FOUNDATIONS FOR SCALA: SEMANTICS AND PROOF OF VIRTUAL TYPES

THÈSE N^{O} 3556 (2006)

PRÉSENTÉE LE 1 SEPTEMBRE 2006

À LA FACULTÉ INFORMATIQUE ET COMMUNICATIONS

Laboratoire de méthodes de programmation 1

SECTION D'INFORMATIQUE

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Vincent CREMET

DEA de programmation, Université Paris 7, France
et de nationalité française

acceptée sur proposition du jury:

Prof. E. Sanchez, président du jury
Prof. M. Odersky, directeur de thèse
Prof. C. Petitpierre, rapporteur
Dr F. Pottier, rapporteur
Prof. J. Vitek, rapporteur

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Lausanne, EPFL
2006

Contents

1 Introduction **17**
1.1 Scope . . . 17
1.2 Background . . . 19
1.2.1 Inner Classes . . . 19
1.2.1.1 Terminology . . . 19
1.2.1.2 Enclosing Instances . . . 19
1.2.1.3 Instance Creations . . . 20
1.2.1.4 Aliasing . . . 21
1.2.1.5 Summary . . . 22
1.2.2 Virtual Types . . . 22

2 Scala Semantics **23**
2.1 A Model of Computation for OO Languages . . . 24
2.1.1 Objects and Templates . . . 25
2.1.2 Example of Evaluation . . . 27
2.1.3 Object and Template Combination . . . 28
2.1.3.1 Merging declarations . . . 29
2.1.3.2 Tagging atomic templates . . . 31
2.1.3.3 Comparison of both proposals . . . 32
2.2 A Class-based Calculus: C-cal . . . 33
2.2.1 Some properties of classes . . . 33
2.2.2 Class-based Subset of MoC . . . 34
2.2.3 A Syntax with Primitive Classes . . . 35
2.2.4 Direct Semantics of C-cal . . . 36
2.2.4.1 Super-selections . . . 39
2.3 Semantics of a Functional Core of Scala . . . 40
2.3.1 Syntax of Core-Scala . . . 40
2.3.2 Anonymous Templates . . . 40
2.3.3 Parameters in Declarations . . . 42
2.3.3.1 Principle . . . 43
2.3.3.2 Formalization . . . 44
2.3.3.3 Invalid translation of parameterized objects . . . 46
2.3.4 Translating Core-Scala into C-cal . . . 46
2.3.5 Comparing Mixins in Scala and Core-Scala . . . 48
2.3.5.1 Linearization of classes and mixins . . . 48
2.3.5.2 Expansion of mixins in the Scala compiler . . . 49
2.3.6 Class Combination . . . 50
2.3.6.1 Virtual classes . . . 51

2.3.6.2 Class overriding 51
2.4 Scaletta 52
2.4.1 Tree Interpretation of Values 52
2.4.2 Syntax 53
2.4.2.1 Elimination of self variables 55
2.4.2.2 Link with the de Bruijn notation 55
2.4.3 Semantics of Scaletta 56
2.4.3.1 Super-selections 57
2.5 Conclusion 57
2.5.1 Summary 57
2.5.2 Related Work 59
2.5.2.1 Comparing MoC and ς-calculus 59
2.5.2.2 Lambda-calculi with records 60

3 A Soundness Proof of Virtual Types 61
3.1 Introduction 61
3.1.1 Extending Featherweight Java with Virtual Types 61
3.1.2 Comparison with Featherweight Generic Java 62
3.1.3 Overview 64
3.2 Sound Subtyping 64
3.2.1 Subtyping and Subclassing 64
3.2.2 Graph of Symbols 65
3.2.3 Naive Subtyping Rules 66
3.2.4 Naive Rules are Unsound 67
3.2.5 Backward Moves are Wanted for Type Aliases 68
3.2.6 Transitivity by Confluence 68
3.2.7 Cycles 70
3.2.8 Well-founded Relation 71
3.2.9 Incompatible bounds 72
3.2.10 Conclusion 72
3.3 Syntax 73
3.3.1 Programmer's View 73
3.3.2 Mathematician's View 75
3.3.3 Example 78
3.3.4 De Bruijn's Notation 78
3.4 Semantics 81
3.5 Typing 81
3.6 Well-formedness 85
3.7 Structured Subtyping 89
3.7.1 Definition 89
3.7.2 Motivations 90
3.7.3 Implementation 92
3.8 Compatibility of Bounds 96
3.9 Soundness Proof 99
3.10 Conclusion 99
3.10.1 Summary and Future Work 99
3.10.2 Evaluation Criteria 102
3.10.3 Related Work 103

A Complete Proof of Soundness **105**
A.1 Miscellaneous 105
A.2 Type Ordering 111
A.3 Subclassing 115
A.4 Admissibility of Transitivity 116
A.5 Progress 123
A.6 De Bruijn's Indices 126
A.7 Substitution Lemmas 130
A.8 Subject-reduction 137
A.9 Soundness 141

Résumé

SCALA est un langage de programmation attractif parce qu'il est à la fois expressif et fortement typé statiquement. Ce mariage contre nature vient au prix d'une certaine complexité dans les constructions du langage et dans l'analyse statique. Cette complexité rend la sûreté du typage incertaine. Le but de cette thèse est d'initier un processus de validation du système de typage de SCALA qui soit à la fois rigoureux et formel. La correction d'un système de typage ne peut être établie que par rapport à une description formelle de l'exécution d'un programme. La première contribution de cette thèse est la définition d'une sémantique formelle de SCALA, par traduction dans un calcul minimal basé sur les classes. SCALA donne deux moyens d'exprimer des abstractions de types : les paramètres de types et les membres de types, aussi appelés types virtuels. Les types virtuels paraissent plus primitifs vu qu'ils permettent d'encoder les paramètres de types alors que l'inverse n'est pas avéré. La correction des types virtuels a été l'objet d'un long débat dans la communauté, et il est désormais admis qu'ils sont sûrs. Cependant, à ce jour, aucun argument formel n'est venu confirmer cette croyance. La deuxième contribution de cette thèse est une preuve formelle de correction des types virtuels.

Mots-clés : Programmation orientée-objet, théorie des langages de programmation, SCALA, sémantique, système de types, types virtuels, sûreté du typage, modèle de calcul.

Abstract

SCALA is an attractive programming language because it is both very expressive and statically strongly typed. This marriage against nature comes at the price of a certain complexity in the language constructs and the static analysis. This complexity makes type safety unclear. The goal of this thesis is to initiate a rigorous and formal process of validation of the SCALA type system. The correctness of a type system can only be established in relation to a formal description of a program execution. The first contribution of this thesis is the definition of a formal semantics for SCALA, by translation in a minimal class-based calculus. SCALA offers two means for expressing type abstraction: type parameters and type members, also called virtual types. Virtual types seem more primitive since they allow to encode type parameters whereas the existence of a reverse encoding is not clear. The soundness of virtual types has been the object of a long debate in the community; they are now commonly believed to be safe, but at this time, there exists no formal argument that would confirm this belief. The second contribution of this thesis is to provide a formal proof of type safety for virtual types.

Keywords: Theory of programming languages, object-oriented programming, SCALA, semantics, type system, virtual types, type safety, model of computation.

Acknowledgements

I first thank Prof. Martin Odersky for accepting me as his PhD student, for the liberty he gave me in my work and for keeping confidence in me until the end of my thesis. I owe him also having designed an interesting programming language like Scala, which pleasantly occupied my life during these years.

I thank Philippe Altherr with whom I worked very closely during the last two years. The initial competition between us to formalize the Scala language turned quickly into a fruitful collaboration. There is almost nothing that one of us found without discussing with the other. We supervised each other so much that I like to think that our respective theses are two parts of the same work.

I thank Eduardo Sanchez for accepting being the president of my jury. I also would like to express my gratitude to the other members of my jury, Dr. François Pottier, Prof. Jan Vitek and Prof. Claude Petitpierre, for taking the time to read my thesis and for making useful comments that helped me improving its clarity.

I would like to thank François Garillot for proof-reading my chapter on the proof of soundness.

I thank Rachele Fuzzati for supporting me almost every minute during the last months of my thesis and particularly before the private defense, and for proof-reading the final version of the document.

If I enjoyed my time at LAMP, and in Switzerland in general, a lot is due to the presence of the "vrais potes" which roughly correspond to the French speaking part of the lab: Sébastien Briais, Michel Schinz, Stéphane Micheloud, Daniel Bünzli and Gilles Dubochet.

I thank my other colleagues and collaborators at LAMP with whom I had the chance and the great pleasure to work: Christine Röckl, Christoph Zenger, Matthias Zenger, Iulian Dragos, Burak Emir, Nikolay Mihaylov, Fabien Salvi, Yvette Dubuis, Johannes Borgström, Uwe Nestmann, Sebastian Maneth, Simon Kramer, Erik Stenmann. Some of them contributed to the development of Scala, on which everything in this thesis is about.

Finally, I thank all my family and my friends Sonia, Pierre and Yannick for accepting the fact that I pass more time with my ideas than with them and for always having confidence in me.

List of Figures

2.1 Semantics of CORE-SCALA by translation into SCALETTA and MOC 24
2.2 Syntax of MOC 26
2.3 Reduction of objects ($t \to u$) and templates ($T \to U$) in MOC . . 26
2.4 Declaration lookup in templates ($T \ni_x d$) and objects ($t \ni_x d$) in MOC 26
2.5 Syntax of C-CAL 36
2.6 Reduction of objects ($\Gamma \vdash t \to u$) and templates ($\Gamma \vdash T \to U$) C-CAL 38
2.7 Declaration lookup in templates ($\Gamma \vdash V \ni_x \overline{d}$) and objects ($\Gamma \vdash t \ni d$) in C-CAL 38
2.8 Super-selections in C-CAL 39
2.9 Syntax of CORE-SCALA 41
2.10 Parameter elimination in classes and objects 45
2.11 From CORE-SCALA to C-CAL 47
2.12 Example with a mixin inherited twice 48
2.13 Syntax of SCALETTA 54
2.14 From C-CAL to SCALETTA 55
2.15 Reduction of objects ($\Gamma \vdash t \to u$) and templates ($\Gamma \vdash T \to U$) in SCALETTA 58
2.16 Declaration lookup ($\Gamma \vdash t \ni d$) in SCALETTA 58
2.17 Super-selections in SCALETTA 58

3.1 Graph representation of a program 66
3.2 Oracle's example 68
3.3 Example needing a backward alias move 69
3.4 Cyclic example 70
3.5 Unfolded cyclic example 71
3.6 Incompatible bounds 72
3.7 Calculus Syntax 76
3.8 Mathematician's view of a program 79
3.9 Recovering the Programmer's View 79
3.10 Term reduction ($t \to u$) 82
3.11 Subclassing ($C \triangleleft C'$) 82
3.12 Instance Completeness (isComplete($C, \overline{f}$)) 85
3.13 Typing ($\Gamma \vdash t : T$) 86
3.14 Subtyping ($\Gamma \vdash T <: U$) 86
3.15 Type Well-formedness ($\Gamma \vdash T$ WF) 87

3.16 Member Well-formedness ($C \vdash d$ WF) 87
3.17 Class Well-formedness (D WF) 87
3.18 Roles of subtyping in a type-checker 95
3.19 Rules for class types with refinements 101

A.1 Context Well-formedness (wellFormed(Γ)) 131

List of Lemmas

3.1 Type expansion and type lowering . . . 94
3.2 Algorithmic rules use structured subtyping . . . 94
A.1 Admissibility of subsumption . . . 105
A.2 Self in self substitution . . . 105
A.3 Free parameters are in the context . . . 106
A.4 Free variables are in the context . . . 106
A.5 Values are irreducible . . . 109
A.6 Some terms are not typable in empty context . . . 109
A.7 Paths typable in empty context are values . . . 109
A.8 Chaining self substitutions . . . 109
A.9 Chaining self and parameter substitutions . . . 109
A.10 Chaining self and variable substitutions . . . 110
A.11 Interpretation and substitution . . . 111
A.12 Interpretation and substitution of a value . . . 111
A.13 Declarations and type ordering . . . 112
A.14 Type ordering and substitution of a value . . . 113
A.15 Facts about type ordering . . . 113
A.16 Multiset extension preserves well-foundedness . . . 113
A.17 Type ordering is well-founded . . . 113
A.18 The multiset extension of type ordering is well-founded . . . 114
A.19 Transitivity of subclassing . . . 115
A.20 Subclassing defines a hierarchy . . . 115
A.21 Subtyping implies subclassing . . . 115
A.22 Subclassing implies subtyping . . . 115
A.23 Unstructuring derivations . . . 116
A.24 Strengthening . . . 116
A.25 Admissibility of transitivity for class type subtyping . . . 117
A.26 Substitution for self in structured derivations . . . 117
A.27 Admissibility of transitivity for structured subtyping . . . 119
A.28 Structuring derivations . . . 122
A.29 Progress . . . 123
A.30 Lifting a closed term or type . . . 126
A.31 Dropping a closed term or type . . . 126
A.32 Values are invariant by lifting . . . 126
A.33 Permuting lifting and self substitution . . . 126
A.34 Permuting lifting and parameter substitution . . . 126
A.35 Permuting lifting and variable substitution . . . 127
A.36 Permuting dropping and substitutions . . . 129
A.37 Facts about lifting . . . 129

A.38 Weakening 130
A.39 Substitution lemmas 130
A.40 Subject reduction 137
A.41 Multi-step subject reduction 141
A.42 Soundness 141

Chapter 1

Introduction

1.1 Scope

SCALA is a new programming language developed by Martin Odersky and his team at LAMP [19]. From the programmer's point of view, SCALA is both a functional programming language and an object-oriented programming language. From the functional world, it takes the concepts of higher-order functions, algebraic data-types and pattern-matching; from the object-oriented world, it takes the concepts of objects, classes and mixins. SCALA is conceptually a pure object-oriented language because every value manipulated by the language is an object, even functional constructs are mapped to their object-oriented equivalents.

SCALA is an attractive programming language because it is both very expressive and statically strongly typed. Expressiveness and static typing are by nature two antagonist qualities: in the effort of filtering programs that behave badly at runtime, a type system tends to filter some correct and interesting programs too. Furthermore, powerful language constructs are likely to be difficult to type. SCALA provides powerful programming constructs and abstraction mechanisms like mixin composition, first-class functions, generic classes and methods, pattern-matching and arbitrary nesting of definitions, while still being able to statically check that they are always used in a safe way. In short SCALA reaches a trade-off between expressiveness and safety that has rarely been matched by statically typed object-oriented languages. This achievement comes at the price of a rich language of types and a complex static analysis. This complexity makes the soundness of the language unclear.

