@R. More generally, the current instance Q**.this** of an indirectly enclosing record Q is by definition the instance obtained by successively selecting the outer field of the record R and all enclosing records that separate R from Q.

Like outer fields, indexed current instances are a means to select enclosing instances but in the context of record types. Indeed, each record type introduces a current instance, namely the instance that the record type describes. This instance can be referenced within the record type. It must therefore be possible to distinguish the current instance of the different record types, which can be arbitrarily deeply nested, and the one of the current class. That is the role of the index suffixing the keyword **this**. To illustrate this, we consider the Scala code below.

```
abstract class C { type T; type U; }
abstract class D { type V; val c: C { type U >: T <: V; }; }
```

The type of the field `c` is a record type or a refined type in Scala terms. It specifies that the field `c` of any instance of the class `D` must contain an instance of the class `C` whose virtual type `U` is lower bounded by its own virtual type `T` and upper bounded by the virtual type `V` of the instance of the class `D`. This record type clearly references two different instances: the instance of the class `C` whose virtual type `U` is refined and the instance of the class `D` whose field `c` contains the first instance. In other words, the lower bound `T` is syntactic sugar for the expression **this.**`T` where the expression **this** denotes the current instance of the class `C` whose virtual type `U` is refined and the upper bound `V` is syntactic sugar for the expression **this.**`V` where the expression **this** denotes the current instance of the class `D`. The expression **this** denotes here two different instances. This cannot be and that is why the keyword **this** is suffixed by an index. This index indicates which one of the possible current instances is designated. If the expression is nested within n record types, 0 designates the current instance of the innermost record type, $n-1$ the current instance of the outermost record

type and n the current instance of the innermost enclosing record. Indices greater than n are not allowed and cannot be used to designate the current instance of further enclosing records. Given this definitions, it is possible to translate the code above into unambiguous Core code:

```
class C { type T; type U; }
class D { type V; val c: C { override U >: this(0).T
                                        <: this(1).V; }; }
```

Many code example use the keyword **this** without suffix. This expression always designates the current instance of the innermost enclosing record. Thus, it is syntactic sugar for the expression **this**(n) where n is the number of enclosing record types.

Implicit outer fields and indexed current instances are both used to access enclosing instances. Would it be possible to get rid of one of these two mechanisms by generalizing the other one?

One could try to get rid of indexed current instances. In our example, the lower bound of `U` would become **this**`.T` but how could we replace the expression **this**`(1)` of the upper bound? The expression **this.outer**`@C` is clearly wrong as the enclosing record of `C` is the root context and not the class `D`. We could try to treat record types like a new kind of record, name them (give them a symbol) and use their implicit outer fields. It is however unclear how it would be possible to create instances that do inherit those fields as classes can only inherit from other classes and not from record types. Naming record types is also problematic because typing rules tend to rewrite record types into new ones, which would require fresh names whereas the type system of the Core language never needs to create fresh names. For example, if `d` is an instance of the class `D`, the expression `d.c` has the type `C { override U >: this.T <: d.V; }`.

One could also try to get rid of implicit outer fields by generalizing the usage of indexed current instances to reference the current instance of enclosing records. This does not work either, at least if like in Scala, it is possible to designate instances of inner classes without specifying their exact enclosing instance, like in the following example where the variable `d` contains an instance of the inner class `D` whose enclosing instance is unknown.

```
class C { type T; class D { val v: T; } }
val d: C#D;
```

The Core version of this code is given below.

```
class C { type T; class D { val v: this(0).outer@D.T; } }
val d: D {};
```

Symbols				
Record	P, Q, R	$=$	$\texttt{Root} \mid C \mid f \mid b$	
Class	A, B, C	$=$	$\ldots$	
Function	f, g	$=$	$\ldots$	
Block	b	$=$	$\ldots$	
Value	u, v, w	$=$	$\texttt{outer@}R \mid \ldots$	
Type	U, V, W	$=$	$\texttt{Outer@}R \mid \ldots$	
Member	m, n, o	$=$	$f \mid b \mid v \mid V$	
Definitions				
Program	$\mathcal{P}$	$=$	$\overline{\mathcal{D}}$	
Definition	$\mathcal{D}$	$=$	R **defines** $\mathcal{E}$	
Entity	$\mathcal{E}$	$=$	$\mathcal{C} \mid \mathcal{F} \mid \mathcal{B} \mid \mathcal{V} \mid \mathcal{T}$	
Class	$\mathcal{C}$	$=$	**class** C **extends** $\overline{C}\,\{\overline{O}\}$	
Function	$\mathcal{F}$	$=$	**def** $f\{\overline{f}\}[\overline{V}](\overline{v})\texttt{:}\ X = x$	
Block	$\mathcal{B}$	$=$	**block** $b\texttt{:}\ X = x$	
Value	$\mathcal{V}$	$=$	[**mutable**] **val** $v\texttt{:}\ X$	
Type	$\mathcal{T}$	$=$	**type** $V\texttt{>:}\ X\texttt{<:}\ Y$	
Expressions				
Value	$x, y, z,$	$=$	$\texttt{this}(k)$	enclosing context
	p, q, r	$\mid$	$p\texttt{.}v$	variable evaluation
		$\mid$	$p\texttt{.}v = x$	variable assignment
		$\mid$	$p\texttt{.}f[\overline{X}](\overline{p})$	function application
		$\mid$	$p\texttt{.}b$	block evaluation
		$\mid$	**new** X	instance creation
		$\mid$	**if** $[X](x)\ y$ **else** z	conditional
		$\mid$	$x\texttt{;}y$	sequence
Type	X, Y, Z	$=$	$p\texttt{.type}$	singleton type
		$\mid$	$p\texttt{.}V$	abstract type
		$\mid$	$\overline{R}\,\{\overline{O}\}$	record type
		$\mid$	**bottom**	bottom type
Signature	S, T	$=$	$\texttt{:}\ X$	value signature
		$\mid$	$\texttt{>:}\ X\texttt{<:}\ Y$	type signature
Refinement	O	$=$	**override** $m\ S$	refinement
Miscellaneous				
Integer	i, j, k, l	$=$	$0, 1, 2, \ldots$	
Symbol	Σ	$=$	$R \mid v \mid V$	
Expression	e	$=$	$x \mid X \mid S \mid O$	
Environment	Γ	$=$	$\overline{X}$	
Substitution	σ	$=$	$\epsilon \mid \sigma, p \mapsto q \mid \sigma, X \mapsto Y$	

Figure 5.2: Core Syntax (flat version)

If the implicit outer fields were replaced by indexed current instances, the class `D` would no longer have implicit outer fields and the expression `this(0).outer@D` would be replaced by the expression `this(1)`. With this change, it would no longer be possible to type the expression `d.v` although `v` is obviously a member of `d`. Indeed, the only possible type of this expression is `d.outer@D.T`.

Thus, implicit outer fields and indexed current instances are both necessary. This can be understood by the fact that these two mechanisms describe two different kinds of relationships. The implicit outer fields describe the fact that given an instance of an enclosed record it is possible to retrieve an instance of its enclosing record. The indexed current instances describe the fact that the members of an instance are constrained by other instances (the current instances of the enclosing record types and records). In general, given an instance that conforms to a record type it is not possible to retrieve an instance that conforms to an enclosing record type or an instance of an enclosing record.

5.2.8 Flat Syntax

With the syntax described in Figure 5.1, ownership is expressed through nesting; every entity is syntactically nested in the record that owns it. This is very intuitive and eases the reading and writing of Core code but it has also a drawback: the separation between definitions and expressions is not perfect. Indeed, a block expression also plays the role of a definition; it defines a new block and introduces a fresh block symbol. Furthermore, it contains nested classes, functions and variables. Thus, it is untrue that expressions only reference symbols; they also define new symbols through block expressions and their nested definitions. This slightly complicates the description of the typing rules but the main problem is that it severely complicates the description of code transformations. Indeed transformations like lambda-lifting that lift entities like classes and functions into enclosing records cannot be described as processes that simply transform every definition and expression into a new one because definitions and expressions may contain nested entities that have to be lifted out. Thus, the lambda-lifting of a definition or an expression should return a new one plus a set of lifted entities, which must then be plugged into the right enclosing record.

Figure 5.2 describes a flat syntax where ownership is no longer implicitly specified through nesting but explicitly in every definition. A program consists of a list of definitions where every definition describes an entity $\mathcal{E}$

and specifies its owner R. With this syntax, definitions and expressions are truly separated; all definitions are gathered in the list constituting the program and expressions only reference already defined symbols. Lifting an entity into an enclosing record is as simple as modifying its owner in its definition. It is no longer needed to literally move code around.

As all definitions are gathered in the list constituting the program, classes no longer contain the definition of their inner classes and their members, only their refinements remain in their body. Similarly, functions no longer contain type and value variable definitions in their parameter lists. The definitions are replaced with the symbols of the parameters. The presence of these symbols is slightly redundant as the list of parameters of a function could be easily inferred from the list of variables owned by the function. Even the order of the parameters could be inferred from the order of the variables in that list. However, the presence of these symbols simplifies the description of some typing rules and some code transformations.

Block expressions are affected in two ways by the flat syntax. First of all, like classes, blocks no longer contain the definitions of their nested entities. Thus, the body of a block consists only of a value expression. Then, the flat syntax distinguishes the definition of blocks from their evaluation. A block is a new kind of entity, which is defined at the top-level like any other function or variable. The evaluation of a block is similar to a function call without arguments. It specifies the block b to evaluate and an instance p of its owner. So, blocks, like functions and variables, are selected on an instance of their owner and constitute therefore a new kind of record member.

The transcription into the flat syntax of the Core code of Section 5.2.4 is given below. Observe how all entities, including function parameters, are defined at the top-level and how the block expression that constituted the body of the function factorial has been replaced with the block evaluation `this.local` and a block definition at the top-level.

```
Root defines class Math;
Math defines def factorial(n): Int = this.local;
factorial defines val n: Int;
factorial defines block local: Int = this.loop(1, 1);
local defines def loop(a, i): Int =
  if [Int](this.i > this.outer@loop.outer@local.n) this.a
  else this.outer@loop.loop(this.a * this.i, this.i + 1);
loop defines val a: Int;
loop defines val i: Int;
```

The code above is a good example of the fact that code using the flat syntax is much harder to understand than the same code using the nested syntax. Therefore, all examples are given with the nested syntax although all formal descriptions of the type system and the code transformations use the flat syntax. It should be clear that the two versions differ only in their syntax but are otherwise equivalent. It is true that the flat version is slightly more expressive. For example, it is possible to define blocks that are owned by a class or evaluate the same block at two different locations with the flat syntax but not with the nested one. However, there are no fundamental differences and if needed these small additional freedoms could easily be forbidden by adding adequate constraints to the rules of program well-formedness.

As the flat syntax is used in all formal descriptions it introduces also some auxiliary notions that are used in those descriptions. The metavariable Σ is used to range over any kind of symbol and the metavariable e to range over any kind of expression. Environments and substitutions are used in some typing rules. An environment Γ consists of a list of types. A substitution σ replaces stable expressions by other stable expressions and types, which are always abstract types, by other types, which may be of any kind.

5.3 Encodings

This section describes how Scala code is encoded into Core code. It describes also how existing Java code is seen from Core code. The possibility to access existing Java code is needed because Scala code can access existing Java code. The same must therefore be true for Core code.

5.3.1 Classes and Class Members

The encoding of classes and class members is straightforward. There are only some subtleties in how the implementation, overriding and refinement of inherited members are encoded. All this is illustrated with the Scala code below.

```
abstract class A {
  type T;
  def f(t: T): T;
  val v: String;
}
abstract class B extends A {
```

```
  type T <: Object;
  def f(t: T): T = t;
  val v: String = "B";
}
class C extends B {
  type T = String;
  override def f(t: T): T = v;
  override val v: String = "C";
}
```

The class A defines a virtual type T, an abstract method f and an abstract field v. The class B refines the upper bound of the virtual type T and implements the method f and the field v. The class C definitely assigns a type to the virtual type T and overrides the implementation of the method f and the field v. The Core version of these three classes is given below. It assumes that the three Scala classes have no constructor at all. Indeed, the encoding of constructors is non-trivial and is described separately in Section 5.3.6.

```
class A {
  type T1 >: bottom <: {};
  def f1(t1: T1): T1 = f1(t1);
  def v1(): String = v1();
}
class B extends A {
  override T1 >: bottom <: Object;
  def f2{f1}(t2: T1): T1 = t2;
  def v2{v1}(): String = v$2;
  val v$2: String;  // initialized to "B" in constructor
}
class C extends B {
  override T1 >: String <: String;
  def f3{f2}(t3: T1): T1 = v3();
  def v3{v2}(): String = v$3;
  val v$3: String;  // initialized to "C" in constructor
}
```

Classes are mapped to corresponding classes with the same class hierarchy. Thus, classes that mixin other classes are mapped to classes with more than one superclass. Inner and local classes are encoded like top-level classes but are mapped to inner and local classes.

Virtual type definitions are mapped to type variable definitions while refinements and definite type assignments of inherited virtual types are both mapped to refinements of inherited type variables. In this context, a

definite type assignment of a virtual type is regarded as a refinement of its two bounds to the assigned type. Thus, the definition, in the class `A`, of the virtual type `T` is mapped to the definition of the type variable $\texttt{T}_1$. And, the refinement, in the class `B`, and the definite type assignment, in the class `C`, of the virtual type `T` are both mapped to refinements of the inherited type variable $\texttt{T}_1$.

Method definitions are mapped to function definitions. If the method is abstract, the body of the function consists of a simple recursive call to itself with its own parameters as arguments. Thus, the definition, in the class `A`, of the abstract method `f` is mapped to the definition of the function $\texttt{f}_1$, which recursively calls itself.

The implementation and the overriding of an inherited method are both encoded as if they were definitions of new methods excepted for the fact that the functions to which these methods are mapped override the inherited function that corresponds to the implemented or overridden method. Thus, the implementation, in the class `B`, of the function `f` is encoded by the definition of the function $\texttt{f}_2$, which overrides the inherited function $\texttt{f}_1$, which corresponds to the implemented method `f`. And, the overriding, in the class `C`, of the method `f` is encoded by the definition of the function $\texttt{f}_3$, which overrides the inherited function $\texttt{f}_2$, which corresponds to the overridden method `f`. With this encoding, a single method can be mapped to several functions. For example, any instance of the class `C` inherits the three functions $\texttt{f}_1$, $\texttt{f}_2$ and $\texttt{f}_3$, which all correspond to the method `f`. Any of these three functions could be called on an instance of the class `C`, but all would yield the same result because the function $\texttt{f}_3$ overrides the two others. Indeed, the function $\texttt{f}_3$ explicitly overrides the function $\texttt{f}_2$, which itself explicitly overrides the function $\texttt{f}_1$. So, by transitivity, the function $\texttt{f}_3$ also overrides the function $\texttt{f}_1$.

Fields are encoded in two steps. They are first desugared as described in Section 2.3.3.2 and then the resulting getter and setter methods and the primitive fields are encoded. The getter and setter methods are encoded like normal methods and definitions of primitive fields are mapped to definitions of value variables. For recollection, the definition of a field is desugared into the definition of a primitive field and the definition of a getter and a setter method, which read and update the primitive field. If the field is immutable, the setter method is omitted. If the field is abstract, the primitive field is omitted and the getter and setter methods are made abstract. The implementation and overriding of inherited fields are desugared in the same way but implementations and overridings of inherited getter and setter methods are generated instead of definitions of new ones. In our example, the definition, in the class `A`, of the abstract field `v`

is encoded by the definition of the function v_1. The implementation, in the class `B`, of the field `v` is encoded by the definition of the value variable `v$`$_2$ and the definition of the function v_2, which overrides the inherited function v_1. The overriding, in the class `C`, of the field `v` is encoded by the definition of the value variable `v$`$_3$ and the definition of the function v_3, which overrides the inherited function v_2.

The encoding of covariant methods and covariant fields is more complex and is described in Section 5.5.1.

5.3.2 Singleton Objects

A singleton object is mapped to a class, a value variable and an access function; the class contains the inner classes and the members of the object, the value variable stores the unique instance of the class and the access function initializes the variable, if needed, and returns its content. All references to the object are mapped to calls of the access function. For example, an object `0` is mapped to the code below and any reference to the object `0` is mapped to the function call `0$f()`.

```
class 0$C { /* inner classes and members of the object 0 */ }
val 0$v: 0$C;
def 0$f(): 0$v.type = { if (0$v == null) 0$v = new 0$C; 0$v };
```

The type of the access function `0$f` is not simply the record type `0$C` but the singleton type `0$v.type`. This lets the type system establish that all calls to the access function always return the same value.

5.3.3 Packages

From a programmer's perspective, packages are perceived as special objects that can never be passed as a value but can only be used to select package members. For example, if an object `0` is defined in a package `foo`, the object `0` can be referenced with the expression `foo.0` but it is forbidden to use the expression `foo` as a value, to pass it as a function argument, for example. For programmers, this encoding is very attractive because a package can be regarded as a kind of restricted object and this decreases by one the number of different notions to learn.

For the compiler, this same encoding is much less attractive. Indeed, at runtime, there exist no values that correspond to packages. So, the expression `foo` in a selection like `foo.0` does not represent any real value. No code needs to be generated for it and it could simply be dropped. In fact, that is exactly what is done.

Packages are almost completely ignored by the Core language. Any entity defined in a package is mapped to an entity owned by the root context. The identity of the package in which the entity was defined is therefore lost. The only aspect of packages that is kept is the fact that each one defines a new namespace. Thus, if two classes with the same name are defined in two different packages, they are mapped to two different classes with different symbols. This encoding has the advantage that there exists only a single value that does not really exist at runtime and for which no code needs to be generated: the unique instance of the root context.

Although the Core language has no notion of package, the compiler should somehow remember for each class the package in which it was defined such that when the code generator finally generates a `class` file for each class, it can place it in the right package.

5.3.4 Java Classes

The encoding of Java classes is even more straightforward than the one of Scala classes; classes, methods and fields are respectively mapped to classes, functions and value variables. There are no getter and setter methods for fields. The only difficulty comes from the fact that in addition to inner classes and instance members Java classes can also contain static classes and static members as illustrated by the class below.

```
public class Java {
  public static class C { /* ... */ }
  public static String f() { /* ... */ }
  public static String v;
  public class D { /* ... */ }
  public String g() { /* ... */ }
  public String w;
}
```

Static entities, like entities defined in packages, are mapped to entities owned by the root context. All other entities remain in place. The Core version of the class `Java` is given below.

```
class Java {
  class D { /* inner classes and instance members of D */ }
  def g(): String = { /* ... */ }
  mutable val w: String;
}
class C { /* inner classes and instance members of C */ }
def f(): String = { /* ... */ }
```

```
mutable val v: String;
/* static classes and static members of D */
/* static classes and static members of C */
```

This encoding is very different from the one perceived by programmers. From a programmer's perspective, each Java class is mapped to a class and an object of the same name. The class contains all the inner classes and instance members of the Java class while the object contains all the static classes and static members. The encoding of the class `Java` is given below.

```
class Java {
  class D { /* inner classes and instance members of D */ }
  object D { /* static classes and static members of D */ }
  def g(): String = { /* ... */ }
  mutable val w: String;
}
object Java {
  class C { /* inner classes and instance members of C */ }
  object C { /* static classes and static members of C */ }
  def f(): String = { /* ... */ }
  mutable val v: String;
}
```

This encoding has the advantage that static members can be accessed like in Java. For example, one can call the static method `f` with the expression `Java.f()`. For the compiler, this encoding has the same disadvantages as the encoding for packages: it creates many objects, like the object `Java`, for which there exist no values at runtime.

5.3.5 Types

The encoding of most types is straightforward. Singleton types and member types are trivially mapped to equivalent singleton types and abstract types. Class types, compound types and refined types are all mapped to record types. The sole types that must be encoded are type parameters and parameterized class types because there are no type parameters, and also qualified class types because classes have no qualifier in record types.

Function type parameters, like function value parameters, are mapped to members of the function's record. They are referenced by selecting them on the current instance of the function. Function type parameters and function return types can be referenced by the bounds and the types of the function parameters. Therefore, the keyword **`this`** designates the current

instance of the function and not its enclosing record in the bounds and the types of its parameters and in its return type. Thanks to this, it is possible to reference the type parameters but also the value parameters of the function. It is therefore possible to encode the following Scala code.

```
class C { type T; val t: T; }
def f[X <: C](x: X, y: x.T): x.type = x;
```

The Core version is given below. If the expression `p` is an instance of the class `C`, the expression `f(p, p.t)` is a legal call of the function `f`. Its type is `p.type`. This type contains one of the arguments of the function call. The expected type of the second argument is `p.T`. It contains also the first argument of the function. These occurrences must be stable expressions and that is the reason why value arguments of function calls must be stable expressions.

```
class C { type T; val t: this.T; }
def f[X <: C](x: this.X, y: this.x.T): this.x.type = this.x;
```

Class type parameters and parameterized class types are encoded as described in Section 2.10. This implies that each type parameter, whether it is covariant, contravariant or invariant is mapped to a virtual type with the same bounds. A reference to a type parameter is mapped to a selection of the corresponding virtual type on the current instance of its class. Parameterized class types become record types with a refinement of the virtual types corresponding to its type parameters. Arguments that correspond to covariant parameters become the upper bound of the refinement. Arguments that correspond to contravariant parameters become the lower bound of the refinement. And, arguments that correspond to invariant parameters become the lower and upper bounds of the refinement. Similarly, the type arguments of the supertype of subclasses of parameterized classes become refinements but here all arguments become the lower and the upper bound of the refinement. The difference is due to the fact that in a supertype, the type arguments provide the effective values of the type parameters while in a class types the type arguments specify a constraint on the expected value of the type parameters. And, in fact, the variance of a type parameter specifies which kind of constraint is applied: covariant means constrain the upper bound, contravariant means constrain the lower bound and invariant means constraint both bounds. To illustrate all this, we consider the Scala code below.

```
abstract class C[+X, Y, -Z] {
  def x(): X; def y(): Y; def z(): Z;
}
```

```
abstract class D extends C[Int, Int, Int] {
  def c(): C[Int, Int, Int];
}
```

This code is translated to the following Core code:

```
class C {
  type X; type Y; type Z;
  def x(): this.X; def y(): this.Y; def z(): this.Z;
}
class D extends C {
  override X >: Int <: Int;
  override Y >: Int <: Int;
  override Z >: Int <: Int;
  def c(): C {
    override X       <: Int;
    override Y >: Int <: Int;
    override Z >: Int      ; };
}
```

Wildcard type arguments of Java 5 [35] can also be encoded. An extends wildcard like <? **extends** X> becomes a refinement of the upper bound and a super wildcard like <? **super** X> becomes a refinement of the lower bound. A normal type argument of Java 5 becomes a refinement of both bounds. This shows that with its wildcards, Java 5 is more flexible than Scala as any type parameter can be upper bound, lower bound or assigned (lower and upper bounded) while in Scala it is decided once for all in the definition of the parameter which kind of constraint will be applied to that parameter. On the other hand, Scala is much less verbose to express covariance and contravariance. To illustrate the encoding of wildcards we translate the Java 5 code below.

```
abstract class C<X, Y, Z> {
  public abstract C<? extends Int, Int, ? super Int> c();
}
```

The Core version:

```
class C {
  type X; type Y; type Z;
  def c(): C {
    override X       <: Int;
    override Y >: Int <: Int;
    override Z >: Int      ; };
}
```

Qualified class types are mapped to recored types as described in Section 2.11.5. For example, the class type `p.C` is mapped to the record type `C { override Outer@C <: p.type; }`. Similarly, supertypes that contain qualified class types are mapped to a refinement of the outer type field of the superclass. But, here, both bounds are refined by the qualifier. The reason is the same as for parameterized class types: in the super type, the qualifier is the effective value of the enclosing instance while in a class type it is only a constraint imposed on the enclosing instance.

It is important to understand that in Scala, almost all class types are qualified because most unqualified class types are implicitly qualified. For example, if `I` is an inner class, the type `I` is legal only if there is an enclosing class `D` that is a subclass of the enclosing class of `I` and the type `I` is syntactic sugar for the type `D.this.I`. Even the class type of top-level and local classes are implicitly qualified by the current instance of the root context or the function or block in which they are defined. Or, at least, their encoding requires a refinement of their outer field corresponding to such a qualifier. To demonstrate this, we consider the following Scala code.

```
val v: String;
abstract class C { def f(): v.type; }
def g(c: C): v.type = c.f();
```

If it is assumed that the class type `C` is unqualified, this code is translated to the Core code below.

```
val v: String;
class C { def f(): this.outer@f.outer@C.v.type; }
def g(c: C {}): this.outer@g.v.type = this.c.f();
```

With this translation, the body of the function `g` is not of the right type. Indeed. its type is `this.c.outer@C.v.type`. Thus, it would be necessary to prove that `this.c.outer@C` and `this.outer@g` designate the same value but there is nothing that can lead us to that conclusion unless the type of the parameter `c` is replaced by the following one:

```
C { override outer@C >: this.outer@g.type <: this.outer@g.type; }
```

The sole class types that can be translated into record types without refinements are those of the form `C#I` where the class `C` is the enclosing class of the inner class `I`. Such a type designates an instance of `I` without any additional constraint on its enclosing instance other than the one already expressed by the implicit definition of its outer type fields, which specifies that it must be an instance of the enclosing class `C`.

The encoding of Java class types never requires any refinements. Indeed, the fact that inner classes can be qualified is only used during the

name analysis to determine which inner class is designated. Thus, the Java type `D.I` is translated into the Core type `I {}` even if the class `D` is a subclass of the enclosing class of the inner class `I`. The reason why the class `D` can be forgotten is that the Java type system has neither virtual types nor singleton types and is therefore unable to exploit in any useful way a better knowledge of an enclosing instance. Note however that including the refinement would not hurt.

With Java 5, things are slightly different because the qualifier of an inner class may specify the type parameters of an enclosing instance. Those cannot simply be dropped. If the class `C` has a type parameter `T` and an inner class `I`, then the type `C<String>.I` is translated into the following record type:

```
I { override outer@I <: C { override T >: String <: String; }; }
```

This corresponds also to the translation of the Scala type `C[String]#I`. The exact class of the enclosing instance is still irrelevant and can still be dropped. For example, if the class `D` was a subclass of `C` that assigns `String` to `T`, the Java type `D.C` could be translated to the same record type as above.

5.3.6 Constructors and Instance Creations

Classes have no constructors in the Core language and instance creations do not implicitly call any initialization function. Constructors must therefore be encoded as normal functions and these must be explicitly called after each instance creation.

A constructor is encoded as a function defined in the same record as its class. It receives the instance to initialize as an argument. This is slightly different from Scala where constructors are members of their class and initialize the current instance. One reason to place constructor outside rather than inside classes is that it simplifies the implementation of lambda lifting. Indeed, free variables of a constructor are not free variables of its class while free variables of a method are free variables of its class. By moving functions corresponding to constructors out of the class they initialize, it is no longer necessary to distinguish between functions corresponding to constructors and those corresponding to normal methods. Another reason is that it is assumed that each function that corresponds to a primary constructor implicitly initializes the outer value field of the new instance. Thus, every constructor must have access to at least two instances: the new instance to initialize and the enclosing instance to put in the outer field. So, one of these two instances must necessarily be passed as an argument.

To encode the constructors of a class, the class is first desugared into a primitive class as described in Section 2.3.4 and the constructors are then normally mapped to Core functions excepted for the fact that they are moved into the enclosing record of their class and all occurrences of **this** are replaced by references to an additional value parameter containing the instance to initialize. For example, let us consider the Scala class below.

```
class C[T](u: T) { val t: T = u; }
```

This class is first desugared into the following primitive class:

```
primitive class C[T] { val t: T; def this(u: T) = this.t = u; }
```

This class is then encoded into the Core code below. Note how the class type parameters are accessed through the instance to initialize in the constructor.

```
class C { type T; val t: T; }
def c(that: C { override Outer@C >: this(1).outer@c.type
                                 <: Root; }, u: that.T): Unit =
  that.t = u;
```

The type of the parameter `that`, which contains the instance to initialize, specifies that its outer type field must be lower bounded by the enclosing instance of the constructor `c`. This is needed because the function `c` corresponds to a primary constructor and thus implicitly contains the following initialization:

```
  that.outer@C = this.outer@c
```

This initialization would not type check in the absence of the lower bound.

Let us now consider a subclass with a primary and a secondary constructor:

```
class D(v: Int) extends C[Int](v) { def this() = this(0); }
```

The desugared Scala version is the following:

```
primitive class D extends C[Int] {
  def this(v: Int) = D.super(v);
  def this() = this(0);
}
```

And, the Core version is given below. The verbose refinements of the outer type field have been left out. They are identical to the one of the constructor `c` but with `C` replaced with `D` and with `c` replaced with $\mathtt{d}_1$ and $\mathtt{d}_2$.

```
class D extends C { override T >: Int <: Int; }
def d1(that: D { /* ... */ }, v: Int): Unit = c(that, v);
def d2(that: D { /* ... */ }): Unit = d1(that, 0);
```

The encoding of each instance creation requires an auxiliary block containing an auxiliary value variable. The instance creation is mapped to the new block. The body of the block first initializes the auxiliary variable with a new instance of the class. The constructor is then called with this new instance. Finally, the block returns the new instance. Let us illustrate this with the following Scala code.

```
def f(): C[Int] = new D();
```

This code is encoded into the Core code below. Note that the type of the local variable must refine the outer type field of the class `D` because otherwise, the call to the function d_2 would not be legal.

```
def f(): C { override T >: Int <: Int; } = { b: D {} |
  val tmp: D {
    override Outer@D >: Root.this.type <: Root.this.type; };
  this.tmp = new D {
    override Outer@D >: Root.this.type <: Root.this.type; };
  Root.this.d2(this.tmp);
  this.tmp
}
```

is-constructor(f)	Returns whether the function f is a constructor.
is-primary-constructor(f)	Returns whether the function f is a primary constructor.
initialized-class(f)	Returns the class whose instances the constructor f initializes.
self-value(f)	Returns the value argument of the constructor f, which contains the instance to initialize (in the examples: the argument `that`).

Figure 5.3: Core Constructor Functions

Functions corresponding to constructors need to be handled in a special way in some operations of the code transformations. For example, code transformations must ensure that functions corresponding to constructors are lifted along with their class. It must therefore be possible to recognize them. That is why it is assumed that the functions described in Figure 5.3 exist.

5.4 Type System

This section describes the type system of the Core language. First, it introduces some auxiliary functions that are used in the formal description of the type system. These functions are also used in the implementation of the code transformations described in Section 5.6 and Section 5.7. The type system is then described in two steps: first all the typing rules and then all the well-formedness rules.

5.4.1 Auxiliary Functions

Figure 5.4 contains the specification of all the auxiliary functions that are used in the type system. It describes also some functions that are only used in the code transformations of Section 5.6 and Section 5.7. The implementation of the auxiliary functions is given in Figure 5.5, Figure 5.6 and Figure 5.7.

The implementation of the auxiliary functions are given in a kind of pseudo-code, which is also used in the formal description of the code transformations. This pseudo-code is a very simple functional language with pattern matching. To avoid any confusion between pseudo-code expressions and Core code expressions, the latter ones are systematically enclosed in double brackets. For example, the expression *is-stable*(⟦`this(0)`⟧) describes a call to the auxiliary function *is-stable* with the Core expression `this(0)` as argument. The Core code within double brackets can reference pseudo-code variables or call auxiliary functions. These variables and functions are distinguished from Core code by their italic font. For example, the expression ⟦p`.v`⟧ represents a Core variable selection whose qualifier is the content of the pseudo-code variable p and whose variable is v. The expression ⟦`outer@`*owner*(f)⟧ represents the outer value field of the owner of the function referenced by the pseudo-code variable f.

The first auxiliary functions are fully described by their description and their implementation and do not need any additional comment.

The function *lookup* returns the signature of the selection of the member m on a value of type $\overline{R}\,\{\overline{O}\}$. The returned signature corresponds to the most refined one that can be found in the type $\overline{R}\,\{\overline{O}\}$. If one of the refinements $\overline{O}$ refines the member m, its signature is returned. Otherwise, a record that refines the member m is searched among the records $\overline{R}$ and their inherited records. If such a record is found, the signature of its refinement of the member m is returned. Otherwise, if no such record is found, the signature declared in the definition of the member m is returned. The search for a record that refines the member m examines the records $\overline{R}$ and

$\mathit{symbol}(\mathcal{D})$	Returns the symbol defined by the definition $\mathcal{D}$.
$\mathit{definition}(\Sigma)$	Returns the definition of the symbol Σ.
$\mathit{owner}(\Sigma)$	Returns the record in which the symbol Σ is defined.
$R \leqslant Q$	Returns whether the record R inherits from the record Q.
$\mathit{is\text{-}mutable}(v)$	Returns whether the value variable v is mutable.
$\mathit{is\text{-}stable}(x)$	Returns whether the value expression x is stable.
$\mathit{signature}(m)$	Returns the declared signature of the member m.
$\mathit{lookup}(m, \overline{R}, \overline{O})$	Returns the signature of the member m when it is selected on a value of type $\overline{R}\,\{\overline{O}\}$.
$\mathit{sink}(e, i)$	Sinks the expression e by i nesting levels. This function increments all free **`this`** indices in the expression e by i.
$\mathit{lift}(e, p)$	Lifts the expression e by one nesting level and replaces occurrences of **`this`**`(0)` by the expression p. This function decrements all free **`this`** indices in the expression e by one.
$\mathit{subst}(e, \sigma)$	Returns the expression resulting from the application of the substitution σ to the expression e.
$\mathit{map\text{-}\mathcal{D}}(F, \mathcal{D})$	Returns the definition resulting from the application of the function F to all subexpressions of the definition $\mathcal{D}$. The function F is passed a record R and an expression e and has to return an expression. The record R corresponds to the current record of the expression e.
$\mathit{map\text{-}e}(F, e, l)$	Returns the expression resulting from the application of the function F to all subexpressions of the expression e. The integer l is the nesting level of the expression e. The function F is passed an expression e' and an integer l' and has to return an expression. The integer l' corresponds to the nesting level of the expression e'.

Figure 5.4: Core Auxiliary Functions

their inherited records from right to left and from child records to parent records. In principle, the retained record is the first found one. However, a later record may still be retained if it refines the member m and furthermore inherits from the formerly retained record. In the implementation of the function *lookup*, the variables S and R are used to keep track of the currently retained signature and the record in which it was found. They are updated each time a refinement of the member m is found in a record that inherits from R. The search starts with the signature found in the definition of the member m and the owner of that definition. The returned

$$
\begin{array}{l}
\mathit{symbol}(\mathcal{D}) = \textbf{case } \mathcal{D} \textbf{ of } [\![P \textbf{ defines } \mathcal{E}]\!] \Rightarrow \textbf{case } \mathcal{E} \textbf{ of} \\
\quad [\![\textbf{class } C \textbf{ extends } \overline{C}\ \{\overline{O}\}]\!] \Rightarrow C \\
\quad [\![\textbf{def } f\{\overline{f}\}[\overline{V}](\overline{v})\texttt{:}\ X = x]\!] \Rightarrow f \\
\quad [\![\textbf{block } b\texttt{:}\ X = x]\!] \Rightarrow b \\
\quad [\![[\textbf{mutable}]\ \textbf{val } v\texttt{:}\ X]\!] \Rightarrow v \\
\quad [\![\textbf{type } V \texttt{>:}\ X \texttt{<:}\ Y]\!] \Rightarrow V \\
\mathit{definition}(\Sigma) = \textbf{case } \Sigma \textbf{ of} \\
\quad [\![\texttt{Root}]\!] \Rightarrow \textbf{undefined} \\
\quad [\![\texttt{outer@}R]\!] \Rightarrow [\![R \textbf{ defines val } \texttt{outer@}R\texttt{: this(0).Outer@}R]\!] \\
\quad [\![\texttt{Outer@}R]\!] \Rightarrow [\![R \textbf{ defines type } \texttt{Outer@}R \texttt{>: bottom <:} \mathit{owner}(R)\ \{\}]\!] \\
\quad \textbf{otherwise} \Rightarrow \text{the definition } \mathcal{D} \text{ defining the symbol } \Sigma \\
\mathit{owner}(\Sigma) = \textbf{case } \Sigma \textbf{ of} \\
\quad [\![\texttt{Root}]\!] \Rightarrow [\![\texttt{Root}]\!] \\
\quad \textbf{otherwise} \Rightarrow \textbf{case } \mathit{definition}(\Sigma) \textbf{ of } [\![P \textbf{ defines } \mathcal{E}]\!] \Rightarrow P \\
R \leqslant Q = (R = Q) \vee \textbf{case } \mathit{definition}(R) \textbf{ of } [\![P \textbf{ defines } \mathcal{E}]\!] \Rightarrow \textbf{case } \mathcal{E} \textbf{ of} \\
\quad [\![\textbf{class } C \textbf{ extends } \overline{C}\ \{\overline{O}\}]\!] \Rightarrow \exists i, C_i \leqslant Q \\
\quad \textbf{otherwise} \Rightarrow \textbf{false} \\
\mathit{is\text{-}mutable}(v) = \textbf{case } \mathit{definition}(v) \textbf{ of } [\![P \textbf{ defines } \mathcal{V}]\!] \Rightarrow \textbf{case } \mathcal{V} \textbf{ of} \\
\quad [\![\textbf{mutable val } v\texttt{:}\ x]\!] \Rightarrow \textbf{true} \\
\quad \textbf{otherwise} \Rightarrow \textbf{false} \\
\mathit{is\text{-}stable}(x) = \textbf{case } x \textbf{ of} \\
\quad [\![\texttt{this(}k\texttt{)}]\!] \Rightarrow \textbf{true} \\
\quad [\![p\texttt{.}v]\!] \Rightarrow \mathit{is\text{-}stable}(p) \wedge \neg \mathit{is\text{-}mutable}(v) \\
\quad \textbf{otherwise} \Rightarrow \textbf{false} \\
\mathit{signature}(m) = \textbf{case } \mathit{definition}(m) \textbf{ of } [\![P \textbf{ defines } \mathcal{E}]\!] \Rightarrow \textbf{case } \mathcal{E} \textbf{ of} \\
\quad [\![\textbf{def } f\{\overline{f}\}[\overline{V}](\overline{v})\texttt{:}\ X = x]\!] \Rightarrow [\![\texttt{:}\ f\ \{\}]\!] \\
\quad [\![\textbf{block } b\texttt{:}\ X = x]\!] \Rightarrow [\![\texttt{:}\ b\ \{\}]\!] \\
\quad [\![[\textbf{mutable}]\ \textbf{val } v\texttt{:}\ X]\!] \Rightarrow [\![\texttt{:}\ X]\!] \\
\quad [\![\textbf{type } V \texttt{>:}\ X \texttt{<:}\ Y]\!] \Rightarrow [\![\texttt{>:}\ X \texttt{<:}\ Y]\!]
\end{array}
$$

Figure 5.5: Core Auxiliary Functions Implementation (1)

signature corresponds to the final value of S.

The functions *sink* and *lift* are used in conjunction with record types. They are used to move expressions inside and out of record types. This usually requires the modification of the index of some current instances. For example, let us consider the two following classes:

$$
\begin{array}{l}
\mathit{lookup}(m, \overline{R}, \overline{O}) = \textbf{if } \exists S, [\![\textbf{override } m\ S]\!] \in \overline{O} \textbf{ then } S \textbf{ else} \\
\quad \mathit{aux}(\mathit{owner}(m), \mathit{signature}(m), \overline{R}) \textbf{ where } \mathit{aux}(R, S, \overline{R}) = \\
\qquad \textbf{let } n = |\overline{R}| \textbf{ in if } n = 0 \textbf{ then } S \textbf{ else let } \overline{R}' = R_1, \ldots, R_{n-1} \textbf{ in} \\
\qquad\quad \textbf{case } \mathit{definition}(R_n) \textbf{ of } [\![P \textbf{ defines } \mathcal{E}]\!] \Rightarrow \textbf{case } \mathcal{E} \textbf{ of} \\
\qquad\qquad \textbf{class } C \textbf{ extends } \overline{C}\ \{\overline{O}\} \textbf{ if } C \leqslant R \Rightarrow \\
\qquad\qquad\quad \exists T, [\![\textbf{override } m\ T]\!] \in \overline{O}\ ?\ \mathit{aux}(C, T, \overline{R}')\ :\ \mathit{aux}(R, S, (\overline{R}', \overline{C})) \\
\qquad\qquad \textbf{otherwise} \Rightarrow \mathit{aux}(R, S, \overline{R}') \\
\mathit{sink}(e, i) = \mathit{aux}(e, 0) \textbf{ where } \mathit{aux}(e, l) = \textbf{case } e \textbf{ of} \\
\quad [\![\textbf{this}(k)]\!] \textbf{ if } k \geq l \Rightarrow [\![\textbf{this}(k + i)]\!] \\
\quad \textbf{otherwise} \qquad \Rightarrow \mathit{map\text{-}e}(\mathit{aux}, e, l) \\
\mathit{lift}(e, p) = \mathit{aux}(e, 0) \textbf{ where } \mathit{aux}(e, l) = \textbf{case } e \textbf{ of} \\
\quad [\![\textbf{this}(k)]\!] \textbf{ if } k = l \Rightarrow \mathit{sink}(p, l) \\
\quad [\![\textbf{this}(k)]\!] \textbf{ if } k > l \Rightarrow [\![\textbf{this}(k - 1)]\!] \\
\quad \textbf{otherwise} \qquad \Rightarrow \mathit{map\text{-}e}(\mathit{aux}, e, l) \\
\mathit{subst}(e, \sigma) = \mathit{aux}(e, 0) \textbf{ where } \mathit{aux}(e, l) = \\
\quad \textbf{if } \exists p \exists q, [\![p \mapsto q]\!] \in \sigma \wedge e = \mathit{sink}(p, l) \quad \Rightarrow \mathit{sink}(q, l) \\
\quad \textbf{if } \exists X \exists Y, [\![X \mapsto Y]\!] \in \sigma \wedge e = \mathit{sink}(X, l) \Rightarrow \mathit{sink}(Y, l) \\
\quad \textbf{otherwise} \qquad\qquad\qquad\qquad\quad \Rightarrow \mathit{map\text{-}e}(\mathit{aux}, e, l)
\end{array}
$$

Figure 5.6: Core Auxiliary Functions Implementation (2)

```
class C { type T; }
class D { type U; }
```

The type `this(0).T` is a valid type within the class `C`. If we want to specify the type of an instance of the class `D` whose type variable `U` is upper bounded by this type, we cannot simply use this type as the upper bound of a refinement of `U`. Indeed, the type `D { override U <: this(0).T; }` is not well-formed because we try to select `T` on the refined instance `D`, which has no such type variable. The problem is that when a type or an expression is moved inside a record type the indexes of its current instances must be updated; all indexes that are free in the expression must be increased by one for each record type in which they are moved. Thus, the correct type is `D { override U <: this(1).T; }`. The function *sink* moves an expression e inside i record types. The function *lift* moves an expression out of one record type. In that case, the index of all current instances of the record and record types enclosing the record type of which the expression is moved out must be decreased by one. Furthermore, all current instances of the record type of which the expression is moved out

must be replaced by some instance of that type. That is why the function *lift* takes a stable expression p as an argument. It is used to replace all these current instances. For example, if p is a stable expression of type `D { override U <: this(1).T; }`, then the type `p.U` has not the signature `<: this(0).T` but the signature `<: p.T`.

$$
\begin{array}{l}
\mathit{map}\text{-}\mathcal{D}(F,\mathcal{D}) = \textbf{case } \mathcal{D} \textbf{ of } [\![P \textbf{ defines } \mathcal{E}]\!] \Rightarrow \textbf{case } \mathcal{E} \textbf{ of} \\
\quad [\![\textbf{class } C \textbf{ extends } \overline{C}\,\{\overline{O}\}]\!] \Rightarrow [\![\textbf{class } C \textbf{ extends } \overline{C}\,\{F(C,\overline{O})\}]\!] \\
\quad [\![\textbf{def } f\{\overline{f}\}[\overline{V}](\overline{v})\texttt{: } X = x]\!] \Rightarrow [\![\textbf{def } f\{\overline{f}\}[\overline{V}](\overline{v})\texttt{: } F(f,X) = F(f,x)]\!] \\
\quad [\![\textbf{block } b\texttt{: } X = x]\!] \Rightarrow [\![\textbf{block } b\texttt{: } F(P,X) = F(b,x)]\!] \\
\quad [\![[\textbf{mutable}]\ \textbf{val } v\texttt{: } X]\!] \Rightarrow [\![[\textbf{mutable}]\ \textbf{val } v\texttt{: } F(P,X)]\!] \\
\quad [\![\textbf{type } V \texttt{>: } X \texttt{<: } Y]\!] \Rightarrow [\![\textbf{type } V \texttt{>: } F(P,X) \texttt{<: } F(P,Y)]\!] \\
\mathit{map}\text{-}e(F,e,l) = \textbf{case } e \textbf{ of} \\
\quad [\![\textbf{this}(k)]\!] \Rightarrow [\![\textbf{this}(k)]\!] \\
\quad [\![p\texttt{.}v]\!] \Rightarrow [\![F(p,l)\texttt{.}v]\!] \\
\quad [\![p\texttt{.}v = x]\!] \Rightarrow [\![F(p,l)\texttt{.}v = F(x,l)]\!] \\
\quad [\![p\texttt{.}f\texttt{[}\overline{X}\texttt{]}(\overline{p})]\!] \Rightarrow [\![F(p,l)\texttt{.}f\texttt{[}F(\overline{X},l)\texttt{]}(F(\overline{p},l))]\!] \\
\quad [\![p\texttt{.}b]\!] \Rightarrow [\![F(p,l)\texttt{.}b]\!] \\
\quad [\![\textbf{new } X]\!] \Rightarrow [\![\textbf{new } F(X,l)]\!] \\
\quad [\![\textbf{if } \texttt{[}X\texttt{]}(x)\ y \textbf{ else } z]\!] \Rightarrow [\![\textbf{if } \texttt{[}F(X,l)\texttt{]}(F(x,l))\ F(y,l) \textbf{ else } F(z,l)]\!] \\
\quad [\![x\texttt{;}y]\!] \Rightarrow [\![F(x,l)\texttt{;}F(y,l)]\!] \\
\quad [\![p\texttt{.type}]\!] \Rightarrow [\![F(p,l)\texttt{.type}]\!] \\
\quad [\![p\texttt{.}V]\!] \Rightarrow [\![F(p,l)\texttt{.}V]\!] \\
\quad [\![\overline{R}\,\{\overline{O}\}]\!] \Rightarrow [\![\overline{R}\,\{F(\overline{O},l+1)\}]\!] \\
\quad [\![\textbf{bottom}]\!] \Rightarrow [\![\textbf{bottom}]\!] \\
\quad [\![\texttt{: } X]\!] \Rightarrow [\![\texttt{: } F(X,l)]\!] \\
\quad [\![\texttt{>: } X \texttt{<: } Y]\!] \Rightarrow [\![\texttt{>: } F(X,l) \texttt{<: } F(Y,l)]\!] \\
\quad [\![\textbf{override } m\ S]\!] \Rightarrow [\![\textbf{override } m\ F(S,l)]\!]
\end{array}
$$

Figure 5.7: Core Auxiliary Functions Implementation (3)

The functions *map-*$\mathcal{D}$ and *map-e* are used to map a function F to all subexpressions of a definition $\mathcal{D}$ or of an expression e. The integer l in the function *map-e* indicates the number of record types in which the expression e is nested. In the case of the function *map-*$\mathcal{D}$, the function F takes two arguments: the subexpression to transform and the symbol of the record enclosing this subexpression. This record corresponds to the record whose current instance is designated by the expression **this** in the subexpression. In the case of the function *map-e*, the function F takes also two arguments:

the subexpression to transform and the number of record types in which it is nested.

5.4.2 Typing Rules

Figure 5.8 describes the various typing relations used in the type system. The typing rules are given in Figure 5.9, Figure 5.10 and Figure 5.11.

$\Gamma \vdash p.m\ S$	The member m selected on the expression p has the signature S.
$\Gamma \vdash X \prec Y$	The type X is a subtype of the type Y.
$\Gamma \vdash S \prec T$	The signature S refines the signature T.
$\Gamma \vdash p \sim q$	The expression p designates the same value as the expression q.
$\Gamma \vdash x \in X$	The expression x has the type X.

Figure 5.8: Core Typing Relations

All relations have an environment Γ, which consists of a non-empty list of types. The first type is the type of the enclosing record and the following ones, if any, are the types of the enclosing record types from the outermost to the innermost.

There is a single rule to determine the signature of a member m selected on an expression p (SG-MBR). This rule checks that the expression p is stable. This check is needed because the expression p may end up in the returned signature. The rule also checks that the member is selected on an value that inherits it.

The subtyping relation is reflexive ($\prec$-REFL) and transitive ($\prec$-TRANS). Singleton types are subtypes if they designate the same value ($\prec$-SNGEQ). Similarly, abstract types are subtypes if they select the same type variables on the same value ($\prec$-ABSEQ). The type **`bottom`** is a subtype of any singleton type ($\prec$-SNGLW) and any record type ($\prec$-RECLW). A singleton type is a subtype of any type of its value ($\prec$-SNGUP). An abstract type is a supertype of its lower bound ($\prec$-ABSLW) and a subtype of its upper bound ($\prec$-ABSUP). The subtyping of record types ($\prec$-RECUP) is the only non-trivial rule. A record type X is a subtype of another record type Y if it inherits from all the records specified by Y and if in addition to that, for every member refined by Y, it has a more refined signature than the one specified in Y.