In principle SCALA should be safe because it has been designed with this goal. However, experience shows that we cannot completely rely on intuition in this domain because safety can easily be broken by small details, as illustrated by the following example.

```
abstract class C { type T >: Int <: String }
val a: C = null
val x: a.T = 42
val y: String = x
```

This piece of code was wrongly accepted by the version 1.3.0.10 of the SCALA compiler. In this example, type `T` of class `C` is declared with a lower-bound `Int`

and an upper-bound String. Since there exists no type that is both supertype of Int and subtype of String, there is no way of assigning a value to T, neither in class C nor in any of its subclasses. A consequence of this is that class C and all its subclasses cannot be instantiated because they inevitably contain at least one abstract member, namely T. Thus, we could wrongly infer that the incompatibility between the bounds of T is harmless. Unfortunately, null is a valid instance of C, which lets us define a concrete field a of type C. Because the lower-bound of T is Int, we can deduce that Int is a subtype of a.T; so it is possible to assign the integer value 42 to a field x declared with type a.T. Because the upper-bound of T is String, we can deduce that a.T is a subtype of String; so it is possible to assign the value x to a field y declared with type String. The problem is that, at this point, y, which is of type String, contains indirectly the integer value 42 via x. Of course, this is not safe because we can now invoke String operations on the integer value 42. Note that accepting a field declaration

```
def loop(): C = loop()
val a: C = loop()
```

would have been problematic for the same reasons, but in this case since fields are always initialized before the enclosing object can be used, the safety problem is converted into a looping initialization at runtime.

After we found this problem in the Scala compiler, type member declarations with incompatible bounds have been forbidden. Unfortunately, nothing ensures that we have not missed other subtle details like this one. In our case, the problem was clearly related to virtual types; this shows that although virtual types are intuitively simple (after all they are just as normal fields except they hold types instead of values), their theory is quite complex.

The ultimate goal of this thesis is to convince people that Scala is safe. This goal is however too ambitious and has actually never been reached for any other real programming language, because a real language is far too complex. So, the more realistic goal of this thesis is to extract the essence of Scala from its reference and its implementation in order to build some sound foundations of the language. With the results presented in this thesis, we have the ambition to initiate a rigorous and formal process of validation of the Scala type system. Our contributions to this process are the formal definition of a semantics for Scala and a soundness proof of a particular aspect of the Scala type system, namely virtual types.

Semantics. The correctness of a type system can only be established in relation to a formal description of a program execution. The first contribution of this thesis is the definition of a formal semantics for a functional subset of Scala, called Core-Scala. The Scala syntax is designed so as to be convenient for writing programs, not to ease the static analysis or the transformations performed by the compiler to reach an executable code. From this consideration, it is likely that the Scala syntax does not fit the direct description of a semantics or a type system. So, in order to define a semantics for Scala we first identify a minimal class-based calculus and then translate Scala into this calculus. This work is described in Chapter 2.

Proof of virtual types. SCALA offers two means for expressing type abstraction: type parameters and type members, also called virtual types. It is now well-established that type parameters can be encoded using virtual types. The soundness of virtual types has been the object of a long debate in the community; they are now commonly believed to be safe, but at this time, there exists no formal argument that would confirm this belief. The second contribution of this thesis is to provide a formal proof of type safety for virtual types. The proof is contained in Appendix A and described in Chapter 3.

1.2 Background

We review in this section two central features of SCALA that are widely discussed in this thesis, namely inner classes and virtual types. The presentation is inspired by the technical report [3], which explains the non-trivial interaction between the two mechanisms.

1.2.1 Inner Classes

In this section, we introduce our terminology about inner classes and explain what we call an inner class. We remind also some, maybe not so well-known, facts about them.

1.2.1.1 Terminology

A *nested class* is a class declared within another one. We distinguish two kinds of nested classes: *inner classes* which can access the current instance of their enclosing class and *static nested classes* which cannot. Within an inner class the current instance of its enclosing class is called the *current enclosing instance* and given an instance `i` of an inner class, it is called the *enclosing instance of* `i`.

Static nested classes are equivalent to top-level classes with some privileged rights to access static members of their enclosing class. These rights pose some interesting and non-trivial issues. However these issues are beyond the scope of this thesis. Here, we are only interested in the additional issues posed by the presence of a current enclosing instance in inner classes. The rest of this section illustrates these issues with some examples written in JAVA [16].

1.2.1.2 Enclosing Instances

In JAVA, any non-static class declared within some class `C` is an inner class. Within the inner class, the current enclosing instance is denoted by the expression `C.this` which is of type `C`. The code below declares an inner class `I` nested in a class `R`. It makes explicit the presence of a current enclosing instance of type `R` by declaring a field `outerI` of that type and initializing it with that instance.

```
public class R {
  class I { final R outerI = R.this; }
}
```

This example explicitly declares a field `outerI` that holds the current enclosing instance. However, every inner class really has a hidden field that holds this instance and the syntax `C.this` is just a way to access this hidden field. In fact, an inner class is nothing else than a static nested class with an additional field holding the current enclosing instance. We call this additional field *the outer field* of the inner class.

When an inner class is declared within another inner class, it can access the current instances of both of its enclosing classes. For example, within class `M` in the code below, the expression `R.this` denotes the current instance of the (indirectly) enclosing class `R`. This instance is, by definition, equal to `R.this` evaluated within class `I`. In other words, in class `M`, `R.this` is equal to the expression `this.outerM.outerI`.

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

So, although an inner class may have access to the current instance of several enclosing classes, a single outer field per class is sufficient to access all these instances. In the general case, within a class $\texttt{C}_0$ nested in a class $\texttt{C}_1$, ..., nested in a class $\texttt{C}_n$, the expression $\texttt{C}_i\texttt{.this}$ is equal to $\texttt{this.outerC}_0$ ... $\texttt{.outerC}_{i-1}$. Therefore, the syntax `C.this` would be superfluous if the outer fields were not hidden and were automatically initialized. It could always be replaced by a succession of outer field selections.

1.2.1.3 Instance Creations

To create a new instance of an inner class `D`, an instance of its enclosing class `C` has to be provided. In JAVA, the syntax *expr*`.new D(`*args*`)` is used to that effect. It creates a new instance of class `D` whose enclosing instance is the result of the expression *expr*. The enclosing instance may be omitted if the instance creation is enclosed, possibly indirectly, in a subclass `E` of `C`. In that case, the expression `new D(`*args*`)` is equivalent to the expression `E.this.new D(`*args*`)`. We illustrate this by augmenting the class `R` above with the two fields declared below. The value `i` is an instance of class `I` whose enclosing instance is the current instance of class `R` (the expression `new I()` really means `this.new I()`) and the value `m` is an instance of 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` declared below has two outer fields, namely `outerI` and `outerJ`, and every instance of the class `N` has the two outer fields `outerM` and `outerN`.

```
public class R {
  class I {
    final R outerI = R.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 R outerJ = R.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. These super constructor calls have the syntax `*expr*.super(*args*)` where *expr* must evaluate to an instance of the enclosing class `C` of the super class. As with instance creations, the enclosing instance may be omitted if the super constructor call is enclosed, possibly indirectly, in a subclass `E` of `C`. In that case, the expression `super(*args*)` is equivalent to `E.this.super(*args*)`.

1.2.1.4 Aliasing

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 `R.this.super()`. This implies that for any instance of class `J`, its two outer fields `outerI` and `outerJ` hold exactly the same value. The constructor of class `N` specifies that the enclosing instance of its superclass `M` is its argument `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.

The next example illustrates that it is sometimes possible to statically establish that two outer fields hold the same value. It implies that the two fields can be assumed by the compiler to share their typing knowledge.

```
public class R {
  class A {
    final R outerA = R.this;
    class X           { final A outerX = A.this; }
  }
  class B extends A {
    final R outerB = R.this;
    class Y extends X { final B outerY = B.this; }
  }
}
```

Instances of the class Y declared above have two outer fields: outerX and outerY. One can establish that both will always hold the same value, but both are not perfectly equivalent; outerY is of type B while outerX is only of type A. However, we know that outerX holds an instance of class B. Thus, we know that each time something is accessed via outerX, it is accessed on an instance of class B. The next section will show that this information is crucial in the context of virtual types.

1.2.1.5 Summary

To sum up, an inner class can be viewed as a top-level class with an additional field (the outer field) holding the current enclosing instance. This instance must be provided to all instance creations and all super constructor calls. An instance of an inner class inherits one outer field from each inner class it is an instance of. All those fields may hold different values, but sometimes it is possible to formally establish that some of them necessarily hold the same value.

1.2.2 Virtual Types

In some object-oriented languages, it is possible to declare abstract type members, i.e. type members that are bounded by a type but have no exact type value. These members may then be given different type values in different subclasses. This means that the exact value of such a type member depends on the exact class of the value from which it is selected. These type members are called *virtual types*. We illustrate virtual types with the following example written in SCALA :