The comparison of signatures is defined for type signatures ($\prec$-TPSIG) but not for value signatures. This has for consequence that only type vari-

Member Signature $\boxed{\Gamma \vdash p.m\ S}$

$$\text{(SG-MBR)}\quad \frac{\mathit{is\text{-}stable}(p) \qquad \Gamma \vdash p \in \overline{R}\,\{\overline{O}\} \qquad R_i \leqslant \mathit{owner}(m)}{\Gamma \vdash p.m\ \mathit{lift}(\mathit{lookup}(m, \overline{R}, \overline{O}), p)}$$

Subtyping $\boxed{\Gamma \vdash X \prec Y}$

$$\text{(}\prec\text{-REFL)}\quad \frac{}{\Gamma \vdash X \prec X} \qquad \text{(}\prec\text{-TRANS)}\quad \frac{\Gamma \vdash X \prec Y \qquad \Gamma \vdash Y \prec Z}{\Gamma \vdash X \prec Z}$$

$$\text{(}\prec\text{-SNGEQ)}\quad \frac{\Gamma \vdash p \sim q}{\Gamma \vdash p.\mathtt{type} \prec q.\mathtt{type}} \qquad \text{(}\prec\text{-ABSEQ)}\quad \frac{\Gamma \vdash p \sim q}{\Gamma \vdash p.V \prec q.V}$$

$$\text{(}\prec\text{-SNGLW)}\quad \frac{}{\Gamma \vdash \mathtt{bottom} \prec p.\mathtt{type}} \qquad \text{(}\prec\text{-SNGUP)}\quad \frac{\Gamma \vdash p \in X}{\Gamma \vdash p.\mathtt{type} \prec X}$$

$$\text{(}\prec\text{-ABSLW)}\quad \frac{\Gamma \vdash p.V >: X <: Y}{\Gamma \vdash X \prec p.V} \qquad \text{(}\prec\text{-ABSUP)}\quad \frac{\Gamma \vdash p.V >: X <: Y}{\Gamma \vdash p.V \prec Y}$$

$$\text{(}\prec\text{-RECLW)}\quad \frac{}{\Gamma \vdash \mathtt{bottom} \prec \overline{R}\,\{\overline{O}\}}$$

$$\text{(}\prec\text{-RECUP)}\quad \frac{\forall Q \in \overline{Q}, \exists R \in \overline{R}, R \leqslant Q \qquad \Gamma' \stackrel{\text{def}}{=} \Gamma, \overline{R}\,\{\overline{O}\} \qquad \forall i, \exists S, \Gamma' \vdash \mathtt{this}(0).m_i\ S \wedge \Gamma' \vdash S \prec T_i}{\Gamma \vdash \overline{R}\,\{\overline{O}\} \prec \overline{Q}\,\{\mathtt{override}\ \overline{m}\ \overline{T}\}}$$

Signature Comparison $\boxed{\Gamma \vdash S \prec T}$

$$\text{(}\prec\text{-TPSIG)}\quad \frac{\Gamma \vdash X' \prec X \qquad \Gamma \vdash Y \prec Y'}{\Gamma \vdash >: X <: Y \prec >: X' <: Y'}$$

Figure 5.9: Core Typing Rules (1)

ables can be refined in record types and not value variables or functions. The reasons for this restriction are explained in Section 5.5.1.

The value equality relation is reflexive ($\sim$-REFL), symmetric ($\sim$-SYM) and transitive ($\sim$-TRANS). Two values are the same if they select the same value variable on equal values ($\sim$-MBREQ). If a value has a singleton type for type then it necessarily designates the same value as the singleton type since singleton types designate exactly one value ($\sim$-SUBTP).

The value typing relation includes the subsumption rule ($\in$-SUB). Any well-typed stable expression has its own singleton type for type ($\in$-STB). The type of a current instance is given by the environment ($\in$-THS). The call to the function *sink* is needed to adapt the type from the environment

Path Equality $\boxed{\Gamma \vdash p \sim q}$

($\sim$-REFL) $\dfrac{}{\Gamma \vdash p \sim p}$

($\sim$-SYM) $\dfrac{\Gamma \vdash q \sim p}{\Gamma \vdash p \sim q}$

($\sim$-TRANS) $\dfrac{\Gamma \vdash p \sim r \qquad \Gamma \vdash r \sim q}{\Gamma \vdash p \sim q}$

($\sim$-MBREQ) $\dfrac{\Gamma \vdash p \sim q}{\Gamma \vdash p.v \sim q.v}$

($\sim$-SUBTP) $\dfrac{\Gamma \vdash p \in q.\mathbf{type}}{\Gamma \vdash p \sim q}$

Figure 5.10: Core Typing Rules (2)

to the current nesting level. The type of a variable evaluation is the type of the variable ($\in$-MBR). The typing rule for variable assignments ($\in$-ASS) checks that the assigned value conforms to the type of the variable. The type of the assignment is `Unit`. The typing rule of function applications ($\in$-APP) first determines the signature of the selected function. This is done to check that the qualifier is stable and that it inherits the function. The rule then checks that the type arguments are well-formed and that the value arguments are stable and well-typed. Finally, the rule checks that the type and value arguments conform to their bounds and types. For this, the outer type and value fields of the function and its type and value parameters must be replaced by the actual type and values in the declared bounds and types of the parameters of the function. The substitution σ is built therefore. The type of the function application is its declared return type with its outer fields and its parameters replaced with the actual types and values. The typing rule of block evaluations ($\in$-BLK) constrains the qualifier of the block to the expression **`this`**`(0)`. This corresponds to what can be expressed with the nested syntax. The signature of the block is determined to check that the block is inherited by its qualifier. The type of the block is its declared type. The typing rule of instance creations ($\in$-NEW) restrains the type of new instances to record types of a single class. This corresponds to what can be done in Scala. The typing rule checks that the instantiated type is well-formed and that all the most refined signature of all its type variables have equal lower and upper bounds. The type of the instance creation is the instantiated type. The typing rule for conditional expressions ($\in$-IF) checks that the condition is of type `Boolean` and that the two alternatives are of the declared type of the conditional expression. The type of the conditional expression is its declared type. The typing rule for sequences ($\in$-SEQ) checks that both expressions are well-typed. The type of the sequence is the type of its second expression.

Value Typing $\boxed{\Gamma \vdash x \in X}$

$$(\in\text{-Sub}) \quad \frac{\Gamma \vdash x \in X \qquad \Gamma \vdash X \prec Y}{\Gamma \vdash x \in Y} \qquad \frac{\Gamma \vdash p \in X \qquad \mathit{is\text{-}stable}(p)}{\Gamma \vdash p \in p.\mathbf{type}}$$

$$(\in\text{-Stb})$$

$$(\in\text{-Ths}) \quad \frac{i \stackrel{\text{def}}{=} |\overline{Z}| - n}{\overline{Z} \vdash \mathbf{this}(n) \in \mathit{sink}(Z_i, n+1)} \qquad (\in\text{-Mbr}) \quad \frac{\Gamma \vdash p.v : X}{\Gamma \vdash p.v \in X}$$

$$(\in\text{-Ass}) \quad \frac{\Gamma \vdash p.v : X \qquad \Gamma \vdash x \in X \qquad \mathit{is\text{-}mutable}(v)}{\Gamma \vdash p.v = x \in \mathtt{Unit}\,\{\}}$$

$$(\in\text{-App}) \quad \frac{\begin{array}{c}\Gamma \vdash p.f : f\,\{\} \qquad \Gamma \vdash \overline{X} \diamond \qquad \Gamma \vdash \overline{p} \in \overline{p}.\mathbf{type} \\ \mathit{definition}(f) \stackrel{\text{match}}{=} P\ \mathbf{defines\ def}\ f\{\overline{f}\}[\overline{V}](\overline{v}) : X = x \\ \mathit{signature}(\overline{V}) \stackrel{\text{match}}{=} >: \overline{X}' <: \overline{Y}' \qquad \mathit{signature}(\overline{v}) \stackrel{\text{match}}{=} : \overline{Z}' \\ \sigma_1 \stackrel{\text{def}}{=} \mathbf{this}(0).\overline{V} \mapsto \overline{X}, \mathbf{this}(0).\mathbf{Outer@}f \mapsto p.\mathbf{type} \\ \sigma_2 \stackrel{\text{def}}{=} \mathbf{this}(0).\overline{v} \mapsto \overline{p}, \mathbf{this}(0).\mathbf{outer@}f \mapsto p \qquad \sigma \stackrel{\text{def}}{=} \sigma_1, \sigma_2 \\ \Gamma \vdash \mathit{subst}(\overline{X}', \sigma) \prec \overline{X} \qquad \Gamma \vdash \overline{X} \prec \mathit{subst}(\overline{Y}', \sigma) \\ \Gamma \vdash \overline{p} \in \mathit{subst}(\overline{Z}', \sigma)\end{array}}{\Gamma \vdash p.f[\overline{X}](\overline{p}) \in \mathit{subst}(X', \sigma)}$$

$$(\in\text{-Blk}) \quad \frac{\begin{array}{c}\Gamma \vdash \mathbf{this}(0).b : b\,\{\} \\ \mathit{definition}(b) \stackrel{\text{match}}{=} P\ \mathbf{defines\ block}\ b : X = x\end{array}}{\Gamma \vdash \mathbf{this}(0).b \in X}$$

$$(\in\text{-New}) \quad \frac{\begin{array}{c}\Gamma \vdash C\,\{\overline{O}\} \diamond \qquad \Gamma' \stackrel{\text{def}}{=} \Gamma, C\,\{\overline{O}\} \\ \forall V, \exists X, \forall S, \Gamma' \vdash \mathbf{this}(0).V\ S \Rightarrow \Gamma' \vdash >: X <: X \prec S\end{array}}{\Gamma \vdash \mathbf{new}\ C\,\{\overline{O}\} \in C\,\{\overline{O}\}}$$

$$(\in\text{-If}) \quad \frac{\Gamma \vdash X \diamond \qquad \Gamma \vdash x \in \mathtt{Boolean}\,\{\} \qquad \Gamma \vdash y, z \in X}{\Gamma \vdash \mathbf{if}\,[X](x)\ y\ \mathbf{else}\ z \in X}$$

$$(\in\text{-Seq}) \quad \frac{\Gamma \vdash x, y \in X, Y}{\Gamma \vdash x; y \in Y}$$

Figure 5.11: Core Typing Rules (3)

5.4.3 Well-Formedness Rules

Figure 5.12 describes the various well-formedness relations used in the type system. The well-formedness rules are given in Figure 5.13 and Figure 5.14.

$\Gamma \vdash X \diamond$	The type X is well-formed.
$\Gamma \vdash S \diamond$	The signature S is well-formed.
$\Gamma \vdash O \diamond$	The refinement O is well-formed.
$f\{g\} \diamond$	The function f is a well-formed override of the function g.
$\mathcal{E} \diamond$	The entity $\mathcal{E}$ is well-formed.
$\mathcal{P} \diamond$	The program $\mathcal{P}$ is well-formed.

Figure 5.12: Core Well-Formedness Relations

The well-formedness relations for expressions have the same kind of environment as all the typing rules. The other relations do not need any environment. This corresponds to the fact that all definitions are global in the Core language and thus do not depend on any environment.

The type **`bottom`** is always well-typed ($\diamond$-BOT). A singleton type is well-formed if its value is well-typed and stable or equivalently if its value is of its own type ($\diamond$-SNG). An abstract type is well-typed if it has a signature ($\diamond$-ABS), which implies that the qualifier is well-typed, stable and inherits the type variable. A record type is well-formed if its refinements are well-formed and if for any member it inherits there exists a signature that is more refined than all other possible signatures of that member when it is selected on an instance of the record type ($\diamond$-REC).

A value signature is well-formed if its type is well-formed and upper bounded by some record type ($\diamond$-VLSIG). The second constraint is to avoid cyclic definitions of value variables like **`val`** `v: v.`**`type`**. A type signature is well-formed if its bounds are well-formed and its lower bound is lower bounded by **`bottom`** and its upper bound upper bounded by some record type ($\diamond$-TPSIG). The last two constraints are to avoid cyclic definitions of type variables like **`type`** `V >: V <: V`.

A refinement of a type variable is well-formed if it has a well-formed type signature ($\diamond$-TPREF). The signature determination is done to check that the current record does inherit the type variable. There is no rule for refinements of value variables or functions. It is therefore impossible to refine value variables of functions. The reasons for this restriction are explained in Section 5.5.1.

The rule ($\diamond$-IMPL) defines when a function f is a correct overriding of a function g. The first line checks that the owner of f inherits from the owner of g and thus inherits the function g. The rest of the rule checks that the body of the function g could be replaced by a call to the function f with the parameters of the function g. This implies that any call to the function g could be replaced by a call to the function f. Note that this does not imply

Type Well-Formedness $\boxed{\Gamma \vdash X \diamond}$

($\diamond$-BOT) $\dfrac{}{\Gamma \vdash \texttt{bottom} \diamond}$

($\diamond$-SNG) $\dfrac{\Gamma \vdash p \in p.\texttt{type}}{\Gamma \vdash p.\texttt{type} \diamond}$

($\diamond$-ABS) $\dfrac{\Gamma \vdash p.V >: X <: Y}{\Gamma \vdash p.V \diamond}$

($\diamond$-REC) $$\dfrac{\begin{array}{c}\Gamma' \stackrel{\text{def}}{=} \Gamma, \overline{R}\,\{\overline{O}\} \qquad \Gamma' \vdash \overline{O} \diamond \\ \forall m, \exists S, \forall T, \Gamma' \vdash \texttt{this}(0).m\ T \Rightarrow \begin{pmatrix}\Gamma' \vdash S \prec T \\ \Gamma' \vdash \texttt{this}(0).m\ S\end{pmatrix}\end{array}}{\Gamma \vdash \overline{R}\,\{\overline{O}\} \diamond}$$

Signature Well-Formedness $\boxed{\Gamma \vdash S \diamond}$

($\diamond$-VLSIG) $\dfrac{\Gamma \vdash X \diamond}{\Gamma \vdash : X \diamond}$

($\diamond$-TPSIG) $\dfrac{\Gamma \vdash X, Y \diamond \qquad \Gamma \vdash \texttt{bottom} \prec X \qquad \Gamma \vdash Y \prec \overline{R}\,\{\overline{O}\}}{\Gamma \vdash >: X <: Y \diamond}$

Refinement Well-Formedness $\boxed{\Gamma \vdash O \diamond}$

($\diamond$-TPREF) $\dfrac{\Gamma \vdash \texttt{this}(0).V >: X <: Y \qquad \Gamma \vdash >: X <: Y \diamond}{\Gamma \vdash \texttt{override}\ V >: X <: Y \diamond}$

Figure 5.13: Core Well-Formedness Rules (1)

that f and g must have the same parameters and return types. In fact, the overriding function is covariant in its return type and contravariant in its parameters.

A class is well-formed if it does not inherit from itself, its refinements are well-formed and its own record type is well-formed ($\diamond$-$\mathcal{C}$). The last check is needed to ensure that for all members of the class, there is a signature that is more refined than all other signatures of that member when it is selected on an instance of the class. A function is well-formed if its declared return type is well-formed, its body conforms to its declared return type and that each overridden function is a well-formed overriding ($\diamond$-$\mathcal{F}$). The first line of constraint of the rule checks that all variables owned by the function are also parameters of the function. A block is well-formed if its type if well-formed in the context of the owner of the block and the body of the block conforms to its declared type ($\diamond$-$\mathcal{B}$). A value variable is well-formed if its type is well-formed ($\diamond$-$\mathcal{V}$). A type variable is well-formed if its bounds are well-formed ($\diamond$-$\mathcal{T}$).

A program is well-formed if all its entities are well-formed ($\diamond$-$\mathcal{P}$).

Overriding Well-Formedness $\boxed{f\{g\}\ \diamond}$

$$R, Q \stackrel{\text{def}}{=} owner(f, g) \qquad R \neq Q \qquad R \leqslant Q$$
$$definition(g) \stackrel{\text{match}}{=} \mathbf{def}\ g\{\overline{g}\}[\overline{V}](\overline{v})\mathbf{:}\ X = x$$
$$Z \stackrel{\text{def}}{=} g\,\{\mathbf{override\ Outer@}g\,\mathbf{>:\ bottom\ <:}\ R\,\{\}\}$$

($\diamond$-IMPL)
$$\frac{Z \vdash \mathbf{this}(0).\mathbf{outer@}g.f[\mathbf{this}(0).\overline{V}](\mathbf{this}(0).\overline{v}) \in X}{f\{g\}\ \diamond}$$

Entity Well-Formedness $\boxed{\mathcal{E}\ \diamond}$

($\diamond$-$\mathcal{C}$)
$$\frac{\neg C_i \leqslant C \qquad C\,\{\} \vdash \overline{O}\ \diamond \qquad \epsilon \vdash C\,\{\}\ \diamond}{\mathbf{class}\ C\ \mathbf{extends}\ \overline{C}\ \{\overline{O}\}\ \diamond}$$

($\diamond$-$\mathcal{F}$)
$$\frac{\overline{V} = \{V \mid owner(V) = f\} \qquad \overline{v} = \{v \mid owner(v) = f\} \qquad f\,\{\} \vdash \mathbf{:}\ X\ \diamond \qquad f\,\{\} \vdash x \in X \qquad f\{\overline{f}\}\ \diamond}{\mathbf{def}\ f\{\overline{f}\}[\overline{V}](\overline{v})\mathbf{:}\ X = x\ \diamond}$$

($\diamond$-$\mathcal{B}$)
$$\frac{P \stackrel{\text{def}}{=} owner(b) \qquad P\,\{\} \vdash X\ \diamond \qquad b\,\{\} \vdash x \in subst(X, \mathbf{this}(0) \mapsto \mathbf{this}(0).\mathbf{outer@}b)}{\mathbf{block}\ b\mathbf{:}\ X = x\ \diamond}$$

($\diamond$-$\mathcal{V}$)
$$\frac{\forall P, v \neq \mathbf{outer@}P \qquad R \stackrel{\text{def}}{=} owner(v) \qquad R\,\{\} \vdash \mathbf{:}\ X\ \diamond}{[\mathbf{mutable}]\ \mathbf{val}\ v\mathbf{:}\ X\ \diamond}$$

($\diamond$-$\mathcal{T}$)
$$\frac{\forall P, V \neq \mathbf{Outer@}P \qquad R \stackrel{\text{def}}{=} owner(V) \qquad R\,\{\} \vdash \mathbf{>:}\ X\ \mathbf{<:}\ Y\ \diamond}{\mathbf{type}\ V\ \mathbf{>:}\ X\ \mathbf{<:}\ Y\ \diamond}$$

Program Well-Formedness $\boxed{\mathcal{P}\ \diamond}$

($\diamond$-$\mathcal{P}$)
$$\frac{\overline{\mathcal{E}}\ \diamond}{\overline{P}\ \mathbf{defines}\ \overline{\mathcal{E}}\ \diamond}$$

Figure 5.14: Core Well-Formedness Rules (2)

5.5 Advanced Encodings

This section describes some additional encodings that are more involved than those described in Section 5.3. Some of them also require small extensions of the type system.

5.5.1 Covariant Fields and Methods

Thanks to the rule ($\diamond$-IMPL), a function that overrides the implementation of another function can refine its return type covariantly. It can even refine the bounds and the types of its parameters contravariantly. For example, the code below where the function $\texttt{f}_2$ overrides the implementation of the function $\texttt{f}_1$ is perfectly well-formed although the two functions have different return types and parameters of different types.

```
class C {
  def f1(x1: Int): Any = x1;
}
class D extends C {
  def f2{f1}(x1: Any): Int = 0;
}
```

Although the implementation of the function $\texttt{f}_1$ is overridden by the function $\texttt{f}_2$, which refines its return type covariantly, the function $\texttt{f}_1$ is not truly covariant in its return type. Indeed, the expression $p.\texttt{f}_1(q)$ is of type `Any` even if the value p is of type `D`. This comes from the fact that lists of overridden functions are only used at runtime to implement dynamic dispatch but are completely ignored by the type system. Thus, the Core language does not support truly covariant methods. However, combined with a smart name analyzer covariant functions can be simulated. Indeed, in Scala the overridden and the overriding functions necessarily have the same name. In our example, both functions would be named `f`. A smart name analyzer could resolve the name `f` in the expression $p.\texttt{f}(q)$ to the symbol $\texttt{f}_1$ if the value p is of type `C` and to the symbol $\texttt{f}_2$ if it is of type `D`. Thus, the expression $p.\texttt{f}(q)$ would have the type `Int` if the value p is of type `D`.

The Core language does not support covariant fields, either. One could try to simulate them the same way as covariant methods. This would require that fields, like methods, come with a list of overridden fields. One could then for example write the code below where the field $\texttt{v}_2$ overrides the field $\texttt{v}_1$ in the class `D`. This would mean that each time the field $\texttt{v}_1$ is accessed on an instance of the class `D`, the content of the field $\texttt{v}_2$ is returned or assigned.

```
class C {
  val v1: Any;
}
class D extends C {
  val v2{v1}: Int;
}
```

A smart name analyzer could here again resolve the name v in the expression p.v to the symbol v_1 if the value p is of type C and to the symbol v_2 if it is of type D. Thus, the expression p.v would have the type Int and not just Any if the value p is of type D.

This is however not very satisfactory because fields can occur within types where the name analyzer is of no help to simulate covariance. For example, let us assume that the class C contains the method definition below and consider the expression p.g().

```
def g1(): v1.type = v1;
```

A name analyzer will resolve the function g to the symbol g_1 even if the value p is of type D because there is no function that overrides the function g_1 in the class D. The expression p.g() has therefore the type p.v_1.**type** and its upper bound is Any and not Int even if the value p is of type D. The name analyzer is here of no help because it is not involved in the computation of type upper bounds.

To obtain the right upper bound, one solution would be to allow field refinements. One could then add the refinement below to the class D. The expression p.g() would then have the type p.v_2.**type**, which is indeed upper bounded by the type Int. The refinement below could also be implicit given the fact that the field v_2 overrides the field v_1.

```
override v1: v2.type;
```

Allowing arbitrary field refinements without changing anything else is however unsafe. Indeed, it makes it impossible to ensure that the value of a field assignment conforms to the type of the assigned field. For example, the code below would be well-typed. The expression p.i_1("foo"), too. Even if the value p is of type D and even though in that case the field assignment in the body of the function i_1 assigns to the field v_1 a value that does not conform to its refined type. For this reason, field refinements are not allowed in the Core language.

```
class C {
  val v1: Any;
  def i1(x1: Any): Unit = v1 = x1;
}
class D extends C {
  override v1: Int;
}
```

Covariant fields can be encoded even though field refinements are not available and without changing any typing rule. The idea is to define

along with each field a type field that holds the type of the field. For example, in the code below, the type field V_1 holds the type of the field v_1. The class `D` refines the type of the field v_1 to `Int` by refining the upper bound of the type field V_1. The class `E` definitely assigns the type of the field v_1 to `Int`. In this code, unlike in the code above, the method i_1 is ill-formed because here the value x_1 does not conform to the type of the field v_1 but the method j_1 is well-formed. The expression $p.v_1 = 0$ is ill-typed if the value p is of type `C` or `D` but it is well-typed if the value p is of type `E`.

```
class C {
  type V1 <: Any;
  val v1: V1;
  def i1(x1: Any): Unit = v1 = x1; // error
  def j1(y1: V1): Unit = v1 = y1;
}
class D extends C {
  override V1 <: Int;
}
class E extends D {
  override V1 >: Int <: Int;
}
```

This technique is applied to make outer fields covariant. That explains why there is an outer type field for each outer value field. There is an alternative to the simulation of covariant methods, presented at the beginning of the section, that is based on the same technique, but it works only partially. Indeed, the return type of a function can reference its parameters and those are not available in the refinement of a type field that would hold the return type of the function. In Scala, one can define the following methods.

```
class C() {
  def f[T](): Collection[T] = f[T]();
}
class D() extends C() {
  override def f[T](): List[T] = f[T]();
}
```

Encoding these methods would only be possible if type fields could have type parameters. One could then write the following code.

```
class C() {
  type F1[X1] <: Collection { override Elem <: X1; }
  def f1[T1](): F1[T1] = f1[T1]();
}
```

```
class D() extends C() {
  override F1[X1] <: List { override Elem <: X1; }
  def f2{f1}[T2](): F2[T2] = f2[T2]();
}
```

In the Scala compiler, covariant fields are however not handled with the technique described above. Indeed, the Scala compiler desugars fields into access methods and primitive fields and replaces field covariance with method covariance. To illustrate this, let us consider the Scala code below where the field `v` is covariantly refined in the subclass `D`.

```
class C() {
  val v: Any = "";
}
class D() extends C() {
  override val v: Int = 0;
}
```

This Scala code is encoded into the following Core code where the definitions of the constructors of the two classes have been omitted.

```
class C {
  val v$1: Any;
  def v1(): Any = v$1;
}
class D {
  val v$2: Int;
  def v2{v1}(): Int = v$2;
}
```

Now let us assume that the following method is added to the Scala class `C`.

```
  def f(): v.type = v;
```

How should this method be encoded in the Core language? A first try yields the code below. This code is however not well-typed because the body of the function does not conform to its return type. Furthermore, even if it did, it would be impossible to establish that the expression p`.f1()` has the type `Int` even if the value p is of type `D`.

```
  def f1(): v$1.type = v1();
```

The Scala compiler encodes the Scala method `f` to the following Core method.

```
  def f1(): v1().type = v1();
```

Strictly speaking, the return type of this method is ill-formed because the expression v_1`()` is not stable. The Scala compiler does however consider that calls to getter methods are stable. This could in principle be generalized to any function whose returned value is guaranteed to be always the same provided it is called with the same qualifier and the same arguments.

With this encoding, the Scala expression p`.f()` would have the Core type p`.`v_1`().type`. This type is however still not upper bounded by the type `Int` even if the value p is of type `D`. In the Scala compiler, this is obtained by adding to the class `D` the equivalent of the following function refinement, which states that the return type of the function f_1 is upper bounded by the type v_2`().type`.

```
override f1: f { override Result@f1 <: v2().type; }
```

5.5.2 Explicit Self Types

The Core language has no notion of explicit self types and these cannot be directly encoded. To encode the explicit self types as proposed in Section 4.6, the Core language must be slightly extended.

First of all, it must be assumed that each class inherits the value variable `self` and the type variable `Self`, which are implicitly defined as follows:

```
type Self >: this.type <: {};
val self: this.Self;
```

It must also be assumed that the value variable `self` is automatically initialized at instance creation with a reference to the new instance and can therefore never be explicitly initialized.

Given these variables, explicit self types can be encoded. Every class C should override the upper bound of the type variable `Self`. If the class has an explicit self type X, it should override the upper bound with X and otherwise it should override it with the type C `{}`. Furthermore, when translating from Scala to Core, everywhere where the explicit self type of a value x is needed for type checking, the expression x should be translated to x`.self`.

This last operation has for consequence that identical values may be mapped to different expressions. Therefore, the rule described in Figure 5.15 needs to be added to let the type system recognize that these two expressions designate the same value.

Path Equality $\boxed{\Gamma \vdash p \sim q}$

($\sim$-SELF) $$\frac{}{\Gamma \vdash p.\texttt{self} \sim p}$$

Figure 5.15: Path Equality of Explicit Self Fields

5.5.3 Custom Outer Fields

In Scala, list operations like for example mapping or filtering operations are implemented as methods of the class `List`. There are also some operations like flattening or unzipping operations that are not implemented as methods of the class `List` although they would perfectly fit in the class `List`. The problem with these functions is that they do not work on any list but only on lists of a particular type. For example, only lists of lists can be flattened and only lists of tuples can be unzipped. In Scala it is impossible to express such constraints and that is why these functions have to be implemented in a companion object where the list to flatten or to unzip is passed as an argument.

With a small extension of Scala, it would be possible to express these operations as methods of the class `List`. This is demonstrated in the code below where the methods `flatten` and `unzip` have each a guard that specifies a constraint on the type parameter of the receiving instance of the class `List`. A similar extension of C# is described in [10].

```
abstract class List[+X] {
  def flatten[A](): List[A] if X <: List[A];
  def unzip[A,B](): Pair[List[A],List[B]] if X <: Pair[A,B];
}
```

The same result can be achieved in the Core language by augmenting function definitions with a guard that specifies the upper bound of their implicit outer type field. This upper bound used to be implicitly defined to R `{}` where the record R is the owner of the function. The explicit bound specified in a guard can be any subtype of this type. This allows the following definitions of the functions `flatten` and `zip`.

```
class List {
  type X;
  def flatten[A](): List { override X <: A }
    if List { override X <: List { override X <: A; }; };
  def unzip[A,B](): Pair {
      override X0 <: List { override X <: A; };
      override X1 <: List { override X <: B; }; }
```

```
    if List { override X <: Pair {
      override X0 <: A; override X1 <: B; }; };
}
```

This extension requires the modification of the function *definition* such that for outer type fields of functions it returns the upper bound specified in the guard of the function definition. The typing rule of function calls must be augmented to additionally check that the type of the receiver is indeed a subtype of the function's outer type field. The rule of well-formedness for functions must also be augmented to additionally check that the explicit upper bound specifies an instance of the owner of the function.

The same extension could also be done for classes. This would allow the specification of inner classes whose enclosing instance is more constrained.

5.6 Lambda Lift

This section gives a formal description of the lambda lift transformation described in Chapter 3.

5.6.1 Introduction

The lambda lift transformation for Core code is based on the classical algorithm described in Section 3.2 and the generalizations to local classes and mutable variables described in Section 3.3 and Section 3.4. It consists of two stages. During the first stage the extra set of each local class and each local function is computed. This corresponds to the first step of the classical algorithm. The set of mutable variables that are referenced by local definition and whose value must therefore be boxed in a mutable field of an auxiliary object is also computed during this stage. During the second stage, the program is rewritten with all local classes and functions lifted into their first enclosing class or the root context. This corresponds to the last three steps of the classical algorithm.

The objects used to box the value of mutable fields that are referenced by local definitions are all instances of the class `Ref` whose Scala definition is the following one:

```
class Ref[Elem](_elem: Elem) { var elem: Elem = _elem; }
```

This translates to the following Core definitions:

```
def ref(
  that: Ref { override Outer@Ref >: Root.this.type <: Root; },
  _elem: this.that.Elem): Unit =
    this.that.elem = this._elem;
class Ref { type Elem; mutable val elem: this.Elem; }
```

To illustrate the transformation, we consider the transformation of the Core version of the following Scala code:

```
def f[X](x: X): X = {
  class C() { def g(): X = x; }
  class D() extends C() {}
  var v: Int = 0;
  def h(): X = { v = 1; x }
  h()
}
```

The Core version is the following one:

```
def f[X](x: this.X): this.X = {b: this.X |
  def c(that: C {
      override Outer@C >: this.outer@c.type <: b {};
    }): Unit =
      {}
  class C {
    def g(): this.outer@g.outer@C.outer@b.X =
      this.outer@g.outer@C.outer@b.x;
  }
  def d(that: D {
      override Outer@D >: this.outer@d.type <: b {};
    }): Unit =
      this.outer@d.c(this.that);
  class D extends C { override Outer@C = this.outer@D.type; }
  mutable val v: Int;
  def h(): this.outer@h.outer@b.X =
    (this.outer@h.v = 1; this.outer@h.outer@b.x);
  this.v = 0;
  this.h()
}
```

In the Core example above and the following ones, the syntactic sugar **override** V = X stands for the expression **override** V >: X <: X.

An important difference with the Scala version is that references to free variables become variable selections whose prefix consist of one or more implicit outer field selections. For example, the reference to the free

variable x in the body of the function h is replaced with the expression **this.outer**@h.**outer**@b.x in the Core version. This has the interesting side effect that it is possible to determine whether a referenced variable is free just by examining the outer fields occurring in its qualifier. For example, the reference to the variable x contains the outer field of the function h. Thus, as it is necessary to leave this function to access the variable x, it means that the variable x is free in it.

The result of the transformation of the code above is the following one:

```
def c[X$c](x$c: this.X$c, that: C {
    override Outer@C >: this.outer@c.type <: Root {};
    override X$C = this.X$c;
  }): Unit = this.that.x$C = this.x$c;
class C {
  type X$C;
  val x$C: this.X$C;
  def g(): this.outer@g.X$C = this.outer@g.x$C;
}
def d[X$d](x$d: this.X$d, that: D {
    override Outer@D >: this.outer@d.type <: Root {};
    override X$D = this.X$d;
  }): Unit = this.outer@d.c[this.X$d](this.x$d, this.that);
class D extends C {
  type X$D;
  override Outer@C = this.outer@D.type;
  override X$C = this.X$D;
}

def h[X$h](x$h: this.X$h, v$h: Ref {
    override Outer@Ref = this.outer@h.outer@f.type;
    override Elem = Int;
  }): this.X$h = (this.v$h.elem = 1; this.x$h);

def f[X](x: this.X): this.X = {b: this.X |
  val v: Ref {
    override Outer@Ref = this.outer@b.outer@f.type;
    override Elem = Int; };
  this.v = new Ref {
    override Outer@Ref = this.outer@b.outer@f.type;
    override Elem = Int; };
  this.outer@b.outer@f.ref(this.v, 0);
  this.h[this.outer@b.X](this.outer@b.x, this.v)
}
```

5.6.2 Computation of the Extra Sets

Figure 5.16 contains the specification of the functions implementing the computation of the extra sets. It describes also the functions needed to compute the set of mutable value variables whose value must be boxed in an instance of the class `Ref`. The implementation of the functions is given in Figure 5.17 and Figure 5.18.

extra-set(R)	Returns the set of type and value variables for which an extra variable is needed in the record R.
refed-vars$()$	Returns the set of mutable value variables whose value must be boxed in an instance of the class `Ref`.
free-vars(R)	Returns the set of type and value variables that are free in the record R and that are referenced by it or by a record enclosed in R.
call-set(R)	Returns the set of records whose extra sets must be included in the extra set of the record R.
refs-$\mathcal{P}(\mathcal{P})$	Returns the set of all free references in the program $\mathcal{P}$.
refs-$\mathcal{E}(\mathcal{E})$	Returns the set of all free references in the entity $\mathcal{E}$.
refs-e(e)	Returns the set of all free references in the expressions e.
refs-m(p, m)	Returns the set of all free references in the selection $p.m$.

Figure 5.16: Lambda Lift Set Computation Functions

The functions *extra-set* and *refed-vars* are the two ones that are used by the program rewriting algorithm. The functions *free-vars* and *call-set* compute the set of free variables and the call set of a record R needed to compute its extra set. The sets of free variables computed by *free-vars* are also used to implement the function *refed-vars*.

The functions *refs-*$\mathcal{P}$, *refs-*$\mathcal{E}$, *refs-e* and *refs-m* are needed to compute the results of the functions *free-vars* and *call-set*. All of these four functions return a set of free references where each free reference is a pair consisting of a record R and a member m. The record R is either a local class or a local function and the member m is a member that is free in the record R and that is referenced by it or by a record enclosed in it.

The implementation of the functions *refs-*$\mathcal{P}$, *refs-*$\mathcal{E}$ and *refs-e* is straightforward. All the interesting code is in the function *refs-m*. A reference to a free member can generate more than one free reference. Indeed a variable can be referenced from within a function that is itself nested in another function. If the variable is free in both functions then the variable reference generates two free references: one for each function. However, there

$$
\begin{array}{l}
\mathit{refs\text{-}\mathcal{P}}(\mathcal{P}) = \textbf{case } \mathcal{P} \textbf{ of } [\![\overline{R} \texttt{ defines } \overline{\mathcal{E}}]\!] \Rightarrow \bigcup \mathit{refs\text{-}\mathcal{E}}(\overline{\mathcal{E}}) \\
\mathit{refs\text{-}\mathcal{E}}(\mathcal{E}) = \textbf{case } \mathcal{E} \textbf{ of} \\
\quad [\![\texttt{class } C \texttt{ extends } \overline{C}\ \{\overline{O}\}]\!] \Rightarrow \bigcup \mathit{refs\text{-}e}(\overline{O}) \\
\quad [\![\texttt{def } f\{\overline{f}\}[\overline{V}](\overline{v})\texttt{: } X = x]\!] \Rightarrow \mathit{refs\text{-}e}(X) \cup \mathit{refs\text{-}e}(x) \\
\quad [\![\texttt{block } b\texttt{: } X = x]\!] \Rightarrow \mathit{refs\text{-}e}(X) \cup \mathit{refs\text{-}e}(x) \\
\quad [\![[\texttt{mutable}]\ \texttt{val } v\texttt{: } X]\!] \Rightarrow \mathit{refs\text{-}e}(X) \\
\quad [\![\texttt{type } V \texttt{>: } X \texttt{<: } Y]\!] \Rightarrow \mathit{refs\text{-}e}(X) \cup \mathit{refs\text{-}e}(Y) \\
\mathit{refs\text{-}e}(e) = \textbf{case } e \textbf{ of} \\
\quad [\![\texttt{this}(k)]\!] \Rightarrow \epsilon \\
\quad [\![p\texttt{.outer@}R]\!] \Rightarrow \mathit{refs\text{-}e}(p) \\
\quad [\![p\texttt{.}v]\!] \Rightarrow \mathit{refs\text{-}m}(p, v) \\
\quad [\![p\texttt{.}v = x]\!] \Rightarrow \mathit{refs\text{-}m}(p, v) \cup \mathit{refs\text{-}e}(x) \\
\quad [\![p\texttt{.}f[\overline{X}](\overline{p})]\!] \Rightarrow \mathit{refs\text{-}m}(p, f) \cup (\bigcup \mathit{refs\text{-}e}(\overline{X})) \cup (\bigcup \mathit{refs\text{-}e}(\overline{p})) \\
\quad [\![p\texttt{.}b]\!] \Rightarrow \mathit{refs\text{-}e}(p) \\
\quad [\![\texttt{new } X]\!] \Rightarrow \mathit{refs\text{-}e}(X) \\
\quad [\![\texttt{if } [X](x)\ y \texttt{ else } z]\!] \Rightarrow \mathit{refs\text{-}e}(X) \cup \mathit{refs\text{-}e}(x) \cup \mathit{refs\text{-}e}(y) \cup \mathit{refs\text{-}e}(z) \\
\quad [\![x\texttt{;}y]\!] \Rightarrow \mathit{refs\text{-}e}(x) \cup \mathit{refs\text{-}e}(y) \\
\quad [\![p\texttt{.type}]\!] \Rightarrow \mathit{refs\text{-}e}(p) \\
\quad [\![p\texttt{.Outer@}R]\!] \Rightarrow \mathit{refs\text{-}e}(p) \\
\quad [\![p\texttt{.}V]\!] \Rightarrow \mathit{refs\text{-}m}(p, V) \\
\quad [\![\overline{R}\ \{\overline{O}\}]\!] \Rightarrow \bigcup \mathit{refs\text{-}e}(\overline{O}) \\
\quad [\![\texttt{bottom}]\!] \Rightarrow \epsilon \\
\quad [\![\texttt{: } X]\!] \Rightarrow \mathit{refs\text{-}e}(X) \\
\quad [\![\texttt{>: } X \texttt{<: } Y]\!] \Rightarrow \mathit{refs\text{-}e}(X) \cup \mathit{refs\text{-}e}(Y) \\
\quad [\![\texttt{override } m\ S]\!] \Rightarrow \mathit{refs\text{-}e}(S) \\
\mathit{refs\text{-}m}(p, m) = \textbf{case } \mathit{owner}(m) \textbf{ of} \\
\quad (f|b) \Rightarrow \mathit{aux}(p) \textbf{ where } \mathit{aux}(p) = \textbf{case } p \textbf{ of} \\
\qquad [\![p\texttt{.outer@}C]\!] \Rightarrow \{(C, m)\} \\
\qquad [\![p\texttt{.outer@}f]\!] \Rightarrow \{(f, m)\} \cup \mathit{aux}(p) \\
\qquad [\![p\texttt{.outer@}b]\!] \Rightarrow \mathit{aux}(p) \\
\qquad \textbf{otherwise} \Rightarrow \epsilon \\
\quad \textbf{otherwise} \Rightarrow \mathit{refs\text{-}e}(p)
\end{array}
$$

Figure 5.17: Lambda Lift Set Computation Functions Implementation (1)

is not one free reference for every local function or local class that separates the current instance from the referenced variable. Indeed, local func-

$$
\begin{array}{|ll|}
\hline
\textit{extra-set}(R) & = \text{Computed from } \textit{free-vars}(R) \text{ and } \textit{call-set}(R) \text{ with} \\
 & \quad \text{Danvy and Schultz's algorithm [7] (see Section 3.2.1)} \\
\textit{refed-vars}() & = \{v \mid \textit{is-mutable}(v) \wedge \exists R, v \in \textit{free-vars}(R)\} \\
\textit{free-vars}(R) & = \{V \mid (R, V) \in \textit{refs-}\mathcal{P}(\mathcal{P})\} \cup \{v \mid (R, v) \in \textit{refs-}\mathcal{P}(\mathcal{P})\} \\
\textit{call-set}(R) & = \{f \mid (R, f) \in \textit{refs-}\mathcal{P}(\mathcal{P})\} \cup \\
 & \quad \textit{is-primary-constructor}(R) \ ? \ \{\textit{initialized-class}(R)\} \ : \ \epsilon \\
\hline
\end{array}
$$

Figure 5.18: Lambda Lift Set Computation Functions Implementation (2)

tions and classes that are themselves nested in a class do not need an extra variable because they can use the one of the enclosing class. This is illustrated by the code below that contains several nested classes and functions and a single variable reference. For simplicity, blocks were ignored but in principle each block should have its own symbol and the reference to the variable `x` should contain selections of the outer fields of these blocks. The reference to the variable `x` generates the following free references: $(\mathtt{g}, \mathtt{x})$, $(\mathtt{h}, \mathtt{x})$, $(\mathtt{C}, \mathtt{x})$. The pairs $(\mathtt{m}, \mathtt{x})$, $(\mathtt{D}, \mathtt{x})$, $(\mathtt{n}, \mathtt{x})$ and $(\mathtt{o}, \mathtt{x})$ are not included because the records `m`, `D`, `n` and `o` can all access the extra value field of the class `C` in the lifted code.

```
def f(x: Int): Int = {
  def g(): Int = {
    def h(): Int = {
      class C {
        def m(): Int = {
          class D {
            def n(): Int = {
              def o(): Int =
                this.outer@o.outer@n.outer@D.outer@m
                  .outer@C.outer@h.outer@g.x;
              0
            }
          }; 0
        }
      }; 0
    }; 0
  }; 0
}
```

The implementation of the function *free-vars* simply collects for a given record all the variables for which there exists a free reference from the record to the variable. Similarly the function *call-set* collects all the func-

tions for which there exists a free reference from the record to the variable. If the given record is a primary constructor, the initialized class is added to the call set. This is done because primary constructors must initialize the extra fields of the instance to initialize.

The variables whose value must be boxed are all the local mutable variables that are referenced by at least one local record in which they are free.

5.6.3 Program Rewriting

The specification of the functions implementing the program rewriting is given in Figure 5.19 and their implementation in Figure 5.20, Figure 5.21 and Figure 5.22.

$ll\text{-}\mathcal{P}(\mathcal{P})$	Rewrites the program $\mathcal{P}$.
$ll\text{-}\mathcal{D}(\mathcal{D})$	Rewrites the definition $\mathcal{D}$.
$ll\text{-}e(P, e)$	Rewrites the expression e in the context of the record P.
$extra\text{-}\overline{\mathcal{D}}(R)$	Returns the definitions of the extra type and value members of the record R.
$extra\text{-}m(R, m)$	Returns the extra member of the record R for the member m of its extra set.
$extra\text{-}\overline{x}(f)$	Returns the extra initialization expressions for the primary constructor f. This function is used in the rewriting of function definitions in $ll\text{-}\mathcal{D}$.
$extra\text{-}\overline{O}(P, l, R)$	Returns the new overrides for the record R in the context of the record P and the nesting level l. This function is used in the rewriting of class definitions in $ll\text{-}\mathcal{D}$ and record types in $ll\text{-}e$.
$path\text{-}to(P, l, R)$	Returns a path to the current instance of the enclosing record R in the context of the record P and the nesting level l.
$is\text{-}refed(v)$	Returns whether the value of the variable v is boxed in an instance of the class `Ref`.
$is\text{-}initialization(x)$	Returns whether the value expression x is an initialization assignment. We assume that there are two kinds of assignments: initialization and updates. Initializations are assignments that set the initial value of a (mutable or immutable) field. Updates are assignments that modify the value of an initialized mutable field.

Figure 5.19: Lambda Lift Rewriting Functions

The function *ll-*$\mathcal{P}$ first appends to the program the definition of all the extra fields and extra parameters and then transforms all the definitions of the resulting program.

$$
\begin{array}{l}
\textit{ll-}\mathcal{P}(\mathcal{P}) = \textbf{case } \mathcal{P} \textbf{ of } \overline{\mathcal{D}} \Rightarrow \textit{ll-}\mathcal{D}((\overline{\mathcal{D}}, \textit{extra-}\overline{\mathcal{D}}(\{R \mid R \in \textit{symbol}(\overline{\mathcal{D}})\}))) \\
\textit{ll-}\mathcal{D}(\mathcal{D}) = \textbf{case } \textit{map-}\mathcal{D}(\textit{ll-e}, \mathcal{D}) \textbf{ of} \\
\quad [\![P \textbf{ defines } \mathcal{E}]\!] \Rightarrow [\![P' \textbf{ defines } \mathcal{E}']\!] \textbf{ where} \\
\quad\quad P' = \textbf{case } \mathcal{E} \textbf{ of } (\mathcal{C}|\mathcal{F}) \Rightarrow \textit{aux}(P) \textbf{ where} \\
\quad\quad\quad \textit{aux}(R) = \textbf{case } R \textbf{ of } (f|b) \Rightarrow \textit{aux}(\textit{owner}(R)) \textbf{ otherwise} \Rightarrow R \\
\quad\quad \mathcal{E}' = \textbf{case } \mathcal{E} \textbf{ of} \\
\quad\quad\quad [\![\textbf{class } C \textbf{ extends } \overline{C}\ \{\overline{O}\}]\!] \Rightarrow \\
\quad\quad\quad\quad [\![\textbf{class } C \textbf{ extends } \overline{C}\ \{\textit{ll-e}(C, (\overline{O}, \textit{extra-}\overline{O}(C, 0, \overline{C})))\}]\!] \\
\quad\quad\quad [\![\textbf{def } f\{\overline{f}\}[\overline{V}](\overline{v})\texttt{:}\ X = x]\!] \Rightarrow \\
\quad\quad\quad\quad [\![\textbf{def } f\{\overline{f}\}[\overline{V}', \overline{V}](\overline{v}', \overline{v})\texttt{:}\ X = x_1\texttt{;} \ldots \texttt{;} x_{|\overline{x}|}\texttt{;} x]\!] \textbf{ where} \\
\quad\quad\quad\quad\quad \overline{V}', \overline{v}' = \textit{extra-m}(f, \textit{extra-set}(f)) \\
\quad\quad\quad\quad\quad \overline{x} \quad\ = \textit{is-primary-constructor}(f)\ ?\ \textit{extra-}\overline{x}(f)\ :\ \epsilon \\
\quad\quad\quad [\![\textbf{mutable val } v\texttt{:}\ X]\!] \textbf{ if } \textit{is-refed}(v) \Rightarrow \\
\quad\quad\quad\quad [\![\textbf{val } v\texttt{:}\ \texttt{Ref}\ \{\textbf{override } \texttt{Elem>:}\ \textit{sink}(X, 1)\ \texttt{<:}\ \textit{sink}(X, 1)\}]\!] \\
\quad\quad\quad \textbf{otherwise} \Rightarrow \mathcal{E}
\end{array}
$$

Figure 5.20: Lambda Lift Rewriting Functions Implementation (1)

The function *ll-*$\mathcal{D}$ updates the owner of local classes and local functions. In addition to that, it augments classes whose superclasses have extra type fields with extra type refinements. These refinements correspond to the extra type arguments of the superclasses in the Scala version. The function *ll-*$\mathcal{D}$ also adds the extra parameters of functions to their parameter lists and augments the body of primary constructors with initializations of the extra fields of the new instance. Finally, it updates the type of mutable variables whose value must be boxed in an instance of the class `Ref`.

The function *ll-e* relies on two auxiliary functions aux_1 and aux_2. The function aux_1 performs the modifications related to the boxing of the value of mutable variables. The function aux_2 performs all the other modifications.