```
abstract class M {
  type T <: Object
  val x: T
  val y: T = x
}
class N extends M {
  type T = String
  val x = "foo"
}
```

In class M, the fields x and y are both declared with the type T. It is therefore legal to assign x to y. Within class M, the exact value of T is unknown. It is only known that this value is bound by (is a subtype of) Object. Although "foo" has type String and String is a subtype of Object, it would be illegal to assign "foo" to x in class M because in subclasses of M, T may be assigned any subtype of Object and the value of x has to be an instance of that type. In the subclass N, T is assigned the type String. As in SCALA type assignments may not be overridden in subclasses, it is possible to assign "foo" to x in class N.

Chapter 2

SCALA Semantics

This chapter is about the semantics of SCALA. The semantics of a programming language consists of a set of definitions that describe the runtime behavior of a program written in this language. We see two main motivations for modeling the execution of a SCALA program. A first motivation is to document/specify the language implementation in a simple and unambiguous way; we hope the semantics given in this chapter can give an interesting perspective on non-trivial SCALA mechanisms such as inner classes and mixins. The second motivation is to let the semantics be the basis of some theoretical studies about the language; in this thesis we are mainly interested in proving the soundness of the SCALA type system.

Overview To be pragmatic we focus on a small functional subset of SCALA that we call CORE-SCALA. Actually we do not give CORE-SCALA a direct semantics, instead we look for the smallest calculus which is both able to support a sound and "reasonably decidable" type system that mimics the typing discipline of SCALA, and in which it is possible to translate CORE-SCALA simply. In our research of such a calculus we first identify a very general model of computation that supports natively the object-oriented concepts of member selection, code inheritance and self-recursion. We call this model of computation MOC. This calculus fulfills the second requirement about the possibility of translating CORE-SCALA simply, but it is dubious it can support a nominal type system as the one of SCALA because it does not contain the concept of class as primitive. This forces us to define a new calculus, named C-CAL, which is class-based but still closely connected with the model of computation. Then, we go from C-CAL to CORE-SCALA incrementally by two successive extensions of C-CAL, the first one adds anonymous classes, the second one adds parameters. Finally, we give a variant of C-CAL whose main innovation is the presence of a primitive operator for selecting an outer instance. By interpreting values as class-labeled trees, we are able to interpret this operator as the selection of a subtree. We claim that such operator simplifies the lookup relation, makes the calculus closer to an implementation and offers extra-possibilities for defining a type system. This last calculus is called SCALETTA. This chapter can be summarized by the diagram of Figure 2.1. Every box in this diagram corresponds to a calculus described in this chapter and every arrow from a calculus c_1 to a calculus c_2 corresponds to an encoding of c_1 in c_2.

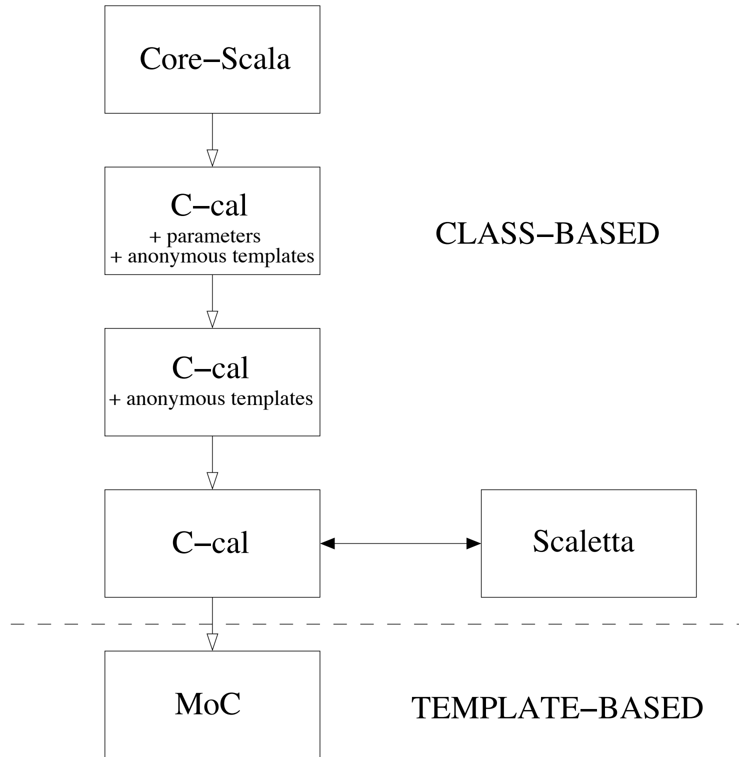


Figure 2.1: Semantics of Core-Scala by translation into Scaletta and MoC

2.1 A Model of Computation for OO Languages

One part of this thesis consists in formalizing the execution of a Scala program. One possibility is to define the semantics directly on the Scala syntax, another possibility is to define the semantics of a simpler language and to translate Scala into this core language. The first impression is that the direct semantics will be easier to read and understand because it is closer to the language we try to describe. However the indirect semantics can be more appropriate when both the core language and the translation from the source language to the core language are simple and natural; in this case it is reasonable to claim that the core language captures the essence of the source language. In this section we describe a core language for Scala.

In the field of programming languages, the lambda-calculus, or one of its variants, is very often chosen as core language. This is particularly true for functional languages, but several important developments for object-oriented calculi have been based on the lambda-calculus too. An overview of main object encodings in the lambda-calculus can be found in [5]. The advantage of choosing the lambda-calculus is clear: it is possible to reuse its large and well-established theory. The drawback is that it does not support natively the basic concepts of object-orientation as inheritance and self-recursion. In this section we describe a core language for object-oriented languages in which these concepts are primitive. In the rest of the thesis we refer to it as the model of

computation, or simply MoC.

2.1.1 Objects and Templates

Our model of computation for OO languages supports as primitives the concepts of member selection, code inheritance and self-recursion. It distinguishes two kinds of entities: templates and objects. The definitions of objects and templates are mutually recursive. An object is a recursive record of labeled terms. There are two kinds of labels and two kinds of terms: a template label L is mapped to a template term T, and an object label l is mapped to an object term t. Templates are also recursive records of labeled terms, but contrary to objects they can be incomplete, which means they can refer to labels for which they do not provide, and they are not supposed to provide, a value. Templates can be combined, each component of the combination contributing to the completion of the other parts. Complete templates can be instantiated as objects. The basic brick for templates is the atomic template $\{x \mid \overline{d}\}$. In this expression, x is a variable that represents the current instance of the template as the keyword `this` represents the current instance of a class in JAVA. The combination of two templates T and U is represented by the expression $T \,\&\, U$. An object can be seen as the fixed point of a template, the object that is the fixed point of a template T is denoted by the object term $\texttt{new}\ T$. Finally, from an object t we can select a template label with the expression $t.L$ or an object label with the expression $t.l$. The syntax and semantics of the model of computation are formalized in Figures 2.2, 2.3 and 2.4.

Most reduction rules of Figure 2.3 are simple congruence rules like rule (OR-CSELECT) that lets us reduce the prefix t of an object selection $t.l$. The most interesting rules are (OR-SELECT) and (TR-SELECT). The two rules are completely similar since the only difference is in the nature of the thing that is selected, an object or a label, so we focus our explanation on object selections. A selection is resolved in two steps: a declaration lookup and a substitution. From the prefix t of the selection we perform a lookup of the selected label l thanks to the auxiliary relation $t \ni_x d$. $t \ni_x (\texttt{object}\ l = u)$ means the object t contains the declaration $(\texttt{object}\ l = u)$ and that inside this declaration the current instance is represented by the variable x. In the rest of this thesis we refer to this variable as the self reference or the self variable [1]. The result of the selection is the term u in which the prefix t is substituted for the variable x, which is noted $u[x\backslash t]$. This last substitution provides the self recursion semantics of objects.

The lookup of a declaration d in a term t is defined by the rule (L-NEW): t must be an expression $\texttt{new}\ T$ and the template T must contain the declaration d. The auxiliary relation $T \ni_x \overline{d}$ is used to collect the declarations $\overline{d}$ owned by the template T. In this relation, the variable x has the same meaning as in the lookup relation. An atomic template $\{x \mid \overline{d}\}$ contains its declarations $\overline{d}$, the declarations owned by a compound template $T \,\&\, U$ are computed from the declarations $\overline{d}$ owned by T and the declarations $\overline{d}'$ owned by U. This computation is abstracted by the operation $\overline{d} \uplus_x \overline{d}'$; for now we consider that declarations are simply concatenated, what we note $(\overline{d}, \overline{d}')$, but later we study

[1] We avoid using the terminology "instance variable" because it is already used with a different meaning in the domain of OO languages.

Object label	l			
Template label	L			
Self variable	x, y			
Object	t, u	$::=$	x	current instance
		$\mid$	$t.l$	object selection
		$\mid$	$\texttt{new}\ T$	instance creation
Template	T, U	$::=$	$\{x \mid \overline{d}\}$	atomic template
		$\mid$	$T \,\&\, U$	compound template
		$\mid$	$t.L$	template selection
Declaration	d	$::=$	$\texttt{object}\ l = t$	object field
		$\mid$	$\texttt{template}\ L = T$	template field
Object value	v, w	$::=$	$x \mid \texttt{new}\ V$	
Template value	V, W	$::=$	$\{x \mid \overline{d}\} \mid V \,\&\, W$	

Figure 2.2: Syntax of MoC

$$(\text{OR-Select})\ \frac{t \ni_x (\texttt{object}\ l = u)}{t.l \to u[x \backslash t]} \qquad (\text{OR-CSelect})\ \frac{t \to u}{t.l \to u.l}$$

$$(\text{TR-Select})\ \frac{t \ni_x (\texttt{template}\ L = U)}{t.L \to U[x \backslash t]} \qquad (\text{TR-CSelect})\ \frac{t \to u}{t.L \to u.L}$$

$$(\text{OR-New})\ \frac{T \to U}{\texttt{new}\ T \to \texttt{new}\ U} \qquad (\text{TR-Left})\ \frac{T \to T'}{T \,\&\, U \to T' \,\&\, U}$$

$$(\text{TR-Right})\ \frac{U \to U'}{T \,\&\, U \to T \,\&\, U'}$$

Figure 2.3: Reduction of objects ($t \to u$) and templates ($T \to U$) in MoC

$$(\text{L-Atom})\ \frac{}{\{x \mid \overline{d}\} \ni_x \overline{d}} \qquad (\text{L-Compound})\ \frac{T \ni_x \overline{d} \quad U \ni_x \overline{d}'}{T \,\&\, U \ni_x \overline{d} \uplus_x \overline{d}'}$$

$$\overline{d} \uplus_x \overline{d}' = \overline{d}, \overline{d}' \qquad (\text{L-New})\ \frac{T \ni_x \overline{d}}{\texttt{new}\ T \ni_x d_i}$$

Figure 2.4: Declaration lookup in templates ($T \ni_x d$) and objects ($t \ni_x d$) in MoC

several refinements of this definition.

In this calculus, the values are the set of terms that do not contain object or field selections. More precisely, an object value v is either a self reference x or a instance creation `new` V, where V is a template value. It might be surprising to consider variables as values. Of course, a closed term, that is, a term in which every occurrence of a variable is bound by the self variable of an atomic template, will never reduce to a variable. However, it is common to consider variables as values, in the lambda-calculus for instance, and to let free variables represent constants. A template value V is either an atomic template $\{x \mid \overline{d}\}$ or a combination $V \,\&\, W$ of two template values. There exists an alternate presentation of this calculus where templates are flattened by the following reduction rule.

$$\text{(TR-FLATTEN)}\ \frac{}{\{x \mid \overline{d}\} \,\&\, \{x \mid \overline{d}'\} \rightarrow \{x \mid \overline{d} \uplus_x \overline{d}'\}}$$

In this case the set of template values consists only of atomic templates and the lookup becomes simpler.

$$\text{(L-NEW)}\ \frac{}{\texttt{new}\ \{x \mid \overline{d}\} \ni_x d_i}$$

The presentation with flattening of templates and the one with lookup of declarations inside compound templates are trivially equivalent. The reason why we chose the presentation where templates are not flattened becomes clear in a next section when we present a mechanism for combining declarations that relies on a traversal through the structure of templates.

2.1.2 Example of Evaluation

A program in MoC is just an object term. The evaluation of a MoC program is performed by successive reductions of this term until a value is reached. To illustrate this mechanism we consider the following MoC program. This program is an object term that we call t. This term is an object value because it has the form `new` T where T is an atomic template. Inside T the self reference is represented by the variable `root`. T contains two template declarations and one object declaration. This example is best understood by analogy with OO languages: we can think of template declarations as class declarations and of object declarations as field declarations. Actually, the translation of SCALA we present later in this chapter is based on this correspondence. We can see that despite its apparent simplicity this example is already quite evolved: we have a template `A` with an inner template `X`, the value of the field `foo` in `X` refers to the value of the field `bar` selected on the enclosing instance `a`. The template `B` is a combination of the template `A` and an atomic template that provides a declaration for the field `bar`; using a class terminology, we would say that the template `B` *extends* the template `A` and *implements* the field `bar`.

```
new { root |
  template A = { a |
    template X = { x |
      object foo = a.bar
    }
```