The first case of the function aux_1 augments references to boxed variables with a selection of the field `elem`. The third case modifies in a similar way updates of boxed variables. The second case handles initializations of boxed variables. In that case, a new instance of the class `Ref` must be created. This instance is used to initialize the variable. The instance must

itself be initialized by calling the constructor of the class `Ref` and passing to it as arguments the new instance and the value that was previously used to initialize the variable.

$$
\begin{array}{l}
\mathit{ll\text{-}e}(P,e) = \mathit{aux}_1(0,e)\ \textbf{where} \\
\quad \mathit{aux}_1(l,e) = \textbf{let}\ e' = \mathit{aux}_2(l,e)\ \textbf{in case}\ e'\ \textbf{of} \\
\qquad [\![p\texttt{.}v]\!]\ \textbf{if}\ \mathit{is\text{-}refed}(v) \Rightarrow [\![p\texttt{.}v\texttt{.elem}]\!] \\
\qquad [\![p\texttt{.}v\texttt{=}x]\!]\ \textbf{if}\ \mathit{is\text{-}refed}(v) \wedge \mathit{is\text{-}initialization}(e') \Rightarrow [\![y\texttt{;}z]\!]\ \textbf{where} \\
\qquad\quad y = [\![p\texttt{.}v\texttt{=}\textbf{new}\ \texttt{Ref}\ \{\mathit{sink}(\overline{O},1)\}]\!] \\
\qquad\quad z = [\![q\texttt{.ref[](}p\texttt{.}v\texttt{,}x\texttt{)}]\!] \\
\qquad\quad \overline{O} = [\![\textbf{override}\ \texttt{Outer@Ref>:}\ q\texttt{.}\textbf{type}\ \texttt{<:}\ q\texttt{.}\textbf{type}]\!], \\
\qquad\qquad\quad [\![\textbf{override}\ \texttt{Elem>:}\ Y\ \texttt{<:}\ Y]\!] \\
\qquad\quad q = \mathit{aux}_1(l,\mathit{path\text{-}to}(P,l,\textbf{Root})) \\
\qquad\quad Y = \textbf{case}\ \mathit{definition}(v)\ \textbf{of}\ [\![R\ \textbf{defines mutable val}\ v\texttt{:}\ X]\!] \Rightarrow \\
\qquad\qquad \mathit{aux}_1(l,\mathit{subst}(X,[\![\textbf{this(0)} \mapsto \mathit{path\text{-}to}(P,l,R)]\!])) \\
\qquad [\![p\texttt{.}v\texttt{=}x]\!]\ \textbf{if}\ \mathit{is\text{-}refed}(v) \Rightarrow [\![p\texttt{.}v\texttt{.elem=}x]\!] \\
\qquad \textbf{otherwise} \Rightarrow e' \\
\quad \mathit{aux}_2(l,e) = \textbf{let}\ e' = \mathit{map\text{-}e}(\mathit{aux}_1,l,e)\ \textbf{in case}\ e'\ \textbf{of} \\
\qquad [\![p\texttt{.}\textbf{outer@}R\texttt{.}\textbf{outer@}(f|b)]\!]\ \textbf{if}\ \forall b, R \neq b \Rightarrow [\![p\texttt{.}\textbf{outer@}R]\!] \\
\qquad [\![p\texttt{.}\textbf{outer@}R\texttt{.}v]\!]\ \textbf{if}\ v \in \mathit{extra\text{-}set}(R) \Rightarrow [\![p\texttt{.}v']\!]\ \textbf{where} \\
\qquad\quad v' = \mathit{extra\text{-}m}(R,v) \\
\qquad [\![p\texttt{.}\textbf{outer@}R\texttt{.}V]\!]\ \textbf{if}\ V \in \mathit{extra\text{-}set}(R) \Rightarrow [\![p\texttt{.}V']\!]\ \textbf{where} \\
\qquad\quad V' = \mathit{extra\text{-}m}(R,V) \\
\qquad [\![p\texttt{.}f\texttt{[}\overline{X}\texttt{](}\overline{p}\texttt{)}]\!] \Rightarrow [\![p\texttt{.}f\texttt{[}\overline{Y},\overline{X}\texttt{](}\overline{q},\overline{p}\texttt{)}]\!]\ \textbf{where} \\
\qquad\quad \overline{Y} = \textbf{let}\ \overline{r} = \mathit{path\text{-}to}(P,l,\mathit{owner}(\overline{V}))\ \textbf{in}\ \mathit{aux}_2(l,[\![\overline{r}\texttt{.}\overline{V}]\!]) \\
\qquad\quad \overline{q} = \textbf{let}\ \overline{r} = \mathit{path\text{-}to}(P,l,\mathit{owner}(\overline{v}))\ \textbf{in}\ \mathit{aux}_2(l,[\![\overline{r}\texttt{.}\overline{v}]\!]) \\
\qquad\quad \overline{V},\overline{v} = \mathit{extra\text{-}set}(f) \\
\qquad [\![\overline{R}\,\{\overline{O}\}]\!] \Rightarrow [\![\overline{R}\,\{\overline{O},\mathit{aux}_1(l+1,\mathit{extra\text{-}}\overline{O}(P,l+1,\overline{R}))\}]\!] \\
\qquad \textbf{otherwise} \Rightarrow e'
\end{array}
$$

Figure 5.21: Lambda Lift Rewriting Functions Implementation (2)

The first case of the function aux_2 removes selections of implicit outer fields of records that are no longer enclosing records. The second and third cases replace references to free variables with references to the corresponding extra parameters or fields. The fourth case adds the required extra type and value arguments to function calls. The fifth case adds extra type refinements to record types. These refinements correspond to the extra type arguments of class types in the Scala version.

$$
\begin{array}{l}
\textit{extra-}\overline{\mathcal{D}}(R) = \textit{extra-}\mathcal{D}(\textit{extra-set}(R)) \textbf{ where} \\
\quad \textit{extra-}\mathcal{D}(m) = \textbf{case } \textit{definition}(m) \textbf{ of} \\
\quad\quad [\![P \textbf{ defines } \mathcal{E}]\!] \Rightarrow [\![R \textbf{ defines } \mathcal{E}'']\!] \textbf{ where} \\
\quad\quad\quad \mathcal{E}'' = \textit{subst}(\mathcal{E}', [\![\texttt{this}(0) \mapsto \textit{path-to}(R, 0, P)]\!]) \\
\quad\quad\quad \mathcal{E}' \;\; = \textbf{case } \textit{extra-m}(R, m), \mathcal{E} \textbf{ of} \\
\quad\quad\quad\quad V', [\![\textbf{type } V \texttt{>:} X \texttt{<:} Y]\!] \quad \Rightarrow [\![\textbf{type } V' \texttt{>:} X \texttt{<:} Y]\!] \\
\quad\quad\quad\quad v', [\![[\texttt{mutable}] \textbf{ val } v \texttt{:} X]\!] \Rightarrow [\![[\texttt{mutable}] \textbf{ val } v' \texttt{:} X]\!] \\
\textit{extra-m}(R, m) = \\
\quad \textbf{if } m \in \textit{extra-set}(R) \textbf{ then } \text{the extra member of } R \text{ for } m \textbf{ else undefined} \\
\textit{extra-}\overline{x}(f) = \textbf{let } \overline{V}, \overline{v} = \textit{extra-set}(C) \textbf{ in } \textit{extra-x}(\overline{v}) \textbf{ where} \\
\quad C, u \;\; = \textit{initialized-class}(f), \textit{self-value}(f) \\
\quad \textit{extra-x}(v) = [\![\texttt{this}(0).u.v' = \texttt{this}(0).w']\!] \textbf{ where} \\
\quad\quad v', w' = \textit{extra-m}(C, v), \textit{extra-m}(f, v) \\
\textit{extra-}\overline{O}(P, l, R) = \textbf{case } \textit{extra-set}(R) \textbf{ of } \overline{V}, \overline{v} \Rightarrow \textit{extra-O}(\overline{V}) \textbf{ where} \\
\quad \textit{extra-O}(V) = [\![\textbf{override } V' \texttt{>:} p.V \texttt{<:} p.V]\!] \textbf{ where} \\
\quad\quad V' = \textit{extra-m}(R, V) \\
\quad\quad p \;\; = \textit{path-to}(P, l, \textit{owner}(V)) \\
\textit{path-to}(P, l, R) = \textit{aux}([\![\texttt{this}(l)]\!], P) \textbf{ where} \\
\quad \textit{aux}(p, Q) = \textbf{if } Q = R \textbf{ then } p \textbf{ else } \textit{aux}([\![p.\texttt{outer@}Q]\!], \textit{owner}(Q)) \\
\textit{is-refed}(v) = \exists u \in \textit{refed-vars}(), v = u \vee \exists R, v = \textit{extra-m}(R, u)
\end{array}
$$

Figure 5.22: Lambda Lift Rewriting Functions Implementation (3)

5.7 Explicit Outer

This section gives a formal description of the explicit outer transformation described in Chapter 4.

5.7.1 Introduction

The explicit outer transformation for Core code is a bit different from the one for Scala code described in Chapter 4. This comes from the fact that there are no type parameters in the Core language. In Scala, a lifted inner class must be augmented with an extra type parameter for each type parameter of its enclosing classes. In the Core language, type parameters are encoded by type fields and those can be extracted from any instance of their class. Therefore, inner classes do not need extra type parameters as they can access the type fields corresponding to the type parameter

of their enclosing classes through their explicit outer field. This implies that encoding Scala code into Core code and then apply the explicit outer transformation does not yield the same result as first applying the transformation and then perform the encoding. It implies also that all inner classes are always augmented with an explicit outer type field and an explicit outer value field and nothing else. To illustrate this difference, we consider the Scala code below.

```
class A[X]() {
  class B[Y](_x: X, _y: Y) { val x: X = _x; val y: Y = _y; }
}
val a: A[Int] = new A[Int]();
val b: a.B[Int] = new a.B[Int]();
```

The application to this code of the explicit outer transformation described in Chapter 4 yields the following code:

```
class A[X]() {}
class B[Outer$B <: A, X$A, Y](_outer$b: Outer$B, _x: X$A, _y: Y) {
  val outer$b: Outer$B = _outer$b;
  val x: X$A = _x;
  val y: Y = _y;
}
val a: A[Int] = new A[Int]();
val b: B[a.type, Int, Int] = new B[a.type, Int, Int](a);
```

The Core version of the untransformed Scala code is the following one where the functions a and b correspond to the constructors of the classes A and B.

```
def a(that: A { override Outer@A >: outer@a.type
                                  <: Root; }): Unit = {};
class A {
  type X;
  def b(that: B { override Outer@B >: outer@b.type <: A; },
        _x: outer@b.X, _y: that.Y): Unit =
    { that.x = _x; that.y = _y; }
  class B { type Y; val x: outer@B.X; val y: Y; }
}
val a: A { override Outer@A = Root.this; override X = Int; } = {
  val v: A { override Outer@A = Root.this; override X = Int; } =
    new A { override Outer@A = Root.this; override X = Int; };
  Root.this.a(v);
  v
}
```

```
val b: B { override Outer@B = a.type; override Y = Int; } = {
  val w: B { override Outer@B = a.type; override Y = Int; } =
    new B { override Outer@B = a.type; override Y = Int; };
  a.b(w);
  w
}
```

The explicit outer transformation lifts the class `B` and its constructor `b` to the top-level. It adds the explicit outer type and value fields `Outer$B` and `outer$B` to the class `B` and the explicit outer type and value parameters `Outer$b` and `outer$b` to the constructor `b`. Furthermore, all references to the implicit outer fields of `B` and `b` are replaced with references to these extra variables. The resulting code is given below. The definitions of the class `A`, its constructor `a` and the variable `a`, which remain unchanged, have been omitted.

```
def b[Outer$b <: A](outer$b: Outer$b,
      that: B { override Outer$B >: outer$b.type <: A; },
      _x: outer$b.X, _y: that.Y): Unit =
  { that.outer$B = outer$b; that.x = _x; that.y = _y; }
class B {
  type Outer$B <: A;
  val outer$B: Outer$B;
  type Y;
  val x: outer$B.X;
  val y: Y;
}
val b: B { override Outer$B = a.type; override Y = Int; } = {
  val w: B { override Outer$B = a.type; override Y = Int; } =
    new B { override Outer$B = a.type; override Y = Int; };
  b[a.type](a, w);
  w
}
```

The lifted class `B` does not have an extra type field for the type parameter `X` of the class `A`. Indeed, the type field corresponding to this type parameter can be accessed through the explicit outer field `outer$B` in the class `B` and through the explicit outer parameter `outer$b` in the constructor `b`.

References to the unique instance of the root context are treated in a special way to avoid the generation of implicit outer fields on plain fields. To illustrate this, we consider the following example:

```
val c: C;
```

```
class C {
  type T;
  class D1 {
    def f(): outer@f.outer@D1.outer@C.c.T = outer@f.f();
  }
  class D2 extends D1 {
    def g(): outer@g.outer@D2.outer@C.c.T = outer@g.f();
  }
}
```

If the class D1 is lifted to the top-level without handling in a special way its reference **outer**@f.**outer**@D1.**outer**@C to the unique instance of the root context, the following code is obtained:

```
class D1 {
  type Outer$D1 <: C;
  val outer$D1: Outer$D1;
  def f(): outer@f.outer$D1.outer@C.c.T = outer@f.f();
}
```

In this code, the implicit outer field **outer**@C is selected on the plain field outer$D1. This is not welcome because the unique instance of the root context is not a real value and the implicit outer field **outer**@C is not a real field. Both do not exist at runtime. In this example, the variable c corresponds to a static variable. Therefore, the implicit outer field **outer**@C should only appear in expressions corresponding to encodings of the expression **Root.this**. Or, in other words, it should only by selected on qualifiers that are constituted only of other implicit outer fields. For this reasons, references to the unique instance of the root context are handled in a special way. For example, the type of the method f is shortened to the type **outer**@f.**outer**@D1.c.T where all implicit outer fields are indeed selected on other implicit outer fields or the current instance. Thus, the lifted definition of the class D1 is the following one:

```
class D1 {
  type Outer$D1 <: C;
  val outer$D1: Outer$D1;
  def f(): outer@f.outer@D1.c.T = outer@f.f();
}
```

The special handling of references to the unique instance of the root context imposes that subclasses of inner classes and record types of inner classes are also handled in a special way. The reason for this can be observed in the lifted version of the class D2 given below.

```
class D2 extends D1 {
```

eo-$\mathcal{P}(\mathcal{P})$	Rewrites the program $\mathcal{P}$.
eo-$\mathcal{D}(\mathcal{D})$	Rewrites the definition $\mathcal{D}$.
eo-e(P, e)	Rewrites the expression e in the context of the record P.
is-lifted(R)	Returns whether the record R is lifted.
extra-$\overline{\mathcal{D}}(R)$	Returns the definitions of the explicit outer type and value fields of the lifted record R. This function is used by *eo-*$\mathcal{P}$.
extra-V(R)	Returns the symbol of the explicit outer type of the lifted record R.
extra-v(R)	Returns the symbol of the explicit outer field of the lifted record R.
extra-$\overline{O}(P, l, \overline{R})$	Returns the new overrides of the implicit outer type of the lifted records among $\overline{R}$ in the context of the record P and the nesting level l. This function is used in the rewriting of class definitions in *eo-*$\mathcal{D}$ and record types in *eo-e*.
path-to-root(P, l)	Returns the path to the unique instance of **`Root`** in the context of the record P and the nesting level l.

Figure 5.23: Explicit Outer Rewriting Functions

```
  type Outer$D2 <: C;
  val outer$D2: Outer$D2;
  def g(): outer@g.outer$D2.c.T = outer@g.f();
}
```

In the lifted class, the body of the method `g` does not longer conform to its return type. Indeed, the type **`this.outer@g.outer@`**`D1.c.t` of its body is not a subtype of its return type **`this.outer@g.outer@`**`D2.c.t` because it is impossible to prove that the expressions **`this.outer@g.outer@`**`D1` and **`this.outer@g.outer@`**`D2` denote the same value. To enable this, the class `D2` must be augmented with the following refinement:

```
override Outer@D1 = outer@D2.type;
```

This refinement states that the enclosing instance of the superclass `D1` is the same as its own one. Similarly, all record types C `{ ... }` of an inner class C must be augmented with a similar refinement like this:

C `{` **`override Outer@`**C `=` **`Root.this.type;`** `... }`

5.7.2 Implementation

The specification of the functions implementing the explicit outer transformation is given in Figure 5.23 and their implementation in Figure 5.24 and Figure 5.25.

The function *eo-*$\mathcal{P}$ transforms each definition of the program and appends the definition of the explicit outer type and value fields added to each inner class and the definition of the explicit outer type and value parameters added to each constructor of all inner classes.

$$
\begin{array}{l}
\textit{eo-}\mathcal{P}(\mathcal{P}) = \textbf{case } \mathcal{P} \textbf{ of } \overline{\mathcal{D}} \Rightarrow \textit{eo-}\mathcal{D}(\overline{\mathcal{D}}), \textit{extra-}\overline{\mathcal{D}}(\{R \mid \textit{is-lifted}(R)\}) \\
\textit{eo-}\mathcal{D}(\mathcal{D}) = \textbf{let } \mathcal{D}' = \textit{map-}\mathcal{D}(\textit{eo-e}, \mathcal{D}) \textbf{ in case } \mathcal{D}' \textbf{ of} \\
\quad [\![P \texttt{ defines class } C \texttt{ extends } \overline{C}\ \{\overline{O}\}]\!] \Rightarrow \\
\qquad [\![\texttt{Root defines class } C \texttt{ extends } \overline{C}\ \{\overline{O}, \textit{extra-}\overline{O}(C, 0, \overline{C})\}]\!] \\
\quad [\![P \texttt{ defines def } f\{\overline{f}\}[\overline{V}](\overline{v})\texttt{:}\ X = x]\!] \textbf{ if } \textit{is-lifted}(f) \Rightarrow \\
\qquad [\![\texttt{Root defines def } f\{\overline{f}\}[V, \overline{V}](v, \overline{v})\texttt{:}\ X = x']\!] \textbf{ where} \\
\qquad\quad V, v = \textit{extra-V}(f), \textit{extra-v}(f) \\
\qquad\quad x' = \textit{is-primary-constructor}(f)\ ?\ [\![y\texttt{;}x]\!]\ :\ x \\
\qquad\quad y = [\![\texttt{this}(0).w.v' = \texttt{this}(0).v]\!] \\
\qquad\quad w, v' = \textit{self-value}(f), \textit{extra-v}(\textit{initialized-class}(f)) \\
\quad \textbf{otherwise} \Rightarrow \mathcal{D}' \\
\textit{eo-e}(P, e) = \textit{aux}(0, e) \textbf{ where} \\
\quad \textit{aux}(l, e) = \textbf{let } e' = \textit{map-e}(\textit{aux}, l, e) \textbf{ in case } e' \textbf{ of} \\
\qquad [\![p\texttt{.outer@}R]\!] \textbf{ if } \textit{owner}(R) = [\![\texttt{Root}]\!] \Rightarrow \textit{path-to-root}(P, l) \\
\qquad [\![p\texttt{.outer@}R]\!] \textbf{ if } \textit{is-lifted}(R) \Rightarrow [\![p\texttt{.}v]\!] \textbf{ where } v = \textit{extra-v}(R) \\
\qquad [\![p\texttt{.Outer@}R]\!] \textbf{ if } \textit{is-lifted}(R) \Rightarrow [\![p\texttt{.}V]\!] \textbf{ where } V = \textit{extra-V}(R) \\
\qquad [\![\texttt{override Outer@}R\ S]\!] \textbf{ if } \textit{is-lifted}(R) \Rightarrow \\
\qquad\quad [\![\texttt{override } V\ S]\!] \textbf{ where } V = \textit{extra-V}(R) \\
\qquad [\![p\texttt{.}f\texttt{[}\overline{X}\texttt{]}(\overline{p})]\!] \textbf{ if } \textit{is-lifted}(f) \Rightarrow \\
\qquad\quad [\![q\texttt{.}f\texttt{[}p\texttt{.type}, \overline{X}\texttt{]}(p, \overline{p})]\!] \textbf{ where } q = \textit{path-to-root}(P, l) \\
\qquad [\![\overline{R}\ \{\overline{O}\}]\!] \Rightarrow [\![\overline{R}\ \{\overline{O}, \textit{extra-}\overline{O}(P, l+1, \overline{R})\}]\!] \\
\qquad \textbf{otherwise} \Rightarrow e'
\end{array}
$$

Figure 5.24: Explicit Outer Rewriting Functions Implementation (1)

The first case of the function *eo-*$\mathcal{D}$ handles classes. It adds a refinement of the outer type field of each lifted superclass. The second case handles constructors of inner classes. It adds the explicit type and value parameters to the parameter lists. And, if the constructor is a primary one, it adds to the body of the constructor an initialization of the explicit outer field of

the new instance.

The first case of the function *eo-e* replaces references to the unique instance of the root context with direct references to this instance including only implicit outer field selections. The second, third and fourth cases replace implicit outer fields with the corresponding explicit ones. The fifth case handles calls to constructors. Finally, the sixth case inserts in record types a refinement of the implicit outer fields of each of its lifted classes.

$$
\begin{array}{l}
\mathit{is\text{-}lifted}(R) = \textbf{case } R \textbf{ of} \\
\quad C \Rightarrow \mathit{owner}(C) \neq [\![\texttt{Root}]\!] \\
\quad f \Rightarrow \mathit{owner}(f) \neq [\![\texttt{Root}]\!] \wedge \mathit{is\text{-}constructor}(f) \\
\quad \_ \Rightarrow \textbf{false} \\
\\
\mathit{extra\text{-}}\overline{\mathcal{D}}(R) = [\![R \textbf{ defines } \mathcal{T}]\!], [\![R \textbf{ defines } \mathcal{V}]\!] \textbf{ where} \\
\quad \mathcal{T}, \mathcal{V} = [\![\texttt{type } V \texttt{>: bottom <: } P\ \texttt{\{\}}]\!], [\![\texttt{val } v\texttt{: this(0).}V]\!] \\
\quad V, v = \mathit{extra\text{-}V}(R), \mathit{extra\text{-}v}(R) \\
\quad P \quad = \mathit{owner}(R) \\
\\
\mathit{extra\text{-}V}(R) = \mathit{is\text{-}lifted}(R) \text{ ? the extra type field of } R \text{ : } \textbf{undefined} \\
\mathit{extra\text{-}v}(R) = \mathit{is\text{-}lifted}(R) \text{ ? the extra value field of } R \text{ : } \textbf{undefined} \\
\\
\mathit{extra\text{-}}\overline{O}(P, l, \overline{R}) = [\![\textbf{override } \texttt{Outer@}\overline{R}' \texttt{>: } X \texttt{ <: } X]\!] \textbf{ where} \\
\quad \overline{R}' = \{R \mid R \in \overline{R} \wedge \mathit{is\text{-}lifted}(R)\} \\
\quad X \ = [\![\mathit{path\text{-}to\text{-}root}(P, l)\texttt{.type}]\!] \\
\\
\mathit{path\text{-}to\text{-}root}(P, l) = \mathit{aux}([\![\texttt{this(}l\texttt{)}]\!], P) \textbf{ where} \\
\quad \mathit{aux}(p, Q) = \textbf{if } Q = [\![\texttt{Root}]\!] \textbf{ then } p \textbf{ else} \\
\qquad \mathit{aux}([\![p\texttt{.outer@}Q]\!], \mathit{is\text{-}lifted}(Q) \text{ ? } [\![\texttt{Root}]\!] \text{ : } \mathit{owner}(Q))
\end{array}
$$

Figure 5.25: Explicit Outer Rewriting Functions Implementation (2)

Chapter 6

Scaletta

This chapter describes Scaletta, a calculus that is focused on the interplay between inner classes and virtual types and which was a great source of inspiration for the design of the Core language.

6.1 Introduction

The combination of inner classes and virtual types can be observed in several object-oriented programming languages like BETA [16] and Scala [20]. Both concepts, inner classes and virtual types, have been studied and well-understood separately [14, 13] but little has been done on the formalization of their interaction. Scaletta is a calculus of classes and objects whose goal is to study this interaction and more specifically to type virtual types in the presence of inner classes.

Scaletta consists of a very limited number of constructs. For example, the calculus has no notion of mutable fields. It is limited to a functional fragment where objects have no state and no identity like in [12]. The calculus also has neither methods nor class constructors. Instead it has a more general concept of abstract inheritance that lets a class extend an arbitrary object. This choice greatly reduces the number of evaluation rules but poses also some specific problems. Finally, thanks to an interpretation of values as types the calculus also unifies type fields and value fields.

Scaletta was developed in collaboration with Vincent Cremet and was described in a technical report [2]. This chapter is an adapted and slightly extended version of that report.

Section 6.2 explains informally that typing virtual types in the presence of inner classes requires a kind of alias analysis. Section 6.3 introduces the untyped fragment of Scaletta. Section 6.4 gives an interpretation of values

as types that lets us simulate type fields with normal value fields and introduces a type system based on the concept of abstract evaluation. We illustrate by an example how this type system performs the required alias analysis identified in Section 6.2. Section 6.5 extends the type system to address the problem of typing abstract inheritance. Section 6.6 describes how higher-level constructs can be encoded in the typed calculus. Section 6.7 discusses the undecidability of the type system and introduces the concept of typing strategies. Section 6.8 compares Scaletta to the Core language. Finally, Section 6.9 reviews the related works.

All examples presented in this chapter have been typed-checked and evaluated using our Scaletta compiler. Both the complete versions of the examples and the compiler can be found on the Scaletta home page [1].

6.2 Typing Virtual Types with Inner Classes

Typing virtual types in the presence of inner classes requires some kind of alias analysis. This is illustrated by the following Scala code.

```
abstract class A {
  type T <: Object;
  abstract class X {
    val x: A.this.T;
  }
}
class B extends A {
  type T = String;
  class Y extends X {
    val x = "foo";
  }
}
```

This code and in particular the assignment of the field `x` in the class `Y` is well-typed. The field `x` is declared in the class `X` with the type `T` and `T` is a virtual type of the class `A`. The exact value of the type `T` is not known, at least not in the class `X`. So, how is it possible to assign `"foo"` to `x` in one of its subclasses? The assignment is legal because it is possible to establish that for any instance of the class `Y`, its enclosing instance and the enclosing instance of its superclass `X` are the same. Thus, the enclosing instance of the superclass `X` is not only an instance of the class `A` but also an instance of the class `B`, which assigns the type `String` to the virtual type `T`. The field `x` can therefore be assigned with values of type `String`.

This example illustrates the fact that typing virtual types in the presence of inner classes requires some kind of alias analysis on enclosing instances. Indeed, in this example, it is possible to establish that in the class `Y` the type `T` of the field `x` is equal to `String` only if it is possible to establish that for any instance of the class `Y` its enclosing instance and the enclosing instance of its superclass are the same. Otherwise, it would only be possible to establish that the type `T` is upper bounded by `Object`.

Parameterized classes have been presented as an alternative to virtual types [5]. For instance, the code above can also be written with type parameters instead of virtual types. The idea of the translation is to replace each virtual type of a class with a type parameter of that class. The code below and in particular the assignment of the field `x` in the class `Y` is well-typed. This is true for exactly the same reasons as for the code above. The well-formedness proof requires the same alias analysis to determine that the enclosing instance of the class Y and the one of its superclass are the same and that the type `T` occurring in the class `X` is therefore equal to the type `String` in class `Y`.

```
abstract class A[T] {
  abstract class X {
    val x: T;
  }
}
class B extends A[String] {
  class Y extends X {
    val x = "foo";
  }
}
```

This observation shows that Scaletta whose goal is to type virtual types in the presence of inner classes can also be used to understand the interaction between inner classes and parameterized classes in languages like Scala or Java 5.

6.3 Untyped Calculus

This section describes the untyped fragment of Scaletta and explains its semantics. The full syntax is summed up in Figure 6.1 and the semantics is given in Figure 6.2.

6.3.1 Syntax

A Scaletta program consists of a list of class declarations and a main expression. Each class has a name, zero or one parent and a list of field valuations. Within a program, classes are referred to through their name, therefore all classes must have a globally unique name.

Class name	B, C	$=$	$\ldots$	
Field name	f	$=$	$\ldots$	
Class declaration	D	$=$	**`class`** C **`extends`** p `{` $\overline{d}$ `}`	
Class parent	p	$=$	$t \mid$ **`nothing`**	
Class member	d	$=$	**`field`** f `=` t	field valuation
Term	t, u	$=$	**`this`**	current instance
		$\mid$	t`!`C	instance creation
		$\mid$	t`.`f	field selection
		$\mid$	t`@`C	outer field selection
Program	P	$=$	$\overline{D}\ t$	
Evaluation context	E	$=$	$\langle\rangle \mid E$`!`$C \mid E$`.`$f \mid E$`@`C	
Values	v	$=$	**`this`** $\mid v$`!`C	

Figure 6.1: Scaletta Syntax

All classes are defined at the top-level but all are nonetheless inner classes. Indeed, each class has an implicit outer field and an enclosing instance has to be provided to instantiate them. To bootstrap the whole, there is an implicit root class `Root`, which may never be explicitly instantiated and whose unique instance is provided from the outside.

Inheritance is implemented through delegation. This means that any instance c of a class C with inherited members contains a value that implements those members. This value is called the *delegate* of c. Each time the implementation of a member is requested on c, it is searched in the class C. If it is not found, the request is forwarded to the delegate of c. The parent of a class C is a term t, which is used to compute the delegate of new instances of the class C. This term is evaluated in the context of the enclosing class of C; the expression **`this`** denotes the enclosing instance of the class C in the term t.

Terms are of four different kinds. The traditional **`this`** denotes the current instance. The field selection t`.`f denotes the evaluation of the field f on the term t. The instance creation t`!`C corresponds to the Scala expression **`new`**t`.`C`()`. It creates a new instance of the class C with the enclosing instance t. In other words, it creates a new instance of the class C whose

implicit outer field is initialized with t. The outer field selection t`@`C corresponds to the Scala expression t`.outer$`C where `outer$`C denotes the implicit outer field of the class C. In some sense, the operation t`@`C is the opposite of $t\,!\,C$; it extracts from an instance of the class C the enclosing instance that was used to create it. One could expect that for any term t and any class C, the expression $t\,!\,C$`@`C is equal to t. Later in this section we will see that this is indeed true.

6.3.2 Semantics

The semantics of Scaletta is defined by three inductive relations and two auxiliary relations. All are implicitly parameterized by the list of class declarations of the program; when a class declaration D appears in the premises of a rule, it simply means that D belongs to this list.

Auxiliary Relations

$$\frac{\texttt{class}\ C\ \texttt{extends}\ p\ \{\,\overline{d}\,\}}{C\ \textit{extends}\ p} \qquad \frac{\texttt{class}\ C\ \texttt{extends}\ p\ \{\,\overline{d}\,\}}{C\ \textit{declares}\ d_i}$$

Reduction $\boxed{t \to u}$

$$(\to\text{-FIELD})\quad \frac{t < C \qquad C\ \textit{declares}\ (\texttt{field}\ f = u)}{t.f \to [t/\texttt{this}]u}$$

$$(\to\text{-OUTER})\quad \frac{t \prec u\,!\,C}{t\texttt{@}C \to u} \qquad (\to\text{-CONTEXT})\quad \frac{t \to u}{E\langle t\rangle \to E\langle u\rangle}$$

Expansion $\boxed{t \prec u}$

$$(\prec\text{-REFL})\quad \frac{}{t \to t} \qquad (\prec\text{-TRANS})\quad \frac{t \to t' \qquad t' \to t''}{t \to t''} \qquad (\prec\text{-RED})\quad \frac{t \to u}{t \prec u}$$

$$(\prec\text{-EXT})\quad \frac{t < C \qquad C\ \textit{extends}\ u}{t \prec [t\texttt{@}C/\texttt{this}]u}$$

Inheritance $\boxed{t < C}$

$$(<\text{-NEW})\quad \frac{t \prec u\,!\,C}{t < C}$$

Figure 6.2: Scaletta Semantics

Among the three semantics relations, there is a reduction relation $t \to u$, which states that t reduces to u. The two others are the expansion rela-

tion $t \prec u$, which states that u is a delegate of t, and the inheritance relation $t < C$, which states that t is a (direct or indirect) instance of class C.

Values are terms that consist only of instance creations. The initial **`this`** of values and also of all successive terms produces during a reduction denotes the unique instance of the implicit root class.

The expansion relation extends the notion of delegate of a term t to include not only the direct delegate of new instances ($\prec$-EXT), but also the term t itself ($\prec$-REFL), all delegates of any of its delegate ($\prec$-TRANS) and the reduction of any of its delegate ($\prec$-RED). The outer field selection t@C in rule ($\prec$-EXT) expresses the fact that the parent term t of a class C has to be evaluated in the context of the enclosing class of C (i.e. in the parent term, **`this`** denotes the current enclosing instance of the class C). We use the notation $[t/\texttt{this}]u$ to represent the term obtained by substituting t for **`this`** in u.

The inheritance relation consists of a single rule ($<$-NEW), which states that a term t is an instance of a class C if one of its delegates is a (direct) new instance of the class C. Strictly speaking, this relation is not really necessary here; its single rule could easily be inlined in the two other relations. We introduce it in anticipation of the typed calculus, which adds two other deduction rules.

The reduction relation imposes no order on evaluation, as expressed by the deduction rule ($\rightarrow$-CONTEXT). In this rule the notation $E\langle t\rangle$ represents the term obtained from the context E by replacing the hole $\langle\rangle$ with the term t. Field selections are reduced the same way as parameterless methods are evaluated in standard object-oriented languages ($\rightarrow$-FIELD) by a lookup of the field starting from the receiver object. As Scaletta is purely functional, this behavior is indistinguishable from one where fields are evaluated only once and cached. The rule ($\rightarrow$-OUTER) expresses the fact that the outer fields of a value are split over its delegates; each delegate stores the value of one outer field, namely the one of the class it is a direct instance of. It expresses also the fact that given the exact class of a value t and the value of the outer field of that class, the value of all other outer fields can be computed simply by recomputing the delegates of t.

6.3.3 Examples

First, we prove that, as stated earlier, for any term t and any class C, the term $t\,!\,C@C$ reduces to t. Here is the proof:

$$\dfrac{\dfrac{}{t\,!\,C \prec t\,!\,C}\ (\prec\text{-Refl})}{t\,!\,C@C \to t}\ (\to\text{-Outer})$$

Now, let us try to encode positive integers with a base class `Int` and two subclasses `Zero` and `Succ`. The idea is to encode 0 with the value **`this`**`!Zero` and any strictly positive integer n with an instance of class `Succ` whose enclosing instance stores the predecessor of n. Thus, any positive integer n is represented by the value **`this`**`!Zero` followed by n instance creations of class `Succ`. For example, the number 2 is represented by the value **`this`**`!Zero!Succ!Succ`. For now, we equip our integers with only two fields: `pred` and `succ`, which return respectively the predecessor and the successor of the receiver integer.

```
class Int extends nothing {
  field succ = this!Succ;
  class Succ extends this@Int!Int {
    field pred = this@Succ;
  }
}
class Zero extends this!Int {
  field pred = this;
}
```

Note that although in the calculus all classes are declared at the same level, in our code examples we nest classes that are logically nested. For example, in the code above, the class `Succ` is nested in class `Int` because we expect that the enclosing instance of any instance of class `Succ` will always be an instance of class `Int`.

Now, let us try to reduce the term **`this`**`!Zero.succ.pred`. We start by proving that **`this`**`!Zero` is an instance of class `Int`:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\texttt{this!Zero} \prec \texttt{this!Zero}}\ (\prec\text{-Refl})}{\texttt{this!Zero} < \texttt{Zero}}\ (<\text{-New}) \qquad \texttt{Zero } \textit{extends } \texttt{this!Int}}{\texttt{this!Zero} \prec \texttt{this!Zero@Zero!Int}}\ (\prec\text{-Ext})}{\texttt{this!Zero} < \texttt{Int}}\ (\to\text{-Outer})$$

As class `Int` declares that field `succ` is equal to **`this`**`!Succ`, we may apply the rule ($\to$-Field) to reduce **`this`**`!Zero.succ` to **`this`**`!Zero!Succ`.

This is an instance of class Succ, which declares that the field pred is equal to **this**@Succ. We may therefore again apply the rule (→-FIELD) to reduce **this**!Zero!Succ.pred to **this**!Zero!Succ@Succ, which can be reduced to **this**!Zero by applying our previous result about terms of the form t!C@C. Altogether, with some additional applications of the rule (→-CONTEXT), this lets us conclude that **this**!Zero.succ.pred reduces to **this**!Zero.

6.3.4 Syntactic Sugar

In order to make our next examples a bit easier to read, we introduce some syntactic sugar. First of all, we will omit the extends clause of classes that extend nothing. We will also replace sequences of outer field selections that designate the current instance of an enclosing class C with the Java syntax C**.this**. Finally, we make the initial **this** of a term optional.

Here is the integer example rewritten with the newly introduced syntactic sugar.

```
class Int {                          // "extends nothing" -> ""
  field succ = !Succ;                // "this"      -> ""
  class Succ extends Root.this!Int { // "this@Int"  -> "Root.this"
    field pred = Int.this;           // "this@Succ" -> "Int.this"
  }
}
class Zero extends !Int {            // "this"      -> ""
  field pred = this;
}
```

6.3.5 Methods

Methods and method calls are not part of the calculus but they can be encoded using classes. To illustrate this, we augment our integers with a method add that computes the sum of the receiver object and its parameter. Its implementation is trivial in the class Zero; the method simply returns its parameter. It is a bit more complex in the class Succ; it involves a method call, namely a recursive call to itself.

```
class Int {
  field succ = !Succ;
  class Succ extends Root.this!Int {
    field pred = Int.this;
    method add(that) = pred.add(that.succ);
  }
```

```
}
class Zero extends !Int {
  field pred = this;
  method add(that) = that;
}
```

The definition of a method m is encoded by an auxiliary class M and a field m, which is initialized with an instance of the class M. The class M has one abstract field for each parameter of m and a result field `result` whose value is the encoding of the body of the method. This encoding replaces all the references to parameters of the method m by references to the corresponding fields of the class M. The values of the parameter fields are provided when the method is called.

The method `add` in the class `Zero` is desugared into the two following definitions.

```
class AddInZero { field result = AddInZero.this.that; }
field add = !AddInZero;
```

Note that in the untyped calculus abstract fields are not declared at all. This explains the occurrence of the undeclared field `that` in the class `AddInZero`. In the class `Succ`, the method `add` is desugared into the following definitions.

```
class AddInSucc {
  field result = pred.add(AddInSucc.this.that.succ);
}
field add = !AddInSucc;
```

A method call $t.m(\overline{u})$ is encoded by the term **this**$!N.r$ where N is an auxiliary class and r the name of the result field of the method m. The class N extends the term $t.m$ and contains a field valuation for each argument u_i.

We can now rewrite the class `AddInSucc` without syntactic sugar for its method call:

```
class AddInSucc {
  class Apply extends pred.add {
    field that = AddInSucc.this.that.succ;
  }
  field result = !Apply.result;
}
```

All instances of the class `Succ` have the same single delegate **this**`!Int`. This is not true for the class `Apply` for which the value of the parent varies from one instance to the other. For example it can be shown that the sole

delegate of `1!AddInSucc!Apply` is `0!AddInZero` while the sole delegate of `2!AddInSucc!Apply` is `1!AddInSucc`. Such a behavior is only possible because classes extend arbitrary terms instead of classes. We call this mechanism *abstract inheritance*. Abstract inheritance is the key element that lets us encode methods.

6.3.6 Blocks

As a second demonstration of the simplicity for encoding classical high-level programming constructs in Scaletta, we show now how we can encode blocks. We introduce the syntactic sugar $\{\overline{D};\overline{d};t\}$ for expressing a block consisting of a list of class definitions $\overline{D}$, field definitions $\overline{d}$ and of a main expression t representing the value of the block. Such a block expression is translated into the class

class `C {` $\overline{D}$;$\overline{d}$; **field** `result =` t; `}`

and the expression **this**`!C.result` where the class name `C` and the field name `result` are both fresh.

6.3.7 Functions

Using inner classes, methods and blocks, it is possible to encode functions as objects containing a method `apply` and function applications as calls to this method. It follows that our calculus has first-class functions and therefore can trivially encode the lambda-calculus.

We formally define here the translation $\langle M \rangle$ from a term M of the lambda-calculus into an untyped Scaletta program $\overline{D}\,t$ consisting of a list of classes $\overline{D}$ and a main term t. For expressing the result of the translation, we use the already introduced syntactic sugar for methods and blocks, and we spatially nest classes that are logically nested.

(Variable) $\quad \dfrac{}{\langle x \rangle = x}$

(Application) $\quad \dfrac{\langle M \rangle = \overline{D}\ t \qquad \langle N \rangle = \overline{D}'\ u}{\langle M\ N \rangle = \overline{D}\overline{D}'\ t\texttt{.apply(}u\texttt{)}}$

(Abstraction) $\quad \dfrac{\langle M \rangle = \overline{D}\ t \qquad C \textit{ fresh class name}}{\langle \lambda x.M \rangle = \textbf{class}\ C\ \{\textbf{method}\ \texttt{apply(}x\texttt{) = \{}\overline{D}\texttt{;}t\texttt{\}\}}\ \textbf{this}\texttt{!}C}$

For instance the encoding of the term $\lambda f.\lambda x.fx$ is

```
class C1 {
  method apply(f) = {
```

```
    class C0 { method apply(x) = f.apply(x); }
    this!C0
  }
}
this!C1
```

6.3.8 Safety and Confluence

The untyped fragment of Scaletta is neither safe nor confluent. It is unsafe because the reduction of field and outer field selections may both be stuck. The reduction of a field selection $t.f$ is stuck if the value denoted by the term t has no valuation for the field f. The reduction of an outer field selection $t@C$ is stuck if the value denoted by the term t is not an instance of the class C. These two issues are addressed by Section 6.4, which augments the calculus with type annotations and presents a well-formedness predicate.

The calculus is not confluent for two reasons. Firstly, a value may inherit a field valuation for a given field f from several classes. If the field f is selected on such a value, the rule ($\rightarrow$-FIELD) does not specify which valuation should be used. Therefore, anyone can be used. Secondly, it is possible to build values that inherit several times from some class C but with different enclosing instances. If the outer field of the class C is selected on such a value, the rule ($\rightarrow$-OUTER) does not specify which enclosing instance should be used. So, here again, anyone can be used. Both problems could be solved by somehow ordering the inherited field valuations and enclosing instances and modifying the rules ($\rightarrow$-FIELD) and ($\rightarrow$-OUTER) to pick the first one. However, we do not do that because our well-formedness predicate described in Section 6.4 cannot tolerate programs where such things may arise. In Section 6.5 we describe the additional rules needed to reject such programs.

6.3.9 Alias Analysis

We terminate this section with an example showing that our calculus can indeed be used to do some alias analysis. We consider the Scaletta version of the example of Section 6.2. but without the virtual type `T` and the field `x`.

```
class A                 { class X                 {} }
class B extends this!A { class Y extends this!X {} }
```

We show here that with the given rules it is already possible to formally establish that, for a given instance of the class `Y`, its enclosing instance and

the enclosing instance of its superclass are the same value.

We do here the proof for the value **this**!B!Y. We have to show that **this**!B!Y@X and **this**!B!Y@Y denote the same value. The formal proof below demonstrates that the first term reduces to the second one.

$$\dfrac{\dfrac{\dfrac{}{\texttt{this!B!Y} \prec \texttt{this!B!Y}}(\prec\text{-Refl})}{\texttt{this!B!Y} < \texttt{Y}}(<\text{-New}) \qquad \texttt{Y } \textit{extends } \texttt{this!X}}{\texttt{this!B!Y} \prec \texttt{this!B!Y@Y!X}}(\prec\text{-Ext})}{\texttt{this!B!Y@X} \rightarrow \texttt{this!B!Y@Y}}(\rightarrow\text{-Outer})$$

6.4 Type System

In this section we introduce a type system for the untyped calculus presented in the previous section and show how it can be used to perform the alias analysis required to type the example described in Section 6.2.

6.4.1 Types

Typing a program written in an object-oriented language requires to approximate abstract fields. We follow the classical solution that consists in attaching a type to a field.

Abstract type fields and abstract value fields share the property of being *virtual*; their value depends on the exact class of the value from which they are selected. For value fields the computation of their value takes place at runtime and is called *late binding* or *polymorphism*. For type fields this computation happens at compile time. To keep our calculus as small as possible we want to factor the mechanism that governs the virtual aspects of value and type fields. To achieve this, we choose the most naive solution we can think of: using values as types.

$$\boxed{\text{Type } T \;=\; t}$$

However we have to find an interpretation of values as types. Here is this interpretation: a value t is of type u, where u is another value, if the value resulting from the evaluation of u is a delegate of the value resulting from the evaluation of t. It makes sense to approximate objects by their delegates because they inherit field valuations and enclosing instances from them.

The unification of types and values lets us simulate type fields with value fields, as illustrated later in the example of Section 6.4.5.

6.4.2 Annotations

To make type-checking feasible we add to a program a list of top-level annotations as presented in Figure 6.3.

Class annotation	A	$=$	**class** C **inside** B
Field annotation	a	$=$	**field** f**:** T **inside** B
Program with annotations		$=$	$\overline{A}\ \overline{a}\ P$

Figure 6.3: Scaletta Annotations

There are two kinds of annotations, one for classes and another for fields. A class annotation **class** C **inside** B constrains the enclosing instance of an instance of C to be an instance of B. In our code examples this requirement is implicitly expressed by nesting classes. Classes declared at the top-level are implicitly nested in a class `Root` where `Root` is a distinguished class name with no corresponding class declaration.

A field annotation **field** f**:** T **inside** B declares a bound T for the field f. The inside clause states that the field f may only be selected on instances of the class B. In our code examples this clause is implicitly expressed by nesting the field annotation in the class B.

6.4.3 Abstract Evaluation

In Section 6.2 we have seen that typing virtual types in the presence of inner classes requires to establish some equalities between enclosing instances. In the example of Section 6.3.9 we have established the equality between two enclosing instances of *a particular* instance of `Y` by evaluating them in our calculus. To apply this result to typing we have to establish that property *for all* instances of `Y`. A priori we cannot use the same technique because we cannot evaluate the two enclosing instances of all instances of `Y` and check that they denote the same value. We have to abstract over the particular instance of `Y`, but we want to keep the appealing principle of establishing equality by evaluation.

In conclusion we need to evaluate terms in a partially unknown context, i.e. a context where the exact class of the current instance is not known, but also where some fields are abstract. Such an *abstract evaluation* is easily obtained by adding a class context B to each evaluation rule. The class context specifies the class in which the terms have to be interpreted. In other words the class B denotes the class **this** is the current instance of.

The relations that constitutes abstract evaluation are summarized in Figure 6.4. They are implicitly parameterized by the lists of class declarations, class annotations and field annotations that constitute the program.

Abstract Reduction $\boxed{B \vdash t \to u}$

($\to^{abs}$-FIELD) $$\frac{B \vdash t < C \qquad C \textit{ declares } (\mathbf{field}\ f = u)}{B \vdash t.f \to [t/\mathbf{this}]u}$$

($\to^{abs}$-OUTER) $$\frac{B \vdash t \prec u\,!\,C}{B \vdash t@C \to u}$$

($\to^{abs}$-CONTEXT) $$\frac{B \vdash t \to u}{B \vdash E\langle t\rangle \to E\langle u\rangle}$$

Abstract Expansion $\boxed{B \vdash t \prec u}$

($\prec^{abs}$-REFL) $$\frac{}{B \vdash t \to t}$$

($\prec^{abs}$-TRANS) $$\frac{B \vdash t \to t' \qquad B \to t't''}{B \vdash t \to t''}$$

($\prec^{abs}$-RED) $$\frac{B \vdash t \to u}{B \vdash t \prec u}$$

($\prec^{abs}$-DEF) $$\frac{\mathbf{field}\ f\mathbf{:}\ u\ \mathbf{inside}\ C}{B \vdash t.f \prec [t/\mathbf{this}]u}$$

($\prec^{abs}$-EXT) $$\frac{B \vdash t < C \qquad C \textit{ extends } u}{B \vdash t \prec [t@C/\mathbf{this}]u}$$

Abstract Inheritance $\boxed{B \vdash t < C}$

($<^{abs}$-THIS) $$\frac{}{B \vdash \mathbf{this} < B}$$

($<^{abs}$-NEW) $$\frac{B \vdash t \prec u\,!\,C}{B \vdash t < C}$$

($<^{abs}$-OUTER) $$\frac{\mathbf{class}\ C\ \mathbf{inside}\ C'}{B \vdash t@C < C'}$$

Figure 6.4: Scaletta Abstract Evaluation

Compared to evaluation there are three new rules: one expansion rule and two instance rules. The first new instance rule ($<^{abs}$-THIS) tells us that in the class context B the current instance **this** is indeed an instance of B.

The second new instance rule ($<^{abs}$-OUTER) and the new expansion rule ($\prec^{abs}$-DEF) have similar roles. The former lets us approximate the value of the outer field of a class C to an instance of the enclosing class of C. The latter lets us approximate the value of a field to its declared bound. This rule can compensate the absence of a valuation for a field.

We want now to prove two claims; first that abstract evaluation is a generalization of evaluation and second that abstract evaluation can be used to compute in a partially unknown context.

The first claim is equivalent to the theorem below, whose proof is trivial, and the second claim is demonstrated in Section 6.4.5 where the value of a field in a partially unknown context is computed.

$$t \to u \Rightarrow \texttt{Root} \vdash t \to u$$

6.4.4 Well-formedness

Abstract evaluation is the core of our type system. We present now typing rules that make use of abstract evaluation to express the well-formedness of terms, class declarations, field declarations and programs. The rules are summarized in Figure 6.5. They are implicitly parameterized by the lists of class declarations, class annotations and field annotations that constitute the program.

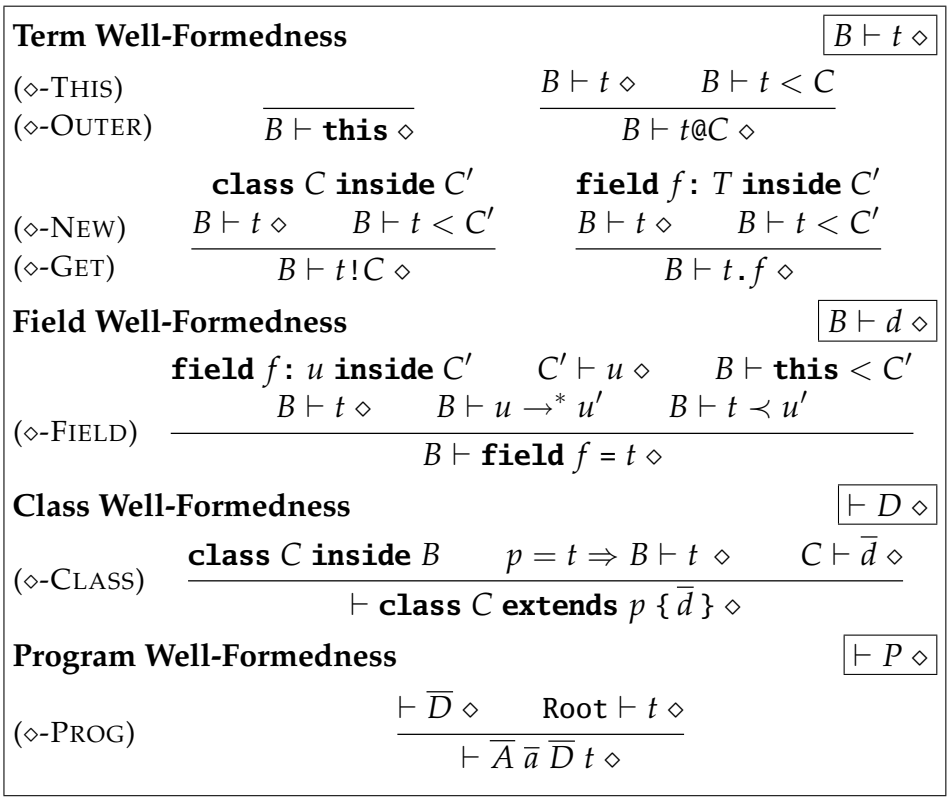

Figure 6.5: Scaletta Well-Formedness Relations

A particularity of this type system is that the relation that checks if a term is well-formed does not assign a type to this term. That is not necessary in our calculus because as types are terms, the more accurate type that can be given to a term is itself.

The well-formedness relation for terms is parameterized by a class B that has the same meaning as the one appearing in the abstract evaluation rules.

In the rule (◇-FIELD) for field valuations, the premise $B \vdash u \rightarrow^* u'$ contains the central idea that the bound u of a field f is reinterpreted in

the context of the class B containing the field valuation and that we get the more accurate value u'.

Note that in the rule ($\diamond$-CLASS) for classes the parent clause of the class must be interpreted in the context of the enclosing class.

The type system presented above is not sound. For instance, it does not ensure that a term t has no abstract fields when used as an enclosing instance, like in $t!C$, or as prefix of a selection, like in $t.f$ or $t@C$. This problem is solved in Section 6.5 where we address the problem of typing abstract inheritance. However it should be clear that this type system can already reject a lot of invalid programs, namely those that select fields or outer fields on terms that do not even inherit a declaration for that fields. Our problem is that we accept programs that select fields and outer fields on terms that declare them but not implement them.