```
  }
  template B = root.A & { b |
    object bar = <something>
  }
  object myB = new root.B
}
```

The term t is not reducible in itself because it is value, it acts as a container for declarations. In order to define a MoC program u that uses these declarations we just have to make t a subterm of u. The example of such a program u is the term $(\texttt{new}\ t.\texttt{myB}.\texttt{X}).\texttt{foo}$. Here are the reduction steps corresponding to the evaluation of the program u into the value $<\texttt{something}>$. For clarity we have underlined the redexes. As we can see the only redexes of this calculus are either object selections or template selections.

$$
\begin{array}{lll}
u & = & (\texttt{new}\ \underline{t.\texttt{myB}}.\texttt{X}).\texttt{foo} \\
 & \rightarrow & (\texttt{new}\ (\texttt{new}\ \underline{t.\texttt{B}}).\texttt{X}).\texttt{foo} \\
 & \rightarrow & (\texttt{new}\ (\texttt{new}\ \underline{t.\texttt{A}}\ \&\ \{\texttt{b}\mid\ \cdots\}).\texttt{X}).\texttt{foo} \\
 & \rightarrow & (\texttt{new}\ \underline{t'.\texttt{X}}).\texttt{foo} \qquad \text{where } t' = \texttt{new}\ \{\texttt{a}\mid\ \cdots\}\ \&\ \{\texttt{b}\mid\ \cdots\} \\
 & \rightarrow & \underline{(\texttt{new}\ \{\texttt{x}\mid \texttt{object foo} = t'.\texttt{bar}\}).\texttt{foo}} \\
 & \rightarrow & \underline{t'.\texttt{bar}} \\
 & \rightarrow & <\texttt{something}>
\end{array}
$$

2.1.3 Object and Template Combination

Even if we assume that every atomic template does not contain several declarations of the same label, because of composition we cannot ensure this property is satisfied by an arbitrary template. Currently nothing is done in the semantics to deal with such conflicted declarations, more precisely every declaration has the same potentiality of being chosen in the reduction process. It implies that the semantics is not deterministic. In this section we propose several solutions to recover determinism.

The fact that an object can potentially see several declarations of the same label can be regarded as a problem or as a chance. If we regard it as a problem, the simplest solution to keep determinism consists in considering that declarations are always hidden by further declarations with the same name. This corresponds to the mechanism called overriding, or late binding, in object-oriented languages. Formally, it is sufficient to replace the definition of the function $\overline{d} \uplus_x \overline{d}'$ that merges declarations by the following one.

$$\overline{d} \uplus_x \overline{d}' = \overline{d} + \overline{d}'$$

In this definition the operation $\overline{d} + \overline{d}'$ gives precedence to the declarations that appear in the right-hand operand $\overline{d}'$. It can be defined by recurrence on the length of the first list of declarations.

$$
\begin{array}{lll}
\epsilon + \overline{d} & = & \overline{d} \\
(d,\, \overline{d}) + \overline{d}' & = & \begin{cases} \overline{d} + \overline{d}' & \text{if } \mathrm{label}(d) \in \mathrm{dom}(\overline{d}') \\ d,\, (\overline{d} + \overline{d}') & \text{otherwise} \end{cases}
\end{array}
$$

This solution is simple but we can think of a more evolved mechanism where the lookup of a label returns some well-defined combination of all the declarations that have been found for this label.

The idea of combining all declarations of a label during the execution of a program is not new, it is inspired by the similar mechanisms that exist in JAVA, SCALA and GBETA [13]. In a JAVA class, a *super-call* $\texttt{super}.m(\overline{t})$ executes the body of the method m corresponding to its definition in the super-class. It corresponds to inlining in the subclass the implementation of the method defined in the super-class. It can be done statically because a class always knows its super-class, so in this case we speak of *static* super-calls. In SCALA, there is a generalization of JAVA super-calls: a mixin can refer to the implementation of a method declared in its parent, but because a mixin does not know its parent at compile time, we then speak of *dynamic* super-call. It is as if the implementation of the super-class was inlined inside the mixin at runtime. In GBETA the body of a method can refer to the implementation of the unknown method by which it will be overridden, with the keyword `INNER`. All these mechanisms conceptually amount to inline a method in a method of another class than the one in which it is declared, statically when it is possible, and dynamically otherwise. Such an inlining can be seen as a way of combining the declarations of both classes.

In the rest of this section we consider two ways of combining MoC declarations which are inspired by the construct `super` in JAVA and SCALA. We refer sometimes to suchs mechanisms as *auto-combination*, or deep combination, because the combination of declarations is completely implicit in a program, it is a side-effect of the explicit combination of their enclosing atomic templates through the construct $T \,\&\, U$.

2.1.3.1 Merging declarations

In this section we present a mechanism for combining declarations, based on the keyword `super`, which implements the intuitive idea that a super-call just corresponds to some inlining of a declaration into another one. In the context of MoC we will not speak of super-calls, because we have no methods, but of *super-selections* ; we extend the syntax of objects and templates as follows.

Object	t	::=	$x.\texttt{super}.l$	object super-selection
		|	$\cdots$	(as before)
Template	T	::=	$x.\texttt{super}.L$	template super-selection
		|	$\cdots$	(as before)

With this syntax, a super-selection $x.\texttt{super}.l$ represents the value of the declaration of l in the super-template of the atomic template of which x is the self reference. Informally, the super-template of an atomic template is the template that is located directly at its left, it is a dynamic concept. For instance in the expression $(T_1 \,\&\, \{x \mid \overline{d}\}) \,\&\, T_2$ the super-template of $\{x \mid \overline{d}\}$ is T_1. Note that the value of this template is not necessarily known before running the program, because T_1 could be a template selection $t.L$.

In order to formally define the meaning of these new constructs, we just need to modify our function for merging declarations $\overline{d} \uplus_x \overline{d}'$ to take into account multiple declarations. The new merging function is actually a generalization of the previous one, it still gives precedence to right-most declarations but it

also inserts in the chosen declaration the contribution of previous declarations by replacing super-selections with their values. Here is a completely formal definition of this function.

$$
\begin{array}{lcl}
\overline{d} \uplus_x \overline{d}' & = & \overline{d} + \mathrm{replaceSuper}(\overline{d}',\, x,\, \overline{d}) \\
\hline
\mathrm{replaceSuper}((\texttt{object}\; l = t),\, x,\, \overline{d}) & = & \texttt{object}\; l = \mathrm{replaceSuper}(t,\, x,\, \overline{d}) \\
\mathrm{replaceSuper}(x.\texttt{super}.l,\, x,\, \overline{d}) & = & t \qquad \text{if } (\texttt{object}\; l = t) \in \overline{d} \\
\mathrm{replaceSuper}(x.\texttt{super}.L,\, x,\, \overline{d}) & = & T \qquad \text{if } (\texttt{template}\; L = T) \in \overline{d} \\
\mathrm{replaceSuper}(t.l,\, x,\, \overline{d}) & = & \mathrm{replaceSuper}(t,\, x,\, \overline{d}).l \\
\ldots & = & \ldots
\end{array}
$$

The auxiliary function $\mathrm{replaceSuper}(t,\, x,\, \overline{d})$ replaces all super-calls in t according to the declarations $\overline{d}$. Note that $\mathrm{replaceSuper}(x.\texttt{super}.l,\, y,\, \overline{d})$ is undefined when x and y are different variables; in such situations the term whose evaluation implied such a replacement simply gets stuck.

The simplicity of this definition is certainly hidden by the complexity of the notations. Let us illustrate the simple and intuitive underlying mechanism by an example. We consider a template T which is built from two sub-templates T_1 and T_2. T_1 contains a valuation for the object label f, and T_2 contains another valuation for f and a valuation for another object label g.

$$
\begin{array}{lcl}
T & = & T_1 \,\&\, T_2 \\
T_1 & = & \{x \mid \overline{d}\} \\
T_2 & = & \{x \mid \overline{d}'\} \\
\overline{d} & = & (\texttt{object}\; f = 3) \\
\overline{d}' & = & (\texttt{object}\; g = x.f * x.\texttt{super}.f,\; \texttt{object}\; f = 4 + x.\texttt{super}.f)
\end{array}
$$

We want to show that

$$T \ni_x (\texttt{object}\; g = x.f * 3,\; \texttt{object}\; f = 4 + 3)$$

By rule (L-Atom), $T_1 \ni_x \overline{d}$ and $T_2 \ni_x \overline{d}'$. By rule (L-Compound), $T \ni_x \overline{d} \uplus_x \overline{d}'$ where

$$
\begin{array}{lcl}
\overline{d} \uplus_x \overline{d}' & = & \overline{d} + \mathrm{replaceSuper}(\overline{d}',\, x,\, \overline{d}) \\
 & = & (\texttt{object}\; f = 3) + (\texttt{object}\; g = x.f * 3,\; \texttt{object}\; f = 4 + 3) \\
 & = & (\texttt{object}\; g = x.f * 3,\; \texttt{object}\; f = 4 + 3)
\end{array}
$$

With the previous example we have understood how the declarations of two templates are combined together. With the next example, we chain three templates, T_1, T_2 and T_3, containing super-selections and we check that we obtain the desired result.

$$
\begin{array}{lcl}
T & = & (T_1 \,\&\, T_2) \,\&\, T_3 \\
T_1 & = & \{x \mid \texttt{object}\; f = 3\} \\
T_2 & = & \{x \mid \texttt{object}\; f = 4 + x.\texttt{super}.f\} \\
T_3 & = & \{x \mid \texttt{object}\; f = 5 * x.\texttt{super}.f\}
\end{array}
$$

As expected the resulting declaration of f is built from the contributions of templates T_1, T_2 and T_3:

$$T \ni_x (\texttt{object}\ f = 5 * (4 + 3))$$

2.1.3.2 Tagging atomic templates

As we have already said, a JAVA, or SCALA, super-call amounts to inline a method of a super-class in its subclass. In this inlining the current instance `this` of the super-class becomes the current instance `this` of the subclass. It means that super-calls are conceptually always performed on the current instance of a class. The same observation can be made about our first proposal for combining declarations: a super-selection $x.\texttt{super}.l$ contains the information of a self variable x. It is quite natural to want to generalize this notation so that super-calls, or super-selections, can be made on arbitrary objects, instead of just the current instance. This is such a generalization that we present in this section. The idea is to have a new set of labels representing tags and to let each atomic template be tagged with a tag k. Then, we modify the lookup relation on templates ($T \ni_x \overline{d}$) so that it becomes sensitive to tags. The modifications of the syntax needed by this extension are summarized below.

Tag label	k			
Object	t	$::=$	$t.\texttt{super}[k].l$	object super-selection
		$\mid$	$\cdots$	(as before)
Template	T	$::=$	$\{\, x \mid \overline{d}\,\}^k$	tagged atomic template
		$\mid$	$t.\texttt{super}[k].L$	template super-selection
		$\mid$	$\cdots$	(as before)
Template value	V, W	$::=$	$\{\, x \mid \overline{d}\,\}^k \mid V \,\&\, W$	

With this syntax, a super-selection $t.\texttt{super}[k].l$ is similar to the selection $t.l$ except the lookup of a declaration for l starts from the super-template of the first atomic template tagged with k, instead of starting from the first atomic template. The modifications in the rules required by this extension are summarized in the following figure.

$$(\text{L-ATOM})\ \frac{}{\{\, x \mid \overline{d}\,\}^k \ni_x \overline{d}} \qquad (\text{L-COMPOUND})\ \frac{V \ni_x \overline{d} \qquad V' \ni_x \overline{d}'}{V \,\&\, V' \ni_x \overline{d} + \overline{d}'}$$

$$(\text{L-MATCH})\ \frac{V \ni_x \overline{d}}{V \,\&\, \{\, x \mid \overline{d}'\,\}^k \ni_x^k \overline{d}} \qquad (\text{L-MISMATCH})\ \frac{k' \neq k \qquad V \ni_x^{k'} \overline{d}}{V \,\&\, \{\, x \mid \overline{d}'\,\}^k \ni_x^{k'} \overline{d}}$$

$$(\text{L-NEW-SUPER})\ \frac{V \ni_x^k \overline{d}}{\texttt{new}\ V \ni_x^k d_i} \qquad (\text{OR-SUPER})\ \frac{t \ni_x^k (\texttt{object}\ l = u)}{t.\texttt{super}[k].l \to u[x \backslash t]}$$

The relation $V \ni_x \overline{d}$ is used for normal selections (and for super-selections once the first template to consider has been determined). It is defined in such a way that precedence is given to right-most declarations. The relation $V \ni_x^k \overline{d}$ is used for super-selections, it represents a lookup initiated from the super-template of the first atomic template tagged with k.

It is important to note that the relation $V \ni^k_x \overline{d}$ is undefined for atomic templates tagged with a tag k' which is different from k. That corresponds to perform a super-selection from a template that has no super-template. In such situations the super-selection that initiated this lookup simply gets stuck. The relation is also undefined for templates that associate to the right like $T_1 \,\&\, (T_2 \,\&\, T_3)$. We could remove such templates from the syntax, it would lead to the following definition for templates.

Template	T	$::=$	$\{\, x \mid \overline{d} \,\}^k$	tagged atomic template
		$\mid$	$t.L$	template selection
		$\mid$	$t.\mathtt{super}[k].L$	template super-selection
		$\mid$	$T \,\&\, \{\, x \mid \overline{d} \,\}^k$	tagged compound template

Templates that are structured this way reduce to template values that are non-empty lists of atomic templates.

$$\text{Template value} \quad V, W \quad ::= \quad \{\, x \mid \overline{d} \,\}^k \mid V \,\&\, \{\, x \mid \overline{d} \,\}^k$$

Static selections. Along the same ideas as super-selections it is possible to define a mechanism of *static selection*. A static selection $t.\mathtt{static}[k].l$ is similar to a super-selection $t.\mathtt{super}[k].l$ except it makes the lookup start from the first template tagged with k instead of making it start from the super-template of this template. In the context of a class-based object-oriented language like JAVA, a static selection corresponds to the possibility of executing the implementation of a method m defined in a given super-class C. JAVA has no syntax for that, contrary to SCALA for which the syntax $\mathtt{super}[C].m(\overline{t})$ is used, where $\overline{t}$ are the arguments of the method call. The reason static calls are less interesting than super calls is that there exists a simple design pattern for emulating them: for each method m of a class C, add a method m_C that calls the method m, then replace each static call $\mathtt{super}[C].m(\overline{t})$ with the normal call $\mathtt{this}.m_C(\overline{t})$.

2.1.3.3 Comparison of both proposals

For defining super-selections as we do in the second proposal, based on tagged atomic templates, we need to keep the structure of templates because the lookup relation is based on this structure. For the first proposal this structure is not used and we could actually have flattened templates, as explained in a previous section.

The second proposal fits better the description of semantics for class-based languages like SCALA. Without anticipating too much on the translation of SCALA into the model of computation, it is clear that in this context the tags attached to atomic templates will correspond to class names.

Another observation is that the translation of the first proposal into the second one is trivial: take a program, tag all atomic templates with a different tag, inside a template $\{\, x \mid \overline{d} \,\}^k$ replace all occurrences of $x.\mathtt{super}.l$ with $x.\mathtt{super}[k].l$, and the same thing for template super selections. It is also possible to encode the second proposal with the first one but it is less elegant. It consists in adding in every atomic template some declarations for the labels that will potentially be selected through super selections. For instance in a

template $\{ x \mid \overline{d} \}^k$ we add the declaration ($\texttt{object}\ l_{\text{super-k}} = x.\texttt{super}.l$). And we replace all super-selections $t.\texttt{super}[k].l$ by normal selections $t.l_{\text{super-k}}$. Actually, in order to implement generalized super-calls on a platform that supports only super-calls invoked on the current instance, we would have to implement the design pattern described above. With separate compilation it is impossible to know which methods are going to be be called through generalized super-selections. And considering that all methods are potentially called this way implies to double each class with a bridge method, which is not realistic.

2.2 A Class-based Calculus: C-CAL

Our model of computation is closer to object-oriented programming languages than the lambda-calculus because it supports natively the following important object-oriented concepts: objects are records of values that can be accessed through their associated label, self-recursion of objects is modeled directly by the self reference inside templates, and finally the concept of code inheritance is present in the operator of template composition. Our goal in this chapter is to model the semantics of SCALA, but in SCALA there is a very important concept that is not present in MoC : classes. In this section, we give an informal definition of classes, based on the properties they satisfy in languages like JAVA, SCALA or GBETA. Then, we characterize a subset of MoC where certain template declarations enjoy similar properties as the ones we identified for classes. Finally, we present a class-based calculus, called C-CAL, which models the properties of classes in a primitive way. For this calculus, we give both an interpretation in MoC and a direct semantics.

In our perspective of designing a nominal type system for modeling the static analysis performed by the SCALA compiler, the direct semantics has the advantage of being based on classes. We justify now the interest of a class-based semantics. A nominal type system considers that some types have a name and compares these types using their names rather than their structure. In the context of object-oriented languages, class names play the role of these named types. So, if a nominal type system for SCALA manipulates class names, it is natural for a proof of soundness to be developed on a class-based semantics.

2.2.1 Some properties of classes

By inspection of their properties in programming languages like JAVA, SCALA and GBETA, we try to capture the essence of classes from a semantic point of view.

In SCALA, fields and methods can be declared abstract in a class and implemented in a subclass. The exact value of such members is determined at run time by a lookup through the class hierarchy. On the contrary, class and mixin declarations are not subject to dynamic binding. Each time a class name or a mixin name appears in the program, the exact value of the entity it represents is determined at compile time. For instance when referring to a class `C` in an instance creation expression `new C`, we know exactly which class declaration will be used, no dynamic lookup is necessary, and a direct link pointing to the declaration of the class can be established statically. The fact that name analysis requires a *static* lookup of the class `C` does not contradict our point. The

static lookup is just a means to distinguish between classes that have the same name but that are conceptually different entities. Unfortunately it is possible in SCALA to define in a class A a mixin M which has the same name as a mixin defined in a super-class B of A, it is then possible to refer inside A to the mixin in B with the expression `super.`M. But contrary to appearances, the second mixin is not an overriding of the first one and the mixin super selection is not resolved at run time. Conceptually it just declares a new mixin M', and `super.`M is just an alias for M'. It is certainly unfortunate to have a misleading syntax like that: for methods, redefinition means overriding and super call means dynamic binding, for classes, redefinition means new definition and super call means static binding. We want to define a calculus and a semantics for SCALA that satisfies the following property: values of classes are determined at compile time, values of fields are determined at runtime. In GBETA the opposite is true, precisely the essence of GBETA is that values of classes are determined at runtime, that is why they are called virtual classes.

In class-based object-oriented languages, classes are privileged locations for declaring new values, functions or types. In SCALA and JAVA this is not the only place where declarations can occur: in JAVA we can also have variable declarations inside methods and blocks, in SCALA method and class declarations can even appear inside blocks and method bodies. However, in a language like GBETA where the concepts of class and method are unified, every single declaration is attached to a class.

The last important semantic role of classes is to specify a mechanism for letting some parts of a program inherit code from some other parts. In object-oriented languages, classes are actually the only location where the code inheritance relation is defined. In SCALA or JAVA, the declaration of a class associates a class name with an optional parent. This parent is usually a class, called the super-class, or a class combined with mixins and interfaces. In GBETA, overriding is also allowed for classes and a class inherits implicitly from the class it overrides in addition to its list of class parents.

Here is a summary of properties that are satisfied by classes in SCALA.

1. The value of a class can be determined statically, i.e. no dynamic lookup is necessary.
2. Classes are place-holders for declarations.
3. Classes represent the only place where the code inheritance relation is defined.

Certainly there is not just one reason why most mainstream object-oriented languages are based on this concept of class. For structuring information in the programmer's head, it is certainly natural to have a single place, not subject to dynamic binding, where declarations and inheritance are defined. Another reason is that structuring information this way can simplify the design of a type system and make its proof of soundness easier.

2.2.2 Class-based Subset of MoC

In this section we characterize a class-based subset of MoC. By class-based we mean that inside programs of this subset some template declarations satisfy the properties we have identified for classes in the previous section.

Informal characterization. By definition, in order to know if a MoC program belongs to the class-based subset, the following properties must be satisfied. Among the template labels occurring in the program we distinguish a set that we call class labels. There must be only one template declaration corresponding to a class label. Only class labels can contain an atomic template in the right-hand side of their declaration, they must contain one and only one. This atomic template must be tagged with the class label. Finally the operator of template composition can occur only in the right-hand side of a class label declaration.

A model of classes. Now we can check that the minimal concept of class characterized by the three informal properties of the previous section is actually present in the restricted calculus. The first property states that contrary to other kinds of members, classes are not subject to dynamic lookup; in the class-based subset class declarations are unique so the value of a class can be determined statically. The second property is that classes are privileged place-holders for declarations; this corresponds to the constraint in the class-based subset that atomic templates are allowed only in the right-hand side of a class-template declaration. Finally, the third property says that classes are the only place where code inheritance is described; the counterpart in the class-based subset is that template composition appears only in class-template declarations.

2.2.3 A Syntax with Primitive Classes

Figure 2.5 contains the syntax of a calculus, named C-CAL, where classes are primitives. C-CAL corresponds closely to the class-based subset of the model of computation that we identified in the previous section.

In MoC a program is just a term and all declarations used for reducing the program are contained inside the term itself. Even if this view simplifies the definition of a calculus, it does not simplify its understanding because it does not convey the idea that inside a program there are parts that do not change like declarations and parts that do change like the current evaluation state of the program. C-CAL makes this distinction between the static part and the dynamic part of a program; a C-CAL program has the form $\{x \mid \overline{d}\ t\}$, it consists of a list of declarations $\overline{d}$ and a main term t to be evaluated in the context of these declarations. The variable x is a binder, it can occur both in the declarations $\overline{d}$ and in the main term t; it represents an implicitly defined root object containing the declarations $\overline{d}$. This presentation has to be connected with mainstream OO languages, where a program consists of a global set of classes and an entry point; in JAVA and SCALA the entry point is the name of a class containing an implementation for a static method called `main`, in our calculus the entry point is just a term.

Another difference with the model of computation is a new kind of declaration which lets us define classes. A class declaration consists of a class name C, a template parent T_{opt}, a self variable x and a set of declarations $\overline{d}$. The parent of a class is an optional template, we represent it with the meta-variable T_{opt}. More generally, we adopt the convention that if e is a meta-variable then e_{opt} represents an optional element of the set denoted by this meta-variable; the absence of such an element is then denoted by `none`. Class declarations correspond to the distinguished class-template declarations of the class-based subset

Class name	C			
Template label	L			
Object label	l			
Self variable	x			
Declaration	d	$::=$	$\texttt{class}\ C\ \texttt{extends}\ T_{opt}\ \{\, x \mid \overline{d}\,\}$	class
		$\mid$	$\texttt{template}\ L = T$	template field
		$\mid$	$\texttt{object}\ l = t$	object field
Template	T, U	$::=$	$t.L$	template selection
		$\mid$	$t.C$	class template
		$\mid$	$t.C :: T_{opt}$	extended template
Object	t, u	$::=$	x	self reference
		$\mid$	$t.l$	object selection
		$\mid$	$\texttt{new}\ T$	instance creation
Program	P	$::=$	$\{\, x \mid \overline{d}\ t\,\}$	
Object value	v	$::=$	$x \mid \texttt{new}\ V$	
Template value	V	$::=$	$v.C :: V_{opt}$	
Evaluation context	Γ	$::=$	$x \ni \overline{d}$	

Figure 2.5: Syntax of C-CAL

of MoC we have informally defined in a previous section. More precisely, a class declaration

$$\texttt{class}\ C\ \texttt{extends}\ T\ \{\, x \mid \overline{d}\,\}$$

corresponds to the declaration of the template

$$\texttt{template}\ C = T\ \&\ \{\, x \mid \overline{d}\,\}^C$$

Also, a program $\{\, x \mid \overline{d}\ t\,\}$ corresponds to the MoC term $t[x \backslash \texttt{new}\ \{x \mid \overline{d}\}]$. The constraint that class declarations must be unique is not specified by the syntax. It is a condition that has to be added to the definition of well-formed C-CAL programs.

In C-CAL templates are either a class template $t.C$, a template selection $t.L$ or an extended template $t.C :: T_{opt}$. This last kind of template is not meant to appear inside programs, rather it is a semantic entity that represents the current evaluation state of a template.

Object and template values correspond to objects and templates such that no sub-expression is a selection or a class template. It means that an object value v is either a self reference x or an instance creation expression $\texttt{new}\ V$ where V is a template value, and a template value V is necessary an extended template $v.C :: V_{opt}$. In Section 2.4.1 we explain how template values can be seen as lists of pairs $(v,\ C)$ and object values as class-labeled trees.

2.2.4 Direct Semantics of C-CAL

In the previous section we have explained how to interpret a C-CAL program as a MoC program. We could formalize this interpretation and it would actually constitute a valid semantics for C-CAL. However, a direct semantics based on

classes is preferable in the perspective of writing type safety proofs, because there is no doubt we want to base a type system for SCALA on the concept of class, and if we do so, the semantics should also be based on classes. Such a direct semantics for C-CAL is summarized in Figures 2.6 and 2.7. In the rest of this section we discuss its novel and interesting aspects.

The lookup relation on templates has the form $\Gamma \vdash V \ni_x \overline{d}$ in C-CAL ; it means that in the evaluation context Γ the template V contains the declarations $\overline{d}$ and that inside these declarations the self reference is represented by the variable x. This relation also exists in MoC, except that it is not parameterized by a context Γ, and it has exactly the same meaning. However the definition of this relation is more complex in C-CAL and it is interesting to see why. The main reason is that a MoC template value consists of atomic templates only, which means that all declarations are directly accessible. On the contrary, template values are not completely unfolded in C-CAL because they still contain class names, and declarations can only be accessed indirectly through these class names. The problem is that it is not possible to simply follow a class name and collect its declarations because inside these declarations there can be references to enclosing current instances that would escape their scope. But let us take an example to illustrate this problem.