6.4.5 Alias Analysis

In Section 6.2 we have informally explained why the assignment of the field `x` in the class `Y` was valid in the given Scala code example. We give here a formal proof that the corresponding field valuation in a Scaletta version given below of that code is well-formed.

```
class A {
  field T: Root.this!Object;
  class X {
    field x: this@X.T;
  }
}
class B extends !A {
  field T = Root.this!String;
  class Y extends this!X {
    field x = "foo";
  }
}
```

We assume the presence of a top-level class `String` and that `"foo"` is an instance of that class.

To show that the field valuation is valid we have to apply the rule ($\diamond$-FIELD) to prove the following judgment:

$$\texttt{Y} \vdash \textbf{field}\ \texttt{x = "foo"}\ \diamond$$

This rule has several premises. However, the main difficulty is to prove the judgment given below. This judgment denotes the intuition that within

class Y it is known that T is equal to String. This is the most difficult part because it is this judgment that requires some alias analysis.

$$\texttt{Y} \vdash \textbf{this}\texttt{@X.T} \rightarrow^{*} \textbf{this}\texttt{@Y@B!String}$$

We prove first that in the context of class Y, **this**@X and **this**@Y denotes the same enclosing instance, i.e. that $\texttt{Y} \vdash \textbf{this}\texttt{@X} \rightarrow \textbf{this}\texttt{@Y}$.

$$\begin{array}{rc} (<^{abs}\text{-This}) & \dfrac{}{\texttt{Y} \vdash \textbf{this} < \texttt{Y}} \qquad \texttt{Y}\ \textit{extends}\ \textbf{this}\texttt{!X} \\ (\prec^{abs}\text{-Ext}) & \dfrac{}{\texttt{Y} \vdash \textbf{this} \prec \textbf{this}\texttt{@Y!X}} \\ (\rightarrow^{abs}\text{-Outer}) & \dfrac{}{\texttt{Y} \vdash \textbf{this}\texttt{@X} \rightarrow \textbf{this}\texttt{@Y}} \end{array}$$

Then we use this result to reduce the prefix of $\textbf{this}\texttt{@}X\texttt{.}T$.

$$(\rightarrow^{abs}\text{-Context})\ \frac{\texttt{Y} \vdash \textbf{this}\texttt{@X} \rightarrow \textbf{this}\texttt{@Y}}{\texttt{Y} \vdash \textbf{this}\texttt{@X.T} \rightarrow \textbf{this}\texttt{@Y.T}}$$

Finally we make use of the value of the field T in the class B.

$$\begin{array}{rc} (<^{abs}\text{-Outer}) & \dfrac{\textbf{class}\ \texttt{Y}\ \textbf{inside}\ \texttt{B}}{\texttt{Y} \vdash \textbf{this}\texttt{@Y} < \texttt{B}} \\ (\rightarrow^{abs}\text{-Field}) & \dfrac{\texttt{B}\ \textit{declares}\ (\textbf{field}\ \texttt{T} = \textbf{this}\texttt{@B!String})}{\texttt{Y} \vdash \textbf{this}\texttt{@Y.T} \rightarrow \textbf{this}\texttt{@Y@B!String}} \end{array}$$

6.5 Typing Abstract Inheritance

The choice of abstract inheritance, i.e. the possibility for a class to inherit from an arbitrary term instead of just a class, implied until here simplifications in the syntax, semantics and typing rules. However, with abstract inheritance it becomes far more challenging to detect the accidental overriding of a type field or an outer field than in a language where the class hierarchy is statically known, like Scala. In this section we illustrate by examples why it is not safe to allow overriding of type fields and outer fields.

The type system presented before does not take into account these two problems. One way of solving these problems is to add dynamic tests that check at runtime that an object does not define two values for a same field and that it has no two different enclosing objects corresponding to a same class. But if we want a safe statically typed calculus we need mechanisms to resolve these problems at compile time. So, in this section we also present our solutions to statically prevent these overridings.

6.5.1 Field Roles

A Scaletta field can conceptually play different roles depending on where it is used. We introduce some terminology for describing these roles: we call *template field* a field that is indented to be used in an extends clause, we call *type field* a field that is used in the bound of another field, and we call *value field* a field that is used as an object. Note that in Scaletta all fields can simultaneously play all these roles, to simplify the formalization we treat all the fields uniformly because they share the same characteristic of being potentially redefined in a subclass.

In real object-oriented languages, fields (also called members) have a statically determined role. In Scala there are type fields and value fields, but no template fields; a field is categorized to be a type or a value from the moment of its declaration through the use of different keywords.

In order to have a safe type system, we need to *forbid* the overriding of type fields, as shown by the example of the next section. And in order to have an expressive type system, we need to *allow* the type system to exploit the actual value of a type field. Our choice of treating uniformly the different field roles forces us to extend these constraints to all kinds of fields. For instance, it forces us to abandon the overriding of value fields, even if it is harmful, and it allows us to exploit the value of a value field at compile time, a thing that most compilers do not do.

6.5.2 Overriding of Type Fields

In the type system presented so far, the typing rule ($\rightarrow^{abs}$-FIELD), which approximates terms like $t.f$ by $[t/\mathbf{this}]u$ in presence of a field valuation $\mathbf{field}\ f\ =\ u$, implicitly assumes that $\mathbf{field}\ f\ =\ u$ is the only existing valuation for f inherited by t. Another valuation with a different value would introduce an inconsistency in the type system and would break type safety. But let us illustrate this problem with a "concrete" example.

The following Scala program is clearly unsafe because in class `C` the string `"foo"` held by the field `x` is added to the integer `3`.

```
abstract class A {
  type T;
  val x: T;
}
class B extends A {
  type T = String;
  val x = "foo";
}
```

```
class C extends B {
  type T = Int;
  val y: Int = x + 3;
}
```

However a naive type system would accept it because:

- in class B, the bound T of the field x resolves to String, so it is legal for x to hold the value "foo".
- similarly, in class C the bound T of the field x resolves to Int, so it is legal to consider x as an integer and add it to 3.

The problem comes from the fact that when type-checking class B, we make the implicit assumption that the value of T is equal to String in any instance of B, which is not always the case if we allow the overriding of this type field in the subclass C.

In languages with a static class hierarchy it is easy to prevent overriding of type fields, by checking that there is no path in this hierarchy that contains two valuations for the same field, as the path C→B→A in our example, which contains two valuations for the field T.

But let us have a look at the following Scaletta code, which is a variant of the previous example.

```
class A {
  field T;
  field x: T;
}
class B extends !A {
  field T = !String;
  field x = "foo";
}
field b: !A = !B;
class C extends b {
  field T = !Int;
  field y: !Int = x + 3;
}
```

Note that the only difference is that class C inherits now from an instance b of B instead of directly inheriting from B. For the purpose of our argumentation, we "hide" the exact value of b behind the bound !A.

Now, without extra mechanism the only way of detecting an overriding of the field T in class C is to use the information that b holds an instance of B. But what if this value is hidden, as in our example, or even worse if b is an abstract field?

```
field b: !A;
```

Then, clearly we need an additional mechanism to follow the absence or presence of a valuation for a field in a given term, in our example in the term `b` used as parent of the class `C`.

6.5.3 Holes

In Scaletta a term can be an instance of a class that has abstract fields. We say that such a term is *incomplete* and we call the fields that have no valuation in the term the *holes* of the term. We propose to add an annotation to field definitions to specify the holes of the value contained by this field. The key idea is that holes become the only fields that can be overridden. In our example we would write:

```
field b: !A misses {};
```

to specify that `b` is an instance of `A` with an empty set of holes. This way it becomes possible to reject the overriding of the field `T` in class `C`, because `T` is not a hole of `b`. More generally, a field definition

field f: t **misses** $\{f_1, \ldots, f_n\}$;

means that f holds a value that expands to t and for which valuations of fields $f_1, \ldots, f_n$ are missing.

Let us see an example where the hole annotation is not empty. In fact such annotations appear in the encoding of methods. In these case the holes will correspond to the method parameters. If the method has type parameters, they will correspond to type fields. As our previous discussion has shown, it is important to forbid the overriding of this kind of the fields.

For instance, encoding the following `add` method declaration in class `Int`

```
field add(that: !Int): !Int;
```

results in the following declarations[1]

```
class Int$add$def {
  field that: !Int;
  field result: !Int;
}
field add: !Int$add$def misses {that};
```

[1] We use the special character `$` as a normal character in the names of classes generated by the method encoding. It must not be confused with an operator of the calculus.

The last line means that the field `add` holds an instance of `Int$add$def` with a missing valuation for the field `that`. It allows a class extending `add` to provide a valuation for the field `that`. Such classes appear in the encoding of an application of the method. For instance, the method call

```
40.add(2)
```

is encoded as[2]

```
class Apply extends 40.add {
  field that = 2;
}
!Apply.result
```

In the encoding of an implementation for the method `add` we subclass the field `add` and provide a body, as for example in class `Zero`:

```
class Zero$add$val extends !Int$add$def {
  field result = that;
}
field add = !Zero$add$val
```

The above valuation for `add` is legal because `!Zero$add$val` conforms to the declared type of `add`: it is indeed an instance of `Int$Add$def` and its only missing field valuation is the one for `that`.

We conclude the informal presentation of holes with the special hole annotation `*`, which works as a wildcard for holes. Holes are mainly useful for value fields and template fields: an empty set of holes for a value field will ensure that the held value is complete and can consequently be used as receiver object, for template fields, holes allow to avoid overriding conflicts. As we want a uniform treatment of fields, type fields must also receive a hole annotation, but we cannot give an exhaustive list of holes for type fields. For instance, a type field `T` bounded by `!Object` could receive the value `!Int` in one subclass and the value `!List` in another. If classes `Int` and `List` are abstract, values `!Int` and `!List` have a priori unrelated holes. To handle this kind of cases where the set of holes is not predictable we use the annotation `*`, which matches an arbitrary set of holes:

```
field T: !Object misses {*};
```

[2]For clarity reasons, we use the numbers 40 and 2 to denote objects built from the classes `Zero` and `Succ`.

6.5.4 Formalizing Holes

To prevent multiple valuations for a field, we define a relation on terms that returns the list of fields for which the term inherits a definition but no valuation. We call these fields the *holes* of the term. The judgment

$$B \vdash \mathit{holes}(t) = H$$

expresses the fact that in the context of the class B the set of holes of term t is H.

With that relation it is easy to prevent multiple valuations simply by allowing a field valuation in a class only if the field is included in the holes of the class parent. In order to be able to define this relation on terms, we need to know for each field its list of holes. For this reason, we augment field types with a list of holes (see Figure 6.6). In addition to field names, this list may contain the symbol $*$, which means that the field can contain more holes than the ones explicitly given.

In addition to multiple valuations, the holes let us solve another problem related to field valuations, namely missing field valuations. Indeed, the typing rule ($\diamond$-GET) states that the term $t.f$ is well-formed if and only if t inherits the declaration of field f. This does not guarantee that t inherits a valuation for the field f. To prevent this, we use the holes information to forbid any selection on terms that have any hole. This for sure will prevent any selection of a field f on a value that inherits no valuation for f.

All the modified and newly introduced rules to handle holes are summarized in Figure 6.6. The accessors $\mathit{defs}(C)$ and $\mathit{vals}(C)$, used for instance in rule (HL-NEW), return respectively the set of field definitions and the set of field valuations directly contained in a given class C.

6.5.5 Overriding of Outer Fields

We have already explained the parallel between outer fields and normal fields, saying that outer fields were a special kind of field that come implicitly with the declaration of a class and whose value cannot make reference to the current instance of this class. Like type fields, outer fields are resolved at compile time, and like type fields they cannot suffer arbitrary overriding.

We start with a theoretical argument for this last claim: the typing rule ($\rightarrow^{abs}$-OUTER), which lets one reduce the term $t@C$ to the term u if t expands to a term of the form $u!C$, implicitly assumes that t does not expand to a term $u'!C$ where $u \neq u'$. If such other term u' exists, the danger is to choose one term at compile time and the other one at runtime. It would

Syntax (modifications only)

Types $T = t\ \texttt{misses}\ H$

Holes $H = \{\overline{l}\} \mid \{\overline{l}, *\}$

Hole Approximation $\boxed{B \vdash holes(t) = H}$

(HL-THIS) $$\frac{}{B \vdash holes(\texttt{this}) = \{\}}$$

(HL-RED) $$\frac{B \vdash t \to u \qquad B \vdash holes(u) = H}{B \vdash holes(t) = H}$$

(HL-NEW) $$\frac{\texttt{class}\ C\ \texttt{extends}\ p\ \{\,\overline{d}\,\} \qquad B \vdash holes([t/\texttt{this}]p) = H}{B \vdash holes(t\,!\,C) = (H \cup defs(C)) \backslash vals(C)}$$

(HL-GET) $$\frac{\texttt{field}\ f\texttt{:}\ t\ \texttt{misses}\ H\ \texttt{inside}\ C}{B \vdash holes(t\,.\,f) = H}$$

(HL-DROP) $$\frac{B \vdash holes(t) = \{\overline{f}, *\} \qquad \overline{f'} \subset \overline{f}}{B \vdash holes(t) = \{\overline{f'}, *\}}$$

(HL-OPEN) $$\frac{B \vdash holes(t) = \{\overline{f}\}}{B \vdash holes(t) = \{\overline{f}, *\}}$$

Term Well-Formedness (modifications only) $\boxed{B \vdash t\ \diamond}$

(◇-NEW) $$\frac{\cdots}{B \vdash holes(t) = \{\}}{B \vdash t\,!\,C\ \diamond}$$

(◇-GET) $$\frac{\cdots}{B \vdash holes(t) = \{\}}{B \vdash t\,.\,f\ \diamond}$$

(◇-OUTER) $$\frac{\cdots}{B \vdash holes(t) = \{\}}{B \vdash t@C\ \diamond}$$

Field Well-Formedness (modifications only) $\boxed{B \vdash d\ \diamond}$

(◇-FIELD) $$\frac{\begin{array}{c}\cdots \\ \texttt{field}\ f\texttt{:}\ u\ \texttt{misses}\ H\ \texttt{inside}\ A \\ B \vdash holes(t) = H \qquad \texttt{class}\ B\ \texttt{extends}\ p\ \{\,\overline{d}\,\} \\ B \vdash holes([\texttt{this@}B/\texttt{this}]p) = H' \qquad f \in H' \cup defs(B)\end{array}}{B \vdash \texttt{field}\ f = t\ \diamond}$$

Figure 6.6: Scaletta Hole Resolution

mean that the approximation made by the type-checker for $t@C$ was in fact wrong, which opens an obvious breach in the type safety.

We now illustrate the problem with a concrete example. The following Scaletta example shows that it is generally not safe to allow overriding of outer fields. An outer field is overridden if some object has two different enclosing instances corresponding to the same class. The following example is a variant of the one presented in Section 6.5.2. It suggests also how mixins can be encoded in Scaletta.

```
class M {
```

```
  field T: !Object;
  field s: !Object;
  class N extends s { field x: T; }
}
class M1 extends M { field T = !String; field s = !Object; }
class A extends !M1!N { field x = "foo"; }

class M2 extends M { field T = !Int; field s = !A; }
class B extends !M2!N { field y: !Int = x + 3; }
```

The program above is unsafe because in class `B` the string value `"foo"` held by the field `x` is added to the integer 3. However a naive type system would accept it because:

- in class `A`, the bound `this@N.T` of the field `x` resolves to `!String`, so it is legal for `x` to hold the value `"foo"`.
- similarly, in class `B` the bound `this@N.T` of the field `x` resolves to `!Int`, so it is legal to consider `x` as an integer and add it to 3.

The problem is that class `B` has two incompatible enclosing instances corresponding to the class `N`, namely `!M1` and `!M2`. More precisely we have the following expansion chain:

$$\texttt{!B} \prec \texttt{!M2!N} \prec \texttt{!M2.s} = \texttt{!A} \prec \texttt{!M1!N}$$

To break the inheritance cycle, we need a way to forbid class `N` to inherit an instance of itself. But the parent `s` of `N` is abstract, so we need to attach a field annotation to the declaration of `s` to specify that it cannot hold an instance of `N`. We call such a mechanism *class exclusion* and formalize it in the next section. The syntax for such annotations is:

```
field s: !Object excludes {N};
```

With this definition a field valuation for `s` may not contain an instance of `N`. The field valuation `field s = !A` in class `M2` becomes illegal because `!A` is clearly an instance of `N`. Thus, the unsafe program gets rejected as expected.

6.5.6 Formalizing Class Exclusion

The problem of multiple enclosing instances arises if a value inherits different instances of a same class. Therefore, to avoid the problem, we make it impossible for a value to inherit more than one instance of a given class.

To do that, we define a relation that associates to a term a list of classes of which the term inherits no instance. This relation is then used to make sure that the parent of a class does not already inherit an instance of that class. The definition of this class exclusion relation requires an additional annotation for field types, namely a list of classes of which the field is not an instance.

The judgment

$$B \vdash t \textit{ is-not-a } \overline{C}$$

expresses the fact that in the context of the class B the term t is not an instance of any of the classes $\overline{C}$. Note that for extensibility reasons the list of classes $\overline{C}$ is generally not exhaustive.

The Figure 6.7 summarizes the modified and the newly introduced rules to handle class exclusion.

Syntax (modifications only)
Types $T \;=\; t\ \texttt{misses}\ H\ \texttt{excludes}\ \overline{C}$

Class Exclusion Approximation $\boxed{B \vdash t \textit{ is-not-a } C}$

(Ex-Red) $$\frac{B \vdash t \to u \qquad B \vdash u \textit{ is-not-a } C}{B \vdash t \textit{ is-not-a } C}$$

(Ex-New) $$\frac{C \textit{ extends } p \qquad B \vdash [t/\texttt{this}]p \textit{ is-not-a } C' \qquad C' \neq C}{B \vdash t\,!\,C \textit{ is-not-a } C'}$$

(Ex-Get) $$\frac{\texttt{field}\ f : t\ \texttt{misses}\ H\ \texttt{excludes}\ \overline{C}\ \texttt{inside}\ C}{B \vdash t.f \textit{ is-not-a } C_i}$$

Field Well-Formedness (modifications only) $\boxed{B \vdash d \diamond}$

(◇-Field) $$\frac{\begin{array}{c}\ldots \\ \texttt{field}\ f : u\ \texttt{misses}\ H\ \texttt{excludes}\ \overline{C}\ \texttt{inside}\ C \\ \forall i.\ B \vdash t \textit{ is-not-a } C_i\end{array}}{B \vdash \texttt{field}\ f = t \diamond}$$

Class Well-Formedness (modifications only) $\boxed{\vdash D \diamond}$

(◇-Class) $$\frac{\begin{array}{c}\ldots \\ \texttt{class}\ C\ \texttt{inside}\ B \qquad p = t \Rightarrow B \vdash t \textit{ is-not-a } C\end{array}}{\vdash \texttt{class}\ C\ \texttt{extends}\ p\ \{\,\overline{d}\,\} \diamond}$$

Figure 6.7: Scaletta Class Exclusion

The basic rules for class exclusion are powerful enough to avoid the

inheritance of a mixin, as demonstrated in our example. In the next section we show the limitations of these rules and suggest a natural extension.

6.5.7 Group Exclusion

The idea of class exclusion is very simple but unfortunately it does not cover all interesting uses of abstract inheritance; for type-checking the encoding of a method call we need to extend the mechanism of class exclusion with the concept of *group of classes*. Remind that a method call $e.m(3)$ is encoded as

```
class Apply extends e.m { field arg = 3; }
!Apply.result
```

For this class declaration to be well-formed, we must prove that `e.m` is not already an instance of `Apply`. When we define the method `m` we can just write

```
field m: M$def misses {arg} excludes {Apply}
```

to specify that `m` can only hold values that are not instance of `Apply`. If there are more than one call to `m` in the program we can extend the class exclusion annotation accordingly:

```
field m: M$def misses {arg} excludes {Apply1, ..., Applyn}
```

The problem with such a schema is that it is not extensible, in the sense that the number of possible applications will necessary be bounded. And we do not want to restrict the use of a method to a particular number of applications because it precludes the idea of separate compilation. So a simple idea is to group classes and to exclude groups instead of classes.

In this case we declare a group called `ApplyGroup`

```
group ApplyGroup;
```

And we let the field `m` exclude this group:

```
field m: M$def misses {arg} excludes {ApplyGroup}
```

Now when declaring a class `Apply`, we link it to the group `ApplyGroup`.

```
class Apply in ApplyGroup extends e.m { field arg = 3; }
!Apply.result
```

The key idea is that if `m` excludes the group `ApplyGroup`, it will a fortiori exclude the class `Apply`, which is a member of this group.

The adding of groups looks as a natural and harmful extension of our mechanism of class exclusion. But rather than formalizing groups directly we prefer simulate them with classes, which has the advantage to

avoid enriching unnecessarily the syntax with new concepts. We consider that each class defines implicitly a group and that a class is member of a group if it inherits from the corresponding class. This idea is formalized by adding a new rule to the class exclusion relation:

$$\text{(EX-GROUP)}\quad \frac{\textbf{class } C' \textbf{ extends } u\ \{\,\overline{d}\,\} \qquad \textbf{class } C' \textbf{ inside } B' \qquad B' \vdash u < C \qquad B \vdash t \textit{ is-not-a } C}{B \vdash t \textit{ is-not-a } C'}$$

Now, to make all methods member of a group `ApplyGroup`, we first define a "mixin" `ApplyGroup`:

```
class MixApplyGroup {
  field superclass: !Object excludes {ApplyGroup};
  class ApplyGroup extends superclass; }
```

And we declare all fields corresponding to methods to exclude the group `ApplyGroup`.

```
field m: M$def misses {arg} excludes {ApplyGroup};
```

Now, when calling a method, we first apply the mixin:

```
class Mix extends MixApplyGroup { field superclass = e.m; }
class Apply extends MixApplyGroupe!ApplyGroup { field arg = 3; }
!Apply.result
```

6.6 Encodings

This section describes how some higher-level constructs can be encoded in the typed calculus.

6.6.1 Methods

The encoding of method definitions described in Section 6.3.5 defines ill-formed members because the field corresponding to the method and the fields corresponding to its parameters are never declared. We show here how the encoding can be corrected for the typed calculus. To illustrate our purpose, we will show how the method `add` of the following example is encoded.

```
class Int {
  field pred: !Int;
  field succ: !Int;
```

```
  field succ = !Succ;
  field add(that: !Int): !Int;
  class Succ extends !Int {
    field pred = Int.this;
    method add(that) = pred.add(that.succ);
  }
}
class Zero extends !Int {
  field pred = this;
  method add(that) = that;
}
```

In the typed calculus, like fields, methods need to be declared. That is why the class `Int` now contains a definition of the method `add`. This definition is encoded like this:

```
  class Add { field that: !Int; field result: !Int; }
  field add: !Add misses {that};
```

The type of the field `add` indicates that it returns a `Add` instance with no field valuation for the field `that`. Therefore, in order to access the field `result` of the value returned by the field `add`, one is forced to first extend this value and assign a term to its field `that`. This corresponds exactly to what happens in a method application; in order to obtain the result of a method, one is forced to first provide an argument.

The encoding of method implementations has almost not changed. The only difference is that the implementation class now extends the method's definition class. Here is the encoding of the `add` implementation in class `Succ`.

```
    class AddInSucc extends !Add {
      field result = pred.add(AddInSucc.this.that.succ);
    }
    field add = !AddInSucc;
```

The encoding of method application has not changed at all. For completeness we repeat here the encoding of the call to the method `add` in the class `AddInSucc`.

```
    class AddInSucc {
      class Apply extends pred.add {
        field that = AddInSucc.this.that.succ;
      }
      field result = !Apply.result;
    }
```

This encoding is still not well-formed. Indeed, the class `Apply` extends a value that may already be an instance of the class `Apply`. Here we need our group exclusion mechanism. The correct definition of the field `add` in the class `Int` is given below. Note the additional exclude clause.

```
field add: !Add misses {that} excludes {ApplyGroup};
```

The correct encoding of the method call in the class `AddInSucc` is the following one:

```
class AddInSucc {
  class Mix extends MixApplyGroup {
    field superclass = pred.add;
  }
  class Apply extends !Mix.ApplyGroup {
    field that = AddInSucc.this.that.succ;
  }
  field result = !Apply.result;
}
```

6.6.2 Class Constructors

We consider the problem of defining class constructors. Given a class C with some uninitialized fields, we would like to define a constructor that initializes those fields. Furthermore, if the class C extends a class B, we would like to define C constructors such that they delegate the initialization of the B part to a B constructor.

Let us try to implement a constructor for each of the two following classes.

```
class Pt2D               { field x: !Int; field y: !Int; }
class Pt3D extends !Pt2D { field z: !Int; }
```

We implement a C constructor with two functions: `initC` and `newC`. The function `initC` takes as an argument an uninitialized C instance and as many additional arguments as needed to initialize the C instance. It returns the given C instance with all its fields initialized. It does that by defining a class `InitC` that extends the given C instance and that contains field valuations for all uninitialized fields defined in the class C. It then returns an instance of the class `InitC`. If the class C extends a class B, then instead of immediately returning the `InitC` instance, the method `initC` passes it to a call to a `init`B method and returns the result of this call.

The `newC` method takes as many arguments as needed to initialize a C instance and returns an initialized instance of C. It does that simply by

calling the initC method with a new C instance and its own arguments.