```
{ root |
  class A extends none { a |
    class B extends none { b |
      object foo = a
      object bar = b
    }
  }
  (new (new root.A).B).foo
}
```

Let Γ be the evaluation context $root \ni \texttt{class}\ A\ \texttt{extends none}\ \{a \mid \cdots\}$, and suppose we want to compute the declarations of $(\texttt{new}\ root.A).B$. If we just collect declarations of class B without further treatment, we deduce the following judgment.

$$\Gamma \vdash (\texttt{new}\ root.A).B \ni_b (\texttt{object}\ foo = a, \texttt{object}\ bar = b)$$

However this cannot be correct because the variable a that was bound in the program becomes free in the lookup judgment. Note that the current instance b does not suffer this problem because it is bound inside the lookup judgment. The correct result of the lookup is obtained by replacing the variable a with its actual value, here $\texttt{new}\ root.A$.

In our example, the depth of nesting of the declarations we consider is two, by definition, because they are enclosed in two classes. More generally, when a declaration d declared at depth n is selected we must somehow replace each of the n self variables corresponding to the n enclosing classes of d with some values. Our solution to define a lookup relation that performs these n substitutions is to let the lookup relation on templates ($\Gamma \vdash V \ni_x \overline{d}$) and the lookup relation on objects ($\Gamma \vdash t \ni d$) have mutually recursive definitions. For defining the latter we must consider two cases: if t is a self variable x, it can only be the root instance of the program and d can be any declaration defined at top-level;

$$(\text{OR-Select}) \frac{\Gamma \vdash t \ni (\texttt{object}\ l = u)}{\Gamma \vdash t.l \to u}$$

$$(\text{TR-Select}) \frac{\Gamma \vdash t \ni (\texttt{template}\ L = T)}{\Gamma \vdash t.L \to T}$$

$$(\text{TR-Extends}) \frac{\Gamma \vdash t \ni (\texttt{class}\ C\ \texttt{extends}\ T_{opt}\ \{\, x \mid \overline{d}\,\})}{\Gamma \vdash t.C \to t.C :: T_{opt}}$$

$$(\text{OR-CSelect}) \frac{\Gamma \vdash t \to u}{\Gamma \vdash t.l \to u.l} \qquad (\text{OR-New}) \frac{\Gamma \vdash T \to U}{\Gamma \vdash \texttt{new}\ T \to \texttt{new}\ U}$$

$$(\text{TR-CSelect}) \frac{\Gamma \vdash t \to u}{\Gamma \vdash t.L \to u.L} \qquad (\text{TR-CClass}) \frac{\Gamma \vdash t \to u}{\Gamma \vdash t.C \to u.C}$$

$$(\text{TR-CExtends}) \frac{\Gamma \vdash T \to U}{\Gamma \vdash t.C :: T \to t.C :: U}$$

Figure 2.6: Reduction of objects ($\Gamma \vdash t \to u$) and templates ($\Gamma \vdash T \to U$) C-cal

$$(\text{L-Templ}) \frac{\Gamma \vdash V_{opt} \ni_x \overline{d} \qquad \Gamma \vdash v \ni (\texttt{class}\ C\ \texttt{extends}\ T_{opt}\ \{\, x \mid \overline{d}'\,\})}{\Gamma \vdash v.C :: V_{opt} \ni_x \overline{d} + \overline{d}'}$$

$$(\text{L-Root}) \frac{\Gamma = x \ni \overline{d}}{\Gamma \vdash x \ni d_i} \qquad (\text{L-New}) \frac{t = \texttt{new}\ V \qquad \Gamma \vdash V \ni_x \overline{d}}{\Gamma \vdash t \ni d_i[x \backslash t]}$$

Figure 2.7: Declaration lookup in templates ($\Gamma \vdash V \ni_x \overline{d}$) and objects ($\Gamma \vdash t \ni d$) in C-cal

$$\text{(L-MATCH)}\ \frac{\Gamma \vdash V_{opt} \ni_x \overline{d}}{\Gamma \vdash v.C :: V_{opt} \ni_x^C \overline{d}} \qquad \text{(L-MISMATCH)}\ \frac{C \neq C' \quad \Gamma \vdash V_{opt} \ni_x^{C'} \overline{d}}{\Gamma \vdash v.C :: V_{opt} \ni_x^{C'} \overline{d}}$$

$$\text{(L-NEW-SUPER)}\ \frac{t = \texttt{new}\ V \qquad \Gamma \vdash V \ni_x^C \overline{d}}{\Gamma \vdash t \ni^C d_i[x \backslash t]}$$

$$\text{(OR-SUPER)}\ \frac{\Gamma \vdash t \ni^C (\texttt{object}\ l = u)}{\Gamma \vdash t.\texttt{super}[C].l \to u}$$

$$\text{(TR-SUPER)}\ \frac{\Gamma \vdash t \ni^C (\texttt{template}\ L = U)}{\Gamma \vdash t.\texttt{super}[C].L \to U}$$

Figure 2.8: Super-selections in C-CAL

if t has the form `new` V, a lookup on the template V is performed and d is any declaration found by this lookup where t has been substituted for the current instance. In the definition of $\Gamma \vdash V \ni_x \overline{d}$, we know that V has the form $v.C :: V_{opt}$. To obtain the declarations of C where all required substitutions have been performed, we can simply perform a lookup of class C on the object v. As an illustration of all the mechanisms that have been introduced in this section we present the derivation corresponding to the evaluation of our example.

$$\text{(OR-SELECT)}\ \cfrac{\text{(L-NEW)}\ \cfrac{\text{(L-TEMPL)}\ \cfrac{\text{(L-NEW)}\ \cfrac{\text{(L-TEMPL)}\ \cfrac{\text{(L-ROOT)}\ \cfrac{}{\Gamma \vdash root \ni (\texttt{class}\ A\ \texttt{extends none}\ \{\, a \mid \cdots \})}}{\Gamma \vdash root.A \ni_a (\texttt{class}\ B\ \texttt{extends none}\ \{\, b \mid \texttt{object}\ foo = a, \cdots \})}}{\Gamma \vdash \texttt{new}\ root.A \ni (\texttt{class}\ B\ \texttt{extends none}\ \{\, b \mid \texttt{object}\ foo = \texttt{new}\ root.A, \cdots \})}}{\Gamma \vdash (\texttt{new}\ root.A).B \ni_b (\texttt{object}\ foo = \texttt{new}\ root.A, \cdots)}}{\Gamma \vdash \texttt{new}\ (\texttt{new}\ root.A).B \ni (\texttt{object}\ foo = \texttt{new}\ root.A)}}{\Gamma \vdash (\texttt{new}\ (\texttt{new}\ root.A).B.foo) \to \texttt{new}\ root.A}$$

2.2.4.1 Super-selections

For MoC we proposed two means of defining a mechanism of auto-combination of declarations. For C-CAL we present a mechanism which is similar to the second proposal based on a tag-sensitive lookup of declarations inside a template. We extend the syntax of C-CAL with object super-selections $t.\texttt{super}[C].l$ and template super-selections $t.\texttt{super}[C].L$; they have the same meaning as the corresponding constructs in MoC.

In Figure 2.8, we summarized the additional rules needed to deal with super-selections in C-CAL. These rules are similar to the ones defined in the tag-based extension of MoC with super-selections (See Section 2.1.3.2).

2.3 Semantics of a Functional Core of SCALA

As stated in the introduction of this chapter, we do not have the ambition to provide a semantics to the complete SCALA language. Rather, we focus on an untyped functional subset of SCALA, that we call CORE-SCALA. The semantics of CORE-SCALA takes the form of a translation into the class-based calculus C-CAL, it is summarized by the following formula.

$$\text{SEM}_{\text{Core-Scala}} = \text{TRANS}_{\text{Core-Scala}\to\text{C-cal}} + \text{SEM}_{\text{C-cal}}$$

The translation from CORE-SCALA to C-CAL is simpler to define and explain by successive small transformations. We start by considering a simple extension of C-CAL with anonymous templates, then we extend this extension by the addition of parameters in declarations. Finally, we translate CORE-SCALA in this last extension of C-CAL. After defining the semantics we informally validate its conformance to the SCALA implementation by focusing on the translation of mixins. To conclude this section we study an extension of CORE-SCALA with deep combination of classes, a mechanism similar to GBETA virtual classes.

2.3.1 Syntax of CORE-SCALA

In Figure 2.9 we give the syntax of CORE-SCALA. Despite its simplicity this calculus is already very expressive. It supports the usual object-oriented concepts of classes, methods and fields. But it also has inner classes, mixins, dynamic super calls, blocks, first-class functions and anonymous classes. The presence of blocks and anonymous classes implies that declarations can be nested at any level inside terms. Our core SCALA is functional, not in the sense that it only contains the concept of functions, but in the sense that it has no imperative features such as I/O primitives, mutable fields and mutable variables.

The main difference between our core SCALA and the real SCALA, except for the type annotations that we do not consider in this chapter, is the way we deal with the current instance and the current enclosing instances of a class. In SCALA the expression `C.this` represents the current instance of the class `C`, where `C` must be the class in which the expression occurs or an enclosing class of this class. In our core SCALA we use a variable for representing the current instance of a class. The correspondence between both notations is immediate. Our treatment of current instances extends better to the treatment of blocks and anonymous classes. Contrary to SCALA, every block or anonymous class contains a variable x that represents the actual instance of this block or anonymous class. Blocks have the form $\{x \mid \overline{d}\ t\}$ and anonymous classes have the form $\texttt{new}\ e_{opt}\ \{x \mid \overline{d}\}$. Inside a block, members of the block can be referenced through the variable x. Another implication of this treatment of current instances is that SCALA super-calls $C.\texttt{super}.m(\overline{t})$ are replaced by super-calls of the form $x.\texttt{super}.m(\overline{t})$.

2.3.2 Anonymous Templates

In this section we extend the syntax of C-CAL with anonymous templates.

$$\text{Template} \qquad T \quad ::= \quad T_{opt}\ \{\, x \mid \overline{d}\,\} \mid \cdots$$

Class name	C			
Mixin name	M			
Field name	f			
Method name	m			
Self reference	x			
Parameter	z			
Declaration	d	::=	$\texttt{class}\ C(\overline{z})\ \texttt{extends}\ p$	class declaration
		|	$\texttt{trait}\ M\ \texttt{with}\ \overline{k}\ \{x \mid \overline{d}\}$	mixin declaration
		|	$\texttt{val}\ f = t$	field declaration
		|	$\texttt{def}\ m(\overline{z}) = t$	method declaration
Pattern	p	::=	$e_{opt}\ \{x \mid \overline{d}\}$	
Extends clause	e	::=	$t.C(\overline{t})\ \texttt{with}\ \overline{k}$	
Mixin	k	::=	$t.M$	mixin use
Term	t, u	::=	x	self reference
		|	z	parameter
		|	$t.f$	field selection
		|	$t.m(\overline{t})$	method call
		|	$x.\texttt{super}.m(\overline{t})$	super-call
		|	$\texttt{new}\ p$	instance creation, anonymous class
		|	$\{x \mid \overline{d}\ t\}$	block
		|	$(\overline{z}) \Rightarrow t$	anonymous function
		|	$t(\overline{t})$	function call
Program	P	::=	$\{x \mid \overline{d}\ t\}$	

Figure 2.9: Syntax of CORE-SCALA

A C-CAL program with anonymous templates can simply be transformed into an equivalent program without anonymous templates. The translation function $[\![\cdot]\!]_y$ is parameterized by a variable y that represents the self reference of the class that contains the term, template or declaration to be translated. The idea of the translation is very simple: it consists in replacing every occurrence of an anonymous template $T_{opt}\ \{\, x \mid \overline{d}\,\}$ with a class template $y.C$ where C is a fresh class name. The class C is generated by the translation and added to the class whose y is the self reference. Its only role is to give a name to the anonymous template. Here is its definition.

$$\texttt{class}\ C\ \texttt{extends}\ [\![T_{opt}]\!]_y\ \{\, x \mid [\![\overline{d}]\!]_x\,\}$$

Even if the idea is very simple, the definition of the translation function becomes quite technical as soon as we want to describe it in a declarative mathematical way, because of the synthetic classes that must be added to existing classes. So for this time, we ask the reader to believe such a function can be defined properly. As an illustration of the principle we give an example.

```
{ root |
  class B { b | }
  class A { a |
    object bar = new root.B { x | object foo = 3 }
  }
}
```

In this example, the translation of the anonymous template `root.B { x | object foo = 3 }` results in the class template `a.Anon$1`, where `Anon$1` is a synthetic class added to the class `A`.

```
  class A { a |
    class Anon$1 extends root.B { x | object foo = 3 }
    object bar = new a.Anon$1
  }
```

2.3.3 Parameters in Declarations

In CORE-SCALA, classes and methods have parameters. In order to bring C-CAL closer to CORE-SCALA, we extend C-CAL with the possibility of having parameters in class and object declarations. We show that parameters can actually be encoded in C-CAL using normal declarations. Eliminating the concept of parameters through encoding has the advantage of reducing the number of concepts in the calculus. Note that not only this simplifies the syntax of the calculus, but also has the potential to simplify all definitions that depend on the syntax, from the semantics to the type system. If we look at this question from an implementation point of view, it also makes a difference. In the usual and efficient implementation of methods, their arguments are stored on the stack. Since we intend to encode the parameters of a method as the fields of a class, they are likely to be stored on the heap.

The parameters that we intend to add to C-CAL represent either objects or templates. The additional syntax needed for this extension is summarized below. The meta-variable z (resp. Z) ranges over parameters that represent an

object (resp. a template). Class and object declarations now have parameters $\overline{p}$. Finally, class templates and object selections must be passed some arguments $\overline{a}$.

Object parameter	z		
Template parameter	Z		
Parameter	p	::=	$z \mid Z$
Argument	a	::=	$t \mid T$
Declaration	d	::=	`class` $C(\overline{p})$ `extends` T_{opt} $\{ x \mid \overline{d} \}$
		$\mid$	`object` $l(\overline{p}) = t \mid \cdots$
Template	T, U	::=	$Z \mid t.C(\overline{a}) \mid \cdots$
Object	t, u	::=	$z \mid t.l(\overline{a}) \mid \cdots$

2.3.3.1 Principle

The principle behind the elimination of parameters is to convert each parameter into a field. Object parameters become object fields and template parameters become template fields. The first idea for implementing this principle is to let a parameter x of a class C be a field of the class C. For instance

```
class C(x, y) { c |
  object foo = new root.C(y, x)
}
```

becomes

```
class C { c |
  object x = c.x
  object y = c.y
  object foo = new root.C { anon |
    object x = c.y
    object y = c.x
  }
}
```

Note that the fields `x` and `y` are conceptually abstract in class `C`, because they correspond to formal parameters. In principle, we could have omitted their default declarations in `C`. They are useful nevertheless to stress the membership of `x` and `y` to the class `C`.

The only problem with this translation scheme is when a parameter is used in the extends clause of its class, like in the following example.

```
class D(y) { d | }
class C(x) extends root.D(x) { c | }
```

By definition, the scope of a self variable is limited to the declarations of its class. However, if we let `x` be a field of class `C`, its scope gets limited to the declarations inside `C`, and its use in the extends clause `root.D(x)` becomes problematic. Actually, the solution exists, as shown by the program below, our argumentation just explains why such a translation can cannot be compositional.

```
class D { d |
  object y = d.y
}
class C extends root.D { c |
  object x = c.x
  object y = c.x
}
```

The solution to have a compositional translation of class parameters is to enclose the class inside a newly generated class and to let the parameters be fields of this class. As an illustration of this translation scheme, we present our solution for the last example.

```
class D' { d' |
  object y = d'.y
  class D { d | }
}
class C' { c' |
  object x = c'.x
  class C extends (new root.D'{ object y = c'.x }).D  { c | }
}
```

2.3.3.2 Formalization

In order to translate this extension of C-CAL into the extension of C-CAL with just anonymous templates, we assume the existence of an infinite set $\{ l_0, l_1, \cdots \}$ of object labels for representing object parameters and an infinite set $\{ L_0, L_1, \cdots \}$ of template labels for representing template parameters. We also assume for each object label l (resp. each class name C) the existence of a template label L_l (resp. a class name C_C). All these labels and names are supposed not to appear in the original program to translate.

The translation function $[\![\cdot]\!]_\sigma$ is described in Figure 2.10, it is parameterized by a substitution σ that maps object parameters to objects and template parameters to templates. When the translation starts, this substitution is empty; it is then extended each time the translation reaches a nested parameterized declaration. The mappings that are added to the substitution are computed by the auxiliary function $\mathrm{param}(x, \overline{p})$ which takes as arguments a self variable x and the sequence of parameters $\overline{p}$ of the declaration to be translated. Then, it is very simple: the first parameter is mapped to $x.l_1$ if it is an object parameter and $x.L_1$ otherwise, the second parameter is mapped to $x.l_2$ if it is an object parameter and $x.L_2$ otherwise, etc.

We illustrate the formalization of parameter elimination by translating the following class which models scalable two-dimensional points. In this example there is a parameterized class `Point`, a parameterized object `scale` and the class template with arguments `root.Point(x * z, y * z)`.

```
{ root |
   class Point(x, y) { Point_this |
     object scale(z) = new root.Point(x * z, y * z)
   }
```

$\mathrm{param}(x, \overline{p})$	$=$	$\mathrm{param}_0(x, p_0), \cdots, \mathrm{param}_{n-1}(x, p_{n-1})$
$\mathrm{param}_i(x, z)$	$=$	$z \mapsto x.l_i$
$\mathrm{param}_i(x, Z)$	$=$	$Z \mapsto x.L_i$
$\arg(\overline{a})$	$=$	$\arg_0(a_0), \cdots, \arg_{n-1}(a_{n-1})$
$\arg_i(t)$	$=$	$\texttt{object}\ l_i = t$
$\arg_i(T)$	$=$	$\texttt{template}\ L_i = T$
$[\![\texttt{object}\ l = t]\!]_\sigma$	$=$	$\texttt{object}\ l = [\![t]\!]_\sigma$
$[\![\texttt{template}\ L = T]\!]_\sigma$	$=$	$\texttt{template}\ L = [\![T]\!]_\sigma$
$[\![\texttt{class}\ C(\overline{p})\ \texttt{extends}\ T_{opt}\ \{\ x \mid \overline{d}\ \}]\!]_\sigma$	$=$	$\texttt{class}\ C_C\ \texttt{extends}\ \ \{\ y \mid d\ \}$
		where $d = \texttt{class}\ C\ \texttt{extends}\ [\![T_{opt}]\!]_{\sigma'}\ \{\ x \mid [\![\overline{d}]\!]_{\sigma'}\ \}$
		where $\sigma' = \sigma, \mathrm{param}(y, \overline{p})$
$[\![\texttt{object}\ l(\overline{p}) = t]\!]_\sigma$	$=$	$\texttt{template}\ L_l =\ \{\ y \mid \texttt{object}\ l = [\![t]\!]_{\sigma, \mathrm{param}(y, \overline{p})}\ \}$
$[\![x]\!]_\sigma$	$=$	x
$[\![t.l]\!]_\sigma$	$=$	$[\![t]\!]_\sigma.l$
$[\![t.\texttt{super}[C].l]\!]_\sigma$	$=$	$[\![t]\!]_\sigma.\texttt{super}[C].l$
$[\![\texttt{new}\ T]\!]_\sigma$	$=$	$\texttt{new}\ [\![T]\!]_\sigma$
$[\![z]\!]_\sigma$	$=$	$\sigma(z)$
$[\![t.l(\overline{a})]\!]_\sigma$	$=$	$(\texttt{new}\ [\![t]\!]_\sigma.L_l\ \{\ x \mid \arg([\![\overline{a}]\!]_\sigma)\ \}).l$
$[\![t.\texttt{super}[C].l(\overline{a})]\!]_\sigma$	$=$	$(\texttt{new}\ [\![t]\!]_\sigma.\texttt{super}[C].L_l\ \{\ x \mid \arg([\![\overline{a}]\!]_\sigma)\ \}).l$
$[\![t.C]\!]_\sigma$	$=$	$[\![t]\!]_\sigma.C$
$[\![t.L]\!]_\sigma$	$=$	$[\![t]\!]_\sigma.L$
$[\![Z]\!]_\sigma$	$=$	$\sigma(Z)$
$[\![t.C(\overline{a})]\!]_\sigma$	$=$	$(\texttt{new}\ [\![t]\!]_\sigma.C_C\ \{\ x \mid \arg([\![\overline{a}]\!]_\sigma)\ \}).C$
$[\![\{\ x \mid \overline{d}\ t\ \}]\!]$	$=$	$[\![\{\ x \mid [\![\overline{d}]\!]_\epsilon\ [\![t]\!]_\epsilon\ \}]\!]$

Figure 2.10: Parameter elimination in classes and objects

```
}
```

As we can see in the result of the translation below, parameterized classes and parameterized objects are treated differently: class `Point` is enclosed in a synthetic *class* `C$Point` whereas object `scale` is enclosed in a synthetic *template* `L$scale`.