We give here the implementation of a constructor for each of the two classes defined above. The two init methods use the notation $t\ \{\overline{M}\}$, which is a syntactic sugar to represent an instantiation of an anonymous class with parent t and members $\overline{M}$.

```
method initPt2D(i: !Pt2D misses {x,y}, x0: !Int, y0: !Int): i =
  i { val x = x0; val y = y0; };
method newPt2D(x0: !Int, y0: !Int): !Pt2D =
  initPt2D(!Pt2D, x0, y0);

method initPt3D(i: !Pt3D misses {x,y,z}, x0: !Int, y0: !Int,
                                                  z0: !Int): i =
  initPt2D(i { val z = z0; }, x0, y0);
method newPt3D(x0: !Int, y0: !Int, z0: !Int): !Pt3D =
  initPt3D(!Pt3D, x0, y0, z0);
```

Note that the return type of the initC methods is not !C but i where i is the argument receiving the uninitialized C instance. This is needed to implement the calls to the init method of the superclass. If the initC methods had the type !C, then the body of the method initPt3D would not be well-formed. Indeed, it would have the type !Pt2D whereas !Pt3D is expected.

6.6.3 Interfaces and Mixins

Interfaces in Java are a means to approach structural subtyping while staying in a nominal world, because they make it possible to abstract over a set of methods. Let us consider the following declarations.

```
class B {}
interface I { int i(); }
interface J { int j(); }
class C extends B implements I,J {
  int i() { /* ... */ }; int j() { /* ... */ };
}
```

A method that accesses only the method i on its argument may declare it to have type I. Another method that accesses only the method j on its argument may declare it to have type J. Instances of classes that implement both interfaces, like C, may be passed to both methods.

Although Scaletta supports only single-class inheritance and has no notion of interface, the same kind of abstraction is possible. The trick is to transform an interface into a kind of function whose argument is the class

to which the interface has to be added. Here is how interface I is written in Scaletta.

```
class MixI {
  field o: !Object misses {*} excludes {I};
  class I extends o { field i: !Int; }
}
```

Assuming the interface J has been encoded the same way, it is possible to translate the Java class C into Scaletta. The translation requires two auxiliary classes:

```
class AuxCI extends !MixI { field o = !B; }
class AuxCJ extends !MixJ { field o = !AuxCI!I; }
class C extends !AuxCJ!J {
  field i = /* ... */; field j = /* ... */;
}
```

In order to declare a field f of type I, one might first try the following declaration:

```
field f: !MixI!I;
```

However, given that declaration, it is impossible to assign the value !C to f although class C inherits from class I. The problem is that !C expands to !AuxCI!I but not to !MixI!I. By giving f the type !AuxCI!I, the value !C would become legal but other values that inherit from class I through other auxiliary classes than AuxCI would still be illegal. What we need to express is that a value v may be assigned to f if it expands to w!I where w is a value that expands to !MixI. This can be done with an auxiliary field:

```
field auxf: !MixI;
field f: auxf!I;
```

To assign a value v that implements the interface I to f one has just to note that for such a value, v@I always expands to !MixI. Therefore, the following is always legal.

```
field auxf = v@I;
field f = v;
```

This shows that although Scaletta supports only single-class inheritance, it is able to provide the same level of abstraction as interfaces in Java, which supports multiple-interface inheritance in addition to single-class inheritance. Furthermore, if one considers mixins as interfaces with code, the techniques described here can also be used to implement mixins because nothing forbids us in our encoding to add value definitions to the classes simulating interfaces.

6.6.4 Type Abstractions

In Scaletta it is possible to abstract over a term by defining in a class a field with no value. A similar mechanism to abstract over a type would be to introduce in the calculus the concept of type fields. But surprisingly, without abandoning our formalism, we can express a kind of type abstraction. The idea is to simulate type fields with our standard value fields. Let us consider the following example where the value field `T` plays the role of an abstract type field.

```
class A { field T: !Int; field x: T; }
```

All we know about the abstract field `T` is that it expands to `!Int`. It implies that giving the value `!Zero!Succ` to `x` would be illegal, because it could happen that, in a subclass, `T` is given the value `!Zero`, which is not a parent of `!Zero!Succ`. That would break the invariant, assumed everywhere during typing, that the value of a field expands always to the value of its bound.

In the absence of a concrete value for `T`, the only values that can be given to `x` are terms that are explicitly declared to have the type `T` (for instance `x` itself). But as soon as `T` has a value, the type-checker can make use this information and show that a particular value given to `x` expands to it, as in the following subclass `B`.

```
class B extends !A { field T = !Int; field x = !Zero!Succ; }
```

The design pattern of letting value fields play the role of type fields can be used to encode a limited form of polymorphism. For example polymorphic lists with a concatenation method can be encoded as follows, where `T` represents the type of the elements.

```
class Lists {
  field T: !Object misses {*};

  class List {
    method concat(that: !List): !List;
  }

  class Nil extends !List {
    method concat(that) = that;
  }
  method cons(head: T, tail: !List): !List = !List {
    field concat(that) = cons(head, tail.concat(that));
  }
}
```

6.7 Undecidability and Typing Strategies

Typing in the field of programming languages can be seen as a way of approximating the result of an expression without evaluating it. Evaluating an expression consists of replacing abstractions by their value. So a good reason not to evaluate an expression at compile time is that some abstractions have no value. Even when abstractions have values, typing algorithms usually prefer to approximate the abstraction using its declared interface (or type), instead of its value, in order to prevent the compiler from looping (types usually do not loop). From this point of view typing can be seen as a kind of abstract evaluation. Scaletta makes this close link between typing and evaluation completely explicit because the typing relations generalize the evaluation relations, as explained in Section 6.4.

It follows that typing is undecidable because it can happen that the only way of checking that an expression conforms to a bound is to evaluate it. A naive solution to "approach decidability" is to never use the value of a field during type-checking but always use its bound. But if we do that we cannot simulate type fields with value fields anymore because the main property of type fields is precisely that their value is used at compile time. There is clearly a trade-off between the need of making the type-checking "more decidable", i.e. that it terminates on more programs, and the need of being able to write typed solutions to interesting problems.

As the excessive usage of field valuations is a source of undecidability in the typing of Scaletta, we define a *typing strategy* as a policy that restricts this usage. From a theoretical point of view, such a policy controls the usage of the rule ($\rightarrow^{abs}$-FIELD). It is important to note that a proof of type-safety for our type system would still hold if the usage of the deduction rules was restricted by a typing strategy. A type-checker using a restrictive typing strategy would just accept less programs than one using a more liberal strategy, but all accepted programs would still be guaranteed not to cause errors at runtime.

In conclusion we have a type system with a degree of decidability and expressiveness that is parameterizable by a typing strategy. We considered two simple typing strategies: one that requires that the programmer explicitly annotates field valuations that can be followed by the type-checker and another that infers from the context this right, for instance if a field plays the role of a type field. In our compiler we adopted the second strategy because it does not require additional annotations from the programmer and because there are in principle cases where a same field could be considered as a type or as a value in different contexts. Our experience with our interpreter indicates that the chosen strategy works well; it

accepts non-trivial programs, signals useful error messages and does not unexpectedly loop.

6.8 Scaletta vs. Core Language

Scaletta was designed with the aim of maintaining its semantics and its typing rules as simple as possible and to avoid as much as possible any redundancy. This makes Scaletta a very useful tool to study and prove properties of inner classes and virtual types. But, it is not very well-suited to be used as an intermediate language because many concepts need to be encoded into complex structures, which could only hardly be mapped to some real assembly code. For example, methods are encoded into several classes and fields and it would be very difficult to map these classes and fields to methods of the Java virtual machine.

The Core language was designed to be used as the intermediate language of a Scala compiler. It shares many aspects with Scaletta. In some sense, it is a pragmatic version of Scaletta. For example, Scaletta has an explicit syntax for functions. Thus, it is not necessary to encode them into complex structures. It makes also abstract inheritance unnecessary and thus avoids all the problems introduced by it. On the other hand, the additional typing rules for function calls duplicate several elements of the rules for instance creations and variable selections. Similarly the presence of explicit blocks avoids the need to encode them but again duplicates some elements of the rules for instance creations and variable selections.

The Core language requires neither hole annotations nor class exclusion annotations. However, the encoding of class constructors in Scaletta described in Section 6.6.2 shows that something similar could maybe be useful to track uninitialized variables, which is currently completely absent in the Core language.

In Scaletta, typing is undecidable because of the presence of abstract inheritance and also because there is no distinction between type fields and value fields. The Core language makes this distinction. Thanks to this and to the absence of abstract inheritance typing is decidable. The price to pay is some redundancy in the typing rules for type fields and value fields. This distinction also prevents the definition of typing as a kind of abstract evaluation.

Like Scaletta, the Core language distinguishes the notions of member definition and member overriding. The Core language furthermore distinguishes the notion of refining the signature of an inherited member from the notion of overriding the implementation of an inherited mem-

ber whereas in Scaletta field valuations play the two roles. There is also a difference in the way members are defined. In Scaletta, member definitions define abstract members, which can be implemented only once in a subclass. In the Core language, there are no abstract members furthermore functions can be overridden multiple times in subclasses and type fields can be refined multiple times. This is possible because there is no abstract inheritance and thus no risk of accidentally overriding or refining a member with an ill-typed body or signature.

The Core language introduces mutable fields. This has for consequence that it is necessary to distinguish stable expressions from ordinary expressions and to allow only stable expression in types. In Scaletta, all expressions are stable. The presence of mutable fields has also for consequence that a semantics for the Core language would be based on a heap and would therefore be completely different from the one of Scaletta.

The Core language also introduces the notion of record types with refinements. Thanks to these refinements, outer fields can be handled like normal fields. Indeed, in the Core language outer fields are just fields with a special symbol. Unlike in Scaletta, there are no special typing rules for outer fields. The record types of the Core language are also more fine-grained than the types of Scaletta. For example, it is possible to express the type of all instances of some inner class *C* with the record type *C* `{}`. This is not possible in Scaletta because one necessarily also has to specify the enclosing instance of the class *C*.

6.9 Related Works

Both concepts, inner classes and virtual types, have been studied and well understood separately. Inner classes are described in [14] and virtual types are formalized in [13].

The interaction between inner classes and virtual types is less well understood. We know only about four works related to this problem.

In [23] the authors aim at formalizing the type system of the Scala programming language. They do not especially focus on inner classes and virtual types. They describe a calculus that also includes other concepts like object identity, mixins and first-class class constructors. Its type system is proved to be sound. However the calculus is rather complex and therefore is very hard to use as a basis to understand the interaction between inner classes and virtual types. We claim to have a simpler calculus with just the required concepts to formalize inner classes and virtual types.

BETA is an object-oriented programming language with inner classes

and virtual classes. Virtual classes in BETA are equivalent to the notion of virtual types described in our paper. In [18] the author considers the problem of typing virtual classes in the presence of inner classes in BETA. The main difference between his and our work is that he describes the algorithms used by the BETA compiler to perform the semantic analysis of a program whereas we give a calculus with a type system that formally defines well-formed programs. We do not describe in the paper an algorithm to build well-formedness proofs. However we have written an interpreter that incorporates such algorithms. The algorithms presented for BETA performs simultaneously name and type analysis. A contribution of our work is to show that it is possible to specify the static semantics without having to include rules to perform name analysis. Note however that our interpreter implements a name analysis and thus does not impose globally unique names. The same paper contains interesting examples describing more intricate situations than our examples.

gbeta [11] is an extension of BETA with the possibility to extend virtual classes. This mechanism is similar to our notion of abstract inheritance. Our understanding of gbeta is very weak, and we would be interested to know if our solutions for typing abstract inheritance could somehow be used to type virtual classes in gbeta.

We lately discovered in [34, 33] that the author had developed ideas and solutions for a calculus of classes and objects very close to the ones we developed for Scaletta. The author distinguishes in his work two approaches to inheritance: the modificationist approach that views classes mainly as code libraries whereas the specialisationist approach insists on considering them as abstractions. As we do, he recommends the later, claiming that overriding is not essential. He develops also a mechanism similar to holes and uses it in exactly the same way as we do for encoding methods. However he is more extremist than we are when he claims that only one kind of names is necessary. On the contrary we think after our experience that it is important to make a clear distinction between class names and field names and that any unification of both concepts is likely artificial. His work goes beyond the work presented in this paper when he tries to integrate in his formalism a kind of structural subtyping for handling full polymorphism. Finally, the author is also missing a soundness proof that would validate his interesting ideas.

Chapter 7

Conclusion and Future Work

7.1 Scala

We have described in details several aspects of Scala with a focus on its numerous type constructs. We have informally explained how typing was performed. We have also described the relationships between different type constructs and in particular we have described how type parameters, virtual types, parameterized class types, qualified class types and refined types are related.

We have also exhibited some shortcomings of Scala due to its syntax and its typing rules and we have proposed generalizations to avoid them.

Finally we have informally described the lambda lift and the explicit outer transformations, which are two important code transformations of the Scala compiler. We have described the different steps of each transformation and explained why they are necessary. We have also discussed some typing issues that remain and described how they could be avoided.

7.2 Core Language

We have described the Core Language, a typed intermediate language for a Scala compiler. It consists only of the most fundamental constructs of Scala but can nonetheless encode most constructs of Scala. It generalizes some aspects of Scala to overcome the shortcomings we have identified in Scala.

We have described how Scala constructs are encoded into Core code. We have presented a type system for the Core language and we have given a formal description of the lambda lift and the explicit outer transformations.

In the future, there are many properties that we would like to prove formally. First of all, we would like to prove that our code transformations do indeed produce well-formed code. We would also like to prove that the type system is sound. To this effect we first need to develop a formal semantics for the Core language. This would also let us prove that our code transformations preserve the semantics of the transformed programs.

The description of the encoding of Scala programs into Core programs is rather informal. It would be interesting to formalize these encodings and also to describe how the Core language can be used by the analyzer not only as a target language but also as a base for all type computations performed during the name analysis and the type inference.

7.3 Scaletta

The main contribution of Scaletta is a better understanding of the interaction between inner classes and virtual types through a simple and novel calculus of classes and objects. We have formally described the non-trivial mechanism needed to type virtual types in the presence of inner classes. We have described the required alias analysis and showed that it requires only a few simple rules provided that each inner class comes with an implicit and accessible outer field.

This work has application for the design of type checkers for languages that combine inner classes and virtual types. It can also be used to understand type systems of existing programming languages that combine these two aspects like BETA or Scala. It may even be used to understand, at least partially, languages with inner classes and parameterized classes like Java 5, which require the same kind of alias analysis as those formalized in Scaletta.

Our priority in the future is to prove the soundness of the type system of Scaletta. This has already been partially achieved in [6] where a proof for a simpler version of Scaletta is given.

Bibliography

[1] Philippe Altherr and Vincent Cremet. Scaletta web page, August 2004. Available at `http://lamp.epfl.ch/$\sim$paltherr/scaletta`.

[2] Philippe Altherr and Vincent Cremet. Inner classes and virtual types. Technical report IC/2005/0133, EPFL, Switzerland, March 2005. Available from `http://scala.epfl.ch/`.

[3] Lennart Augustsson. A compiler for lazy ML. In *LFP '84: Proceedings of the 1984 ACM Symposium on LISP and functional programming*, pages 218–227, New York, NY, USA, 1984. ACM Press.

[4] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In Craig Chambers, editor, *ACM Symposium on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, pages 183–200, Vancouver, BC, Canada, 1998.

[5] Kim B. Bruce, Martin Odersky, and Philip Wadler. A statically safe alternative to virtual types. *Lecture Notes in Computer Science*, 1445:523+, 1998.

[6] Vincent Cremet. *Foundations for Scala: Semantics and Proof of Virtual Types*. PhD thesis, School of Computer and Communication Sciences , EPFL, Switzerland, May 2006. No. 3556.

[7] Olivier Danvy and Ulrik P. Schultz. Lambda-lifting in quadratic time. *Journal of Functional and Logic Programming*, 2004(1), July 2004.

[8] Burak Emir. Compiling regular patterns to sequential machines. In *SAC '05: Proceedings of the 2005 ACM symposium on Applied computing*, pages 1385–1389, New York, NY, USA, 2005. ACM Press.

[9] Burak Emir. Translation of pattern matching in a Java-like language. In *International Kyrgyz Electronics and Computer Conference (IKECCO)*, April 2006.

[10] Burak Emir, Andrew Kennedy, Claudio Russo, and Dachuan Yu. Variance and Generalized Constraints for C# Generics. In *European Conference on Object-Oriented Programming (ECOOP)*, July 2006.

[11] Erik Ernst. *gbeta – a Language with Virtual Attributes, Block Structure, and Propagating, Dynamic Inheritance*. PhD thesis, Department of Computer Science, University of Aarhus, Århus, Denmark, 1999.

[12] Atshushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In Loren Meissner, editor, *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA'99)*, volume 34(10), pages 132–146, New York, NY, USA, 1999.

[13] Atsushi Igarashi and Benjamin C. Pierce. Foundations for virtual types. *Lecture Notes in Computer Science*, 1628:161+, 1999.

[14] Atsushi Igarashi and Benjamin C. Pierce. On inner classes. *Lecture Notes in Computer Science*, 1850:129+, 2000.

[15] Thomas Johnsson. Lambda lifting: transforming programs to recursive equations. In *Functional programming languages and computer architecture. Proc. of a conference (Nancy, France, Sept. 1985)*, New York, NY, USA, 1985. Springer-Verlag.

[16] Bent Bruun Kristensen, Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard. The BETA programming language. In Bruce Shriver and Peter Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 7–48. M.I.T. Press, Cambridge, MA, USA, 1987.

[17] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, second edition, April 1999.

[18] Ole Lehrmann Madsen. Semantic analysis of virtual classes and nested classes. In *OOPSLA '99: Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 114–131, New York, NY, USA, 1999. ACM Press.

[19] Martin Odersky. Scala by example. Available from `http://scala.epfl.ch/`.

[20] Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Sebastian Maneth, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. An overview of the Scala programming language. Technical report IC/2004/64, EPFL, Switzerland, 2004. Available from `http://scala.epfl.ch/`.

[21] Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Sté phane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. An introduction to Scala. Available from `http://scala.epfl.ch/`.

[22] Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Sté phane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. The Scala language specification. Available from `http://scala.epfl.ch/`.

[23] Martin Odersky, Vincent Cremet, Christine Röckl, and Matthias Zenger. A nominal theory of objects with dependent types. In *Proceedings of the European Conference on Object-Oriented Programming*, Darmstadt, Germany, July 2003.

[24] Martin Odersky, Enno Runne, and Philip Wadler. Two ways to bake your pizza - translating parameterised Types into Java. In *Generic Programming*, pages 114–132, 1998.

[25] Martin Odersky and Philip Wadler. Pizza into Java: Translating theory into practice. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages (POPL'97), Paris, France*, pages 146–159, New York, NY, USA, 1997. ACM Press.

[26] Martin Odersky and Matthias Zenger. Independently extensible solutions to the expression problem. In *Proc. FOOL 12*, January 2005.

[27] Martin Odersky and Matthias Zenger. Scalable component abstractions. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 41–57, New York, NY, USA, 2005. ACM Press.

[28] Michel Schinz. A Scala tutorial for Java programmers. Available from `http://scala.epfl.ch/`.

[29] Michel Schinz. *Compiling Scala for the Java Virtual Machine*. PhD thesis, School of Computer and Communication Sciences , EPFL, Switzerland, 2005.

[30] Nathanael Schärli. *Traits: Composing Classes from Behavioral Building Blocks*. PhD thesis, Universität Bern, 2005.

[31] Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew Black. Traits: Composable units of behavior. In *Proceedings ECOOP 2003 (European Conference on Object-Oriented Programming)*, volume 2743 of *LNCS*, pages 248–274, New York, NY, USA, July 2003. Springer-Verlag.

[32] Kresten Krab Thorup and Mads Torgersen. Unifying genericity – combining the benefits of virtual types and parameterized classes. In *ECOOP Proceedings*, New York, NY, USA, June 1999. Springer-Verlag.

[33] Mads Torgersen. *Unifying Abstractions*. PhD thesis, Computer Science Department, University of Aarhus, Århus, Denmark, September 2001.

[34] Mads Torgersen. Inheritance is specialization. In *The Inheritance Workshop, with ECOOP 2002*, June 2002.

[35] Mads Torgersen, Christian Plesner Hansen, Erik Ernst, Peter von der Ahé, Gilad Bracha, and Neal Gafter. Adding wildcards to the Java programming language. In *SAC '04: Proceedings of the 2004 ACM symposium on Applied computing*, pages 1289–1296, New York, NY, USA, 2004. ACM Press.

Index

(), 12

abstract type, 88
alias analysis, 144, 153, 154, 158
`All`, 13
`AllRef`, 13
analyzer, 4
`Any`, 12
`AnyRef`, 12
`AnyVal`, 12
as-seen-from, 23, 26, 30, 43

back-end, 4
base-type, 22, 42, 46
`bottom`, 88
bottom type, 18, 88

call graph, 53
call set, 53
call-set, 130, 132
class
  hierarchy, 12
  member, 13, 97
  qualifier, 40, 72
  type, 18, 102
    parameter, 19, 33, 103
`class` file, 3
compound type, 46, 102
conformance, 18
constructor, 15, 57, 106, 171
contravariant
  method, 14
  position, 24
  type parameter, 24
covariant
  field, 15, 121, 122, 124
  method, 13, 121, 123
  position, 24
  type parameter, 24
current enclosing instance, 37
current instance, 82

definition, 110, 111

`Elem`, 127
`elem`, 61, 127
enclosing instance, 37
entity, 82
eo-$\mathcal{D}$, 140, 141
eo-e, 140–142
eo-$\mathcal{P}$, 140, 141
expand mixin phase, 5
explicit outer phase, 5, 67, 136
explicit self type, 46, 76, 125
extra set, 52
extra-$\overline{\mathcal{D}}$, 133, 136
extra-m, 133, 136
extra-$\overline{O}$, 133, 136
extra-set, 130, 132
extra-$\overline{x}$, 133, 136

field, 14
field refinement, 30
free-vars, 130, 132
front-end, 4
function type parameter, 19, 102

getter method, 14
greatest lower bound, 32

implicit constructor, 15
initialized-class, 108
inner class, 37, 58
instance creation, 17, 38, 106
interface, 12
invariant position, 24
invariant type parameter, 24
is-constructor, 108
is-initialization, 133
is-mutable, 110, 111
is-primary-constructor, 108
is-refed, 133, 136
is-stable, 110, 111

Java virtual machine, 3

lambda lift phase, 5, 49, 127
lambda lifting, 50
least upper bound, 18
lift, 110–112
ll-$\mathcal{D}$, 133, 134
ll-e, 133–135
ll-$\mathcal{P}$, 133, 134
lookup, 19, 23, 29, 31
lookup, 109, 110, 112
lower bound, 21
lower class bound, 21

map-$\mathcal{D}$, 110, 113
map-e, 110, 113
member inheritance, 18
member type, 28, 102
method, 13
method refinement, 30
mixin, 11, 45, 172
mutable variable, 60

nested class, 37

object, 12
outer
    field, 38, 92
    type field, 84
    value field, 84
`override`, 14
owner, 84
owner, 110, 111

package, 11, 100
parameter variance, 24
parameterized class type, 19, 103
parser, 4
path-to, 133, 136
plain type, 26
polymorphic class, 19
polymorphic method, 19
primary constructor, 16
`primitive class`, 16
primitive type, 17
`primitive var`, 15

qualified class type, 40, 44, 105
qualifier, 40, 72

record, 82
record type, 88
`Ref`, 61, 127
`ref`, 127
refed-vars, 130, 132
reference class, 13
refined type, 30, 44, 102
refinement, 28, 30, 88
refs-$\mathcal{E}$, 130, 131
refs-e, 130, 131
refs-m, 130, 131
refs-$\mathcal{P}$, 130, 131
rewriting phase, 4
`Root`, 82
root context, 82

Scala compiler, 3
secondary constructor, 16
self type, 46, 76
self-value, 108

setter method, 14
signature, 88
signature, 110, 111
singleton
    class, 12
    object, 12, 100
    type, 25, 88, 102
sink, 110–112
stable
    expression, 86, 91
    path, 25
    value, 25
      expression, 86
subst, 110, 112
symbol, 80, 82
symbol, 110, 111
syntactic sugar, 9, 14, 38, 86, 89, 91–93, 128, 150–152, 172

top type, 18, 88
trait, 12, 45
trans match phase, 5
type
    equality, 22
    erasure phase, 5
    parameter, 19, 33, 102, 103
    refinement, 28
    signature, 88

unit value, 12
upper bound, 21
upper class bound, 21

value class, 12
value signature, 88
variance, 24
virtual parameter, 33
virtual type, 28

widening types, 81
wildcard, 35, 104

Curriculum Vitæ

Personal information

Name	Philippe Altherr
Citizenship	Swiss (from Speicher AR)
Date of birth	September 18th, 1974
Place of birth	Pietermaritzburg, South Africa

Education

2000–2006	Ph.D. in Computer Science Laboratoire des Méthodes de Programmation École Polytechnique Fédérale de Lausanne, Switzerland
1993–1998	Master in Computer Science École Polytechnique Fédérale de Lausanne, Switzerland
1990–1993	High School (scientific orientation) Gymnase Auguste Piccard, Lausanne, Switzerland

Professional experience

2000–2005	Teaching Assistant Laboratoire des Méthodes de Programmation École Polytechnique Fédérale de Lausanne, Switzerland
1998–1999	Teaching Assistant Laboratoire des Systèmes Répartis École Polytechnique Fédérale de Lausanne, Switzerland