```
{ root |
  class C$Point { C$Point_this |
    class Point { Point_this |
      template L$scale = { L$scale_this |
        object l = new (new root.C$Point { anon_this |
          object l$1 = C$Point_this.l$1 * L$scale_this.l$1
          object l$2 = C$Point_this.l$2 * L$scale_this.l$1
        }).Point
      }
    }
  }
}
```

In the transformed program, the term `C$Point_this.l$1` (resp. the term `C$Point_this.l$2`) corresponds to the use of the parameter `x` (resp. `y`) in the original program, and `L$scale_this.l$1` corresponds to the use of the parameter `z`. When explaining the principle of the translation, we have reused the names of variables as names for the corresponding fields, whereas in the formalization we use labels l_1, l_2, L_1, etc. The advantage of the choice made in the formalization is that the translation of a class application can be made independently of the translation of the corresponding class declaration, just by having a convention in the use of labels l_i's and L_i's.

2.3.3.3 Invalid translation of parameterized objects

For reasons that have been exposed earlier we want our transformation to keep the invariant that there is only one class declaration corresponding to a class name. It means that for encoding a method declaration l we cannot simply declare a class C_l, like in the following translation, because several implementations in several sub-classes would generate several classes with the same class name.

$$\begin{array}{lcl} [\![\texttt{object}\ l(\overline{p}) = t]\!]_\sigma & = & [\![\texttt{class}\ C_l(\overline{p})\ \texttt{extends}\ \{\, x \mid \texttt{object}\ l = t\,\}]\!]_\sigma \\ [\![t.l(\overline{a})]\!]_\sigma & = & [\![(\texttt{new}\ t.C_l(\overline{a})).l]\!]_\sigma \end{array}$$

Contrary to classes we do not impose special constraints on template declarations, so the solution for translating parameterized objects consists in defining a template L_l.

2.3.4 Translating Core-Scala into C-cal

The transformation described in this section deals with removing mixins from Core-Scala and replacing super-calls selected on self-variables with generalized super-calls selected on arbitrary objects.

$[\![x]\!]_\sigma$	$=$	x
$[\![z]\!]_\sigma$	$=$	z
$[\![t.f]\!]_\sigma$	$=$	$[\![t]\!]_\sigma.f$
$[\![t.m(\overline{t})]\!]_\sigma$	$=$	$[\![t]\!]_\sigma.m([\![\overline{t}]\!]_\sigma)$
$[\![x.\texttt{super}.m(\overline{t})]\!]_\sigma$	$=$	$x.\texttt{super}[\sigma(x)].m([\![\overline{t}]\!]_\sigma)$
$[\![\texttt{new}\ p]\!]_\sigma$	$=$	$\texttt{new}\ [\![p]\!]_\sigma$
$[\![\texttt{class}\ C(\overline{z})\ \texttt{extends}\ (e_{opt}\ \{x \mid \overline{d}\})]\!]_\sigma$	$=$	$\texttt{class}\ C(\overline{z})\ \texttt{extends}\ [\![e_{opt}]\!]_\sigma\ \{\ x \mid [\![\overline{d}]\!]_{\sigma,x\mapsto C}\ \}$
$[\![\texttt{trait}\ M\ \texttt{with}\ \overline{k}\ \{x \mid \overline{d}\}]\!]_\sigma$	$=$	$\texttt{class}\ M(Z)\ \texttt{extends}\ \text{mix}_\sigma(\overline{k},\ Z)\ \{\ x \mid [\![\overline{d}]\!]_{\sigma,x\mapsto M}\ \}$
$[\![\texttt{val}\ f = t]\!]_\sigma$	$=$	$\texttt{object}\ f = [\![t]\!]_\sigma$
$[\![\texttt{def}\ m(\overline{z}) = t]\!]_\sigma$	$=$	$\texttt{object}\ m(\overline{z}) = [\![t]\!]_\sigma$
$[\![t.C(\overline{t})\ \texttt{with}\ \overline{k}]\!]_\sigma$	$=$	$\text{mix}_\sigma(\overline{k},\ [\![t]\!]_\sigma.C([\![\overline{t}]\!]_\sigma))$
$[\![e_{opt}\ \{x \mid \overline{d}\}]\!]_\sigma$	$=$	$[\![e_{opt}]\!]_\sigma\ \{x \mid [\![\overline{d}]\!]_\sigma\}$
$\text{mix}_\sigma(\epsilon,\ T)$	$=$	T
$\text{mix}_\sigma(\overline{k}, t.M,\ T)$	$=$	$[\![t]\!]_\sigma.M(\text{mix}_\sigma(\overline{k},\ T))$
$[\![\{x \mid \overline{d}\ t\}]\!]$	$=$	$\{\ x \mid [\![\overline{d}]\!]_\epsilon\ [\![t]\!]_\epsilon\ \}$

Figure 2.11: From CORE-SCALA to C-CAL

We start by treating anonymous functions, function applications and blocks as syntactic sugar, by applying a simple CORE-SCALA to CORE-SCALA transformation.

$$
\begin{array}{lcll}
[\![(\overline{x}) \Rightarrow t]\!] & = & \texttt{new}\ \{y \mid \texttt{def}\ \text{apply}(\overline{x}) = [\![t]\!]\} & (y \text{ fresh })\\
[\![t(\overline{t})]\!] & = & [\![t]\!].\text{apply}([\![\overline{t}]\!]) & \\
[\![\{x \mid \overline{d}\ t\}]\!] & = & (\texttt{new}\ \{x \mid [\![\overline{d}]\!], \texttt{val}\ \text{result} = [\![t]\!]\}).\text{result} &
\end{array}
$$

Functions are treated as objects containing a method `apply`. Function applications are treated as invocation of this method on the function arguments. Finally a block is considered as an object containing all the declarations of the block, plus an additional field `result` which holds the main expression of the block. This field is finally selected to compute the result of the block.

Then, we proceed with the translation with the additional assumption that the program no longer contains functions or blocks. The translation is summarized in Figure 2.11.

The translation is parameterized by a substitution σ that maps a variable x to the class C it is the self reference of. The substitution is used in only one place, for translating method super-calls: a super-call like $x.\texttt{super}.m(\overline{t})$ is translated in the parameterized super-selection $x.\texttt{super}[C].m([\![\overline{t}]\!]_\sigma)$.

The translation of mixin declarations and mixin applications also needs some explanations. The declaration of a mixin M is translated as a declaration of a class M which takes a template parameter Z. This parameter represents the abstract super-class of the mixin, which implies it must be part of the extends clause of M. The extends clause of M is computed by the expression $\text{mix}_\sigma(\overline{k},\ Z)$ which is basically Z, potentially mixed up with super-mixins $\overline{k}$. More generally, $\text{mix}_\sigma(\overline{k},\ T)$ represents the successive applications of mixins $\overline{k}$ to the template T; if $\overline{k}$ is the empty sequence, it returns just T. We take as example a class C that extends a class D while combining it with two mixins M_1 and M_2.

$$\texttt{class}\ C(x)\ \texttt{extends}\ t.D(x,\, x)\ \texttt{with}\ u_1.M_1,\ u_2.M_2\ \{y \mid \}$$

The result of the translation is the following.

$$\texttt{class}\ C(x)\ \texttt{extends}\ u_2.M_2(u_1.M_1(t.D(x,\, x)))\ \{\, y \mid \,\}$$

2.3.5 Comparing Mixins in Scala and Core-Scala

Ideally, our encoding of mixins should correspond to their implementation in the Scala compiler. On the majority of programs, it is actually the case. However there are some situations where there is a difference in the way our encoding and the Scala compiler linearize classes and mixins. In this section, we explain this difference and we give arguments in favor of the linearization performed by our encoding.

2.3.5.1 Linearization of classes and mixins

Our translation of Scala leads to a natural linearization of classes and mixins. Let us consider the following Scala program. The hierarchy of classes and mixins is represented in Figure 2.12.

```
class C1                   { def foo(): Int }
trait M1 extends C1        { def foo(): Int = 1 }
trait M2 extends C1 with M1
class C2 extends C1 with M1 { override def foo(): Int = 2 }
class C3 extends C2 with M2
(new C3).foo()
```

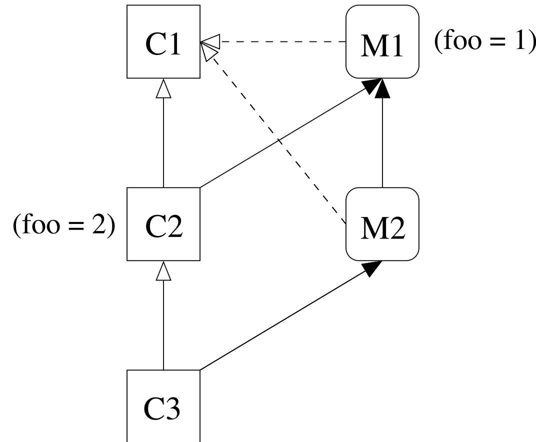


Figure 2.12: Example with a mixin inherited twice

In the diagram, classes are represented with square boxes and mixins with rounded boxes. There are three kinds of arrows between classes and mixins. Arrows with empty head and plain line represent inheritance between two classes. Arrows with empty head and dashed line represent the link between a mixin and

the required interface of its parent. Finally arrows with plain head represent the inheritance of a mixin by a class or another mixin.

Our translation linearizes this way: C3-M2-M1-C2-M1-C1. As we can see, there are two occurrences of the mixin `M1` in the linearization. In order to get this linearization we first remove all dashed arrows because they do no correspond to code inheritance, they just specify the interface of the parent of a mixin, which serves typing purposes only. Then we perform a depth-first and right-most traversal of the diagram starting from C3. We add each node that we encounter, even if it has already been traversed, like the mixin `M1` in our example, which is added twice. This linearization implies that a call of method `foo` on an instance of `C3` resolves in the value `1`, which is first found in mixin `M1`.

The SCALA compiler linearizes this way: C3-M2-C2-M1-C1. It traverses the diagram in the same order, but contrary to our linearization it takes into account dashed arrows and it always keeps the last occurrence of a mixin that has been encountered. In our example, it removes successively the first occurrences of `M1` and `C1` when it reaches them for the second time. With this linearization the lookup of `foo` returns the value `2`.

We claim our linearization is more natural and simpler than the one implemented in the SCALA compiler. Our linearization is natural because it comes as a natural interpretation of mixins as classes parameterized by their super-class. Our linearization is simpler because it does not deal explicitly with the problem of having multiple occurrences of a same mixin in the linearization, it just keeps all mixins in the linearization, letting left-to-right precedence decide on which declaration is finally chosen.

There is yet one advantage of the linearization performed by the SCALA compiler, it is always consistent with the static hierarchy of classes and mixins: each time a class C has a mixin M, C is located before M in the linearization order. This is not the case with our linearization since we have `M1` before `C2` in the linearization order even though class `C2` has `M1` as mixin.

2.3.5.2 Expansion of mixins in the SCALA compiler

In this section we explain how mixins are implemented in the SCALA compiler. The implementation is based on the linearization described in the previous section.

The SCALA compiler transforms a program with mixins into an equivalent program without mixins. We illustrate the translation principle through the following example. The idea is to split a mixin into an interface and a module. The module contains the implementations of all methods declared by the mixin. Each method is converted into a function that takes an extra parameter for representing the runtime value of the object resulting from the composition with this mixin. Inside these functions, super-calls are replaced by calls to abstract methods. It is impossible to provide an implementation to these methods because, at the mixin declaration site, we do not know which method to call, we just know that it has to be the next possible method in the class linearization when the mixin is used. At the use site of mixins (here class `C`), we know the complete linearization, so we can implement the abstract methods representing super-calls.

```
abstract class A {
```

```
  def foo(): Int = 3
}
trait M extends A {
  override def foo(): Int = 4 + super.foo()
}
trait N extends A {
  override def foo(): Int = 5 * super.foo()
}
class C extends A with M with N {}
(new C).foo()
```

Here is the code that is generated by the SCALA compiler.

```
class A {
  def foo(): Int = 3
}
trait M {
  def M_super_foo(): Int
  override def foo(): Int
}
trait N {
  def N_super_foo(): Int
  override def foo(): Int
}
class C extends A with M with N {
  override def foo(): Int = N_class.foo(C.this)
  def N_super_foo(): Int = M_class.foo(C.this)
  def M_super_foo(): Int = C.super.foo() // static super call
                                         // resolved at compile time
}
Console.println(new C.this().foo())

object M_class {
  override def foo(self: M): Int = 4 + self.M_super_foo()
}
object N_class {
  override def foo(self: N): Int = 5 * self.N_super_foo()
}
```

The transformation performed by the compiler to eliminate mixins relies on the property that it is possible to perform a static linearization of classes and mixins at the point where a mixin is used. Our encoding of mixins does not rely on this condition.

2.3.6 Class Combination

In GBETA classes and methods are unified under the concept of pattern. An interesting aspect of GBETA patterns is that they combine automatically and at all depths [14], as soon as they make contact with a homonymous pattern. In GBETA methods are composed using the keyword INNER that represents the implementation of the current method in the direct subclass of the current class. When translating CORE-SCALA we already presented a mechanism for combining methods based on super-calls. Both mechanisms are similar but

inverse: in one case (SCALA) we can refer to an implementation defined in the direct super-class, in the other case (GBETA) we refer to an implementation defined in the direct subclass. In this section we propose an extension of CORE-SCALA for combining classes which can still be encoded in the basic calculus.

2.3.6.1 Virtual classes

We extend the syntax of CORE-SCALA with two new kinds of declarations for virtual classes. The first one is similar to a normal class declaration and corresponds to the first implementation of a virtual class.

$$\texttt{virtual class}\ C(\overline{x})\ \texttt{extends}\ p$$

The second kind of declaration corresponds to the overriding of an existing virtual class. Like mixin declarations, overriding class declarations have no explicit parent, because they implicitly inherit from the class they override.

$$\texttt{override class}\ C(\overline{x})\ \texttt{with}\ \overline{k}\ \{x \mid \overline{d}\}$$

Note that it is required for all implementations of a same virtual class to have exactly the same number of arguments. It is the task of a type-system to enforce such policies.

2.3.6.2 Class overriding

We have insisted on the fact that one basic property of classes in SCALA and C-CAL is that they cannot be overridden. If they were we could simply combine classes by extending an overriding class with a super-call to the same class, like in the following example.

```
class A { this |
  class C(z) ... { ... }
}
class B extends root.A { this |
  class C(z) extends this.super.C(z) with ... { ... }
}
new (new B).C(v)
```

Fortunately it is possible to encode classes that can be overridden using templates: each implementation of such a class is given a unique name and original class declarations become template declarations which are just aliases for the class implementations. With this design pattern, the previous example is translated as follows.

```
class A { this |
  class C$A(z) ... { ... }
  template C(z) = C$A(z)
}
class B extends root.A { this |
  class C$B(z) extends this.super.C(z) with ... { ... }
  template C(z) = C$B(z)
}
new (new B).C(v)
```

2.4 Scaletta

The C-cal calculus aims to be a simple target language for translating Scala programs; for this purpose it contains the concept of class as a primitive and it has a simple and natural interpretation in MoC, our general model of computation for object-oriented languages. However, C-cal is not completely satisfactory as a class-based calculus. The reason is simple: values are isomorphic to class-labeled trees, but there exists no primitive in C-cal corresponding to the selection of a subtree. Actually, adding such a primitive to the calculus allows to simplify the treatment of self references and to simplify the lookup of a declaration in a template. The result of this last improvement is a calculus that we call Scaletta.

2.4.1 Tree Interpretation of Values

In C-cal object and template values are defined by the following abstract syntax.

$$\begin{array}{lcl} v & ::= & x \mid \texttt{new}\, V \\ V & ::= & v.C :: V_{opt} \end{array}$$

A template value V is isomorphic to a list of pairs $(v,\, C)$ whose first component is an object value v and second component is a class name C. It implies that object values can be represented as trees labeled with class names. Leaf nodes of such trees correspond to the object value x, which represents the implicit root object of a program. Internal nodes correspond to object values $\texttt{new}\, V$: if V is isomorphic to the list $((v_1,\, C_1),\, \cdots,\, (v_n,\, C_n))$, the node will have n outgoing edges, labeled with class names C_i's, and leading to the tree representations of values v_i's. The inverse interpretation of trees as object values is also clear: if an object value v has a child w labeled with C it means that v is an instance of class C and that its C enclosing instance is w.

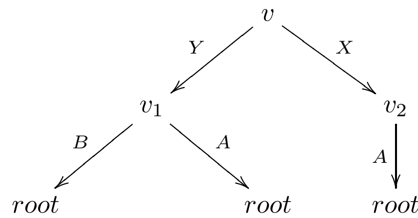
As an example, let us find the tree representation of the value returned by the following program.

```
{ root |
  class A {
    class X
  }
  class B extends root.A {
    class Y extends (new root.A).X
  }
  new (new root.B).Y
}
```

The result of this program is the object value v such that

$$\begin{array}{lcl} v & = & \texttt{new}\, ((v_1,\, Y),\, (v_2,\, X)) \\ v_1 & = & \texttt{new}\, ((root,\, B),\, (root,\, A)) \\ v_2 & = & \texttt{new}\, ((root,\, A)) \end{array}$$

Its tree representation is given below. The tree representation reveals the two dimensions that exist in the world of classes, namely class inheritance and class nesting. All classes that lead to sibling nodes of the tree are in a subclass relationship; in our example Y is a subclass of X, which is reflected by the fact that Y and X are edges coming from the same node. The nesting of classes can be recovered by following some branches of the tree. More precisely, if a class is the label of an edge at one level, its enclosing class appears among the classes labeling edges at the level just below; in our example, the branch $v \xrightarrow{Y} v_1 \xrightarrow{B} root$ reflects the fact that Y is enclosed in class B, and the branch $v \xrightarrow{X} v_2 \xrightarrow{A} root$ reflects that class X is enclosed in class A.



If an object value v is a tree labeled with class names, it is natural to speak of the subtree corresponding to a label C. By analogy with the Unix file system we can use the notation v/C to represent this subtree. In our example v/Y represents the value v_1 and $v/Y/A$ represents the value $root$. Surprisingly, whereas it seems that the selection of a subtree is a basic operation of C-CAL, there is no corresponding primitive in the calculus. In this section we give an alternate presentation of C-CAL where v/C is a primitive operation. Some simplifications arise from the addition of this operator which justify it status of primitive: by replacing self references with applications of the new operator we are able to delay the time where a substitution must take place. For instance, without self references no substitution has to intervene during the lookup of a declaration. We claim that a type system can also benefit from this property and type a larger set of programs.

2.4.2 Syntax

The addition of a mechanism for selecting outer instances to C-CAL leads to the definition of SCALETTA. The syntax of SCALETTA is summarized in Figure 2.13.

Note that this syntax is not just C-CAL with the addition of an operator t/C. It also includes some natural simplifications implied by its introduction. All simplifications come from the observation that references to current instances of enclosing classes can be replaced by successive applications of the operator / on the current instance. For example, in the following program, the occurrence of a in the definition of foo can be replaced with $c/C/B$.

```
{ root |
  class A { a |
    class B { b |
      class C { c |
        object foo = a.bar
```

Class name	C			
Template label	L			
Object label	l			
Owner	O	::=	`Root`	root
		|	C	class
Template	T, U	::=	$t.L$	template selection
		|	$t.C$	class template
		|	$t.C :: T_{opt}$	extended template
Object	t, u	::=	`this`	current instance
		|	$t.l$	object selection
		|	t/C	outer object selection
		|	`new` T	instance creation
Declaration	d	::=	`class` C `extends` T_{opt}	class
		|	`template` $L = T$	template field
		|	`object` $l = t$	object field
Anchored declaration	D	::=	d `in` O	
Program	P	::=	$\overline{D}\ t$	
Object value	v	::=	`this` | `new` V	
Template value	V	::=	$v.C :: V_{opt}$	
Evaluation context	Γ	::=	$\overline{D}$	

Figure 2.13: Syntax of SCALETTA

```
    }}}
}
```

If we can consider that self references never designate the current instance of an enclosing class, alpha-renaming of these variables is no longer needed, because no name capture is possible with binders that are never nested. It means that we can assume that all self variables have by convention the name `this`, which becomes a keyword of the calculus.

Also, if class declarations no longer refer to a binder defined outside, they somehow become relocatable. More precisely, it is possible to flatten all class declarations, and more generally all declarations, as long as we keep the information of their enclosing context. An enclosing context, or owner, O is either a class name C or `Root` for top-level declarations. A declaration d together with its owner O is called an anchored declaration and is noted d `in` O. Finally, a SCALETTA program $\overline{D}\ t$ is just a list of anchored declarations $\overline{D}$, together with a main term t.

Object and template values of SCALETTA are similar to those of C-CAL; the only difference is in the representation of the root instance: in C-CAL a self variable x is used, in SCALETTA it is replaced with `this`.

We say that a SCALETTA program $\overline{D}\ t$ is well-formed if for each occurrence of a label or class name inside the declarations $\overline{D}$ and the term t, there exists a corresponding declaration in $\overline{D}$. Furthermore for a given class name C there can only be one declaration in $\overline{D}$.

$\mathrm{self}_\Gamma(x)$	$=$	$\mathrm{self}_\Gamma(x, \mathtt{this})$
$\mathrm{self}_{\Gamma,x\mapsto C}(x,t)$	$=$	t
$\mathrm{self}_{\Gamma,y\mapsto C}(x,t)$	$=$	$\mathrm{self}_\Gamma(x, t/C) \qquad \text{if } x \neq y$
$[\![x]\!]_\Gamma$	$=$	$\mathrm{self}_\Gamma(x)$
$[\![t.l]\!]_\Gamma$	$=$	$[\![t]\!]_\Gamma.l$
$[\![t.\mathtt{super}[C].l]\!]_\Gamma$	$=$	$[\![t]\!]_\Gamma.\mathtt{super}[C].l$
$[\![\mathtt{new}\ T]\!]_\Gamma$	$=$	$\mathtt{new}\ [\![T]\!]_\Gamma$
$[\![t.L]\!]_\Gamma$	$=$	$[\![t]\!]_\Gamma.L$
$[\![t.\mathtt{super}[C].L]\!]_\Gamma$	$=$	$[\![t]\!]_\Gamma.\mathtt{super}[C].L$
$[\![t.C]\!]_\Gamma$	$=$	$[\![t]\!]_\Gamma.C$
$[\![t.C :: T_{opt}]\!]_\Gamma$	$=$	$[\![t]\!]_\Gamma.C :: [\![T_{opt}]\!]_\Gamma$
$[\![\mathtt{object}\ l = t]\!]_\Gamma$	$=$	$(\mathtt{object}\ l = [\![t]\!]_\Gamma)\ \mathtt{in}\ O \qquad \text{if } \Gamma = \Gamma', x \mapsto O$
$[\![\mathtt{template}\ L = T]\!]_\Gamma$	$=$	$(\mathtt{template}\ L = [\![T]\!]_\Gamma)\ \mathtt{in}\ O \qquad \text{if } \Gamma = \Gamma', x \mapsto O$
$[\![\mathtt{class}\ C\ \mathtt{extends}\ T_{opt}\ \{\ y \mid \overline{d}\ \}]\!]_\Gamma$	$=$	$(\mathtt{class}\ C\ \mathtt{extends}\ [\![T_{opt}]\!]_\Gamma)\ \mathtt{in}\ O,\ [\![\overline{d}]\!]_{\Gamma,y\mapsto C}$ $\text{if } \Gamma = \Gamma', x \mapsto O$
$[\![\{\ x \mid \overline{d}\ t\ \}]\!]$	$=$	$[\![\overline{d}]\!]_{x\mapsto\mathtt{Root}}\ [\![t]\!]_{x\mapsto\mathtt{Root}}$

Figure 2.14: From C-CAL to SCALETTA

2.4.2.1 Elimination of self variables

The translation from C-CAL to SCALETTA is formally defined in Figure 2.14. The translation is parameterized by an ordered list Γ of bindings that associate a self reference x with an owner O.

An inverse translation from SCALETTA to C-CAL is also possible. The only difficulty is in the translation of outer selections t/C. The idea is to add to each class C an object field l_C that is initialized with the value of the direct enclosing instance. This way an outer selection t/C can simply be replaced with the normal field selection $t.l_C$. As an example the SCALETTA program

```
object foo = this.foo
class A {
  object bar = this/A.foo
}
```

is translated into the following C-CAL program.

```
{ root |
  object foo = root.foo
  class A { a |
    object l$A = root
    object bar = this.l$A.foo
  }
}
```

2.4.2.2 Link with the de Bruijn notation

The idea of replacing C-CAL self variables with chained outer instance selections is connected to the de Bruijn notation for the lambda-calculus. In this

notation, variables are replaced with integers that represent the number of λ's to traverse before finding the λ corresponding to the declaration of the variable. For instance the lambda-term $\lambda x.\lambda y.x\ y$ is written $\lambda.\lambda.1\ 0$ in the de Bruijn notation.

In order to explain the analogy, we have to name lambdas in the lambda-term $\lambda x.\lambda y.x\ y$; let us call the first one f and the second one g. Because the second lambda is nested inside the first one, every call of the function g is done in the context of a call of the function f. In other words, every activation frame of g is nested inside one of f. So, to get the argument x of f from g, we follow the following procedure: we start from the activation frame of g, we escape it in order to get the activation frame of f and finally we select the argument of this last activation frame. This can be expressed with the SCALETTA term $\texttt{this}/g.arg$. In this term `this` represents the current activation frame, $\texttt{this}/g$ represents the enclosing activation frame of g, namely the one of f, and finally $\texttt{this}/g.arg$ represent the argument of $\texttt{this}/g$, that is the argument of f. Similarly the argument y of g can be expressed inside g by the SCALETTA term $\texttt{this}.arg$.

The SCALETTA terms $\texttt{this}/g.arg$ and $\texttt{this}.arg$ are to be put together with the notations 1 and 0 for representing the arguments of f and g from inside g in the de Bruijn notation. The SCALETTA notation can be seen as a generalization of the de Bruijn notation. The fact that the de Bruijn notation is more compact is that it makes some simplifying assumptions: first there is no inheritance between functions, which has as consequence there is always a single way of escaping an activation frame and that using function names is useless. Second, there is only one argument per λ whereas classes can have many fields, which means that in the de Bruijn notation the name of the argument is useless. If both the name of the activation frame to escape and the name of the argument are useless, it is possible for the de Bruijn notation to merge the operations of finding an activation frame and selecting its argument.

2.4.3 Semantics of SCALETTA

As for all semantics presented until now the semantics of SCALETTA is based on two kinds of relations: declaration lookup and reduction. In SCALETTA an evaluation context Γ consists of the set of anchored declarations of the program to evaluate.

The declaration lookup relation is defined in 2.16. It maps an object value to the set of its accessible members. In case the value is `this` the list of top-level declarations is returned. For values of the form $\texttt{new}\ V$, the auxiliary function $\text{members}_\Gamma(V)$ is used. It takes a template value V and returns the list of members contained in this template. It is defined as follows, where the operator $+$ still designates the concatenation of two lists of declarations with precedence given to the declarations of the second list.

$$\text{members}_\Gamma(v.C :: V_{opt}) \quad = \quad \text{members}_\Gamma(V_{opt}) + (\text{decls}_\Gamma(C),\ \texttt{outer}\ C = v)$$

As we can see, pseudo-declarations of the form $(\texttt{outer}\ C = v)$ are added to this set and represent values of outer instances. In this definition, $\text{decls}_\Gamma(C)$ represents the set $\{\, d \mid (d\ \texttt{in}\ C) \in \Gamma \,\}$.

What is remarkable in the definition of declaration lookup for SCALETTA is that it just returns a list of declarations as they appear in the original program. The use of the operator t/C allows to avoid the substitution for current enclosing instances; the only required substitution is the one for the current instance, which is represented by `this` in SCALETTA. With SCALETTA, the replacement of enclosing instances with their values is done neither during the reduction of template selections as in MoC, neither during the lookup of a declaration as in C-CAL, but only when an enclosing instance is used in the computation.

The reduction relations for objects and templates are summarized in 2.15. A selection of an object field l is resolved by searching the corresponding declaration in the prefix t and substituting t for `this` in the result u of the search. Compared to C-CAL there are also additional rules for reducing an outer field selection t/C. Actually an outer field selection is almost treated as a normal field selection except no substitution needs to be applied to the result of the lookup.

2.4.3.1 Super-selections

The primitives for super-selections in SCALETTA are the same as the ones in C-CAL, namely $t.\texttt{super}[C].l$ for object super-selections and $t.\texttt{super}[C].L$ for template super-selections. The definition of their semantics uses the following function $\text{membersSuper}_\Gamma(C, V)$ which returns the members found *after* class C in the template V.

$$\begin{array}{lcll} \text{membersSuper}_\Gamma(C, v.C :: V_{opt}) & = & \text{members}_\Gamma(V_{opt}) & \\ \text{membersSuper}_\Gamma(C', v.C :: V_{opt}) & = & \text{membersSuper}_\Gamma(C', V_{opt}) & \text{if } C \neq C' \end{array}$$

The rule (OR-SUPER) for reducing an object super-selection is given in Figure 2.17.

2.5 Conclusion

2.5.1 Summary

In this chapter we have equipped an untyped functional subset of SCALA, called CORE-SCALA, with a semantics. Rather than giving a direct ad-hoc semantics we preferred to define a translation from CORE-SCALA to a more general language. In the quest of this language, we have identified a very general model of computation for object-oriented languages that we called MoC. MoC models the object-oriented concepts of field selection, self recursion and code inheritance but it does not have classes as primitives. Because classes are going to play an important role in the typing of a language like SCALA, we have inserted a class-base calculus, called C-CAL, in the translation from CORE-SCALA to MoC. Finally we have argued that this class-based calculus is similar to a simpler one, called SCALETTA, where the selection of an outer instance is a primitive of the language.

In the global picture of this thesis, which has for main goal to prove that SCALA is safe, the definition of a semantics for SCALA is only a first step towards this goal. A first validation of our semantics would be to show that SCALETTA

$$\text{(OR-SELECT)}\ \frac{\Gamma \vdash t \ni (\texttt{object}\ l = u)}{\Gamma \vdash t.l \rightarrow u[\texttt{this}\backslash t]}$$

$$\text{(TR-SELECT)}\ \frac{\Gamma \vdash t \ni (\texttt{template}\ L = T)}{\Gamma \vdash t.L \rightarrow T[\texttt{this}\backslash t]}$$

$$\text{(TR-EXTENDS)}\ \frac{\Gamma \vdash t \ni (\texttt{class}\ C\ \texttt{extends}\ T_{opt})}{\Gamma \vdash t.C \rightarrow t.C :: T_{opt}[\texttt{this}\backslash t]}$$

$$\text{(OR-OUT)}\ \frac{\Gamma \vdash t \ni (\texttt{outer}\ C = u)}{\Gamma \vdash t/C \rightarrow u}$$

$$\text{(OR-CSELECT)}\ \frac{\Gamma \vdash t \rightarrow u}{\Gamma \vdash t.l \rightarrow u.l} \qquad \text{(OR-COUT)}\ \frac{\Gamma \vdash t \rightarrow u}{\Gamma \vdash t/C \rightarrow u/C}$$

$$\text{(TR-CCLASS)}\ \frac{\Gamma \vdash t \rightarrow u}{\Gamma \vdash t.C \rightarrow u.C} \qquad \text{(OR-NEW)}\ \frac{\Gamma \vdash T \rightarrow U}{\Gamma \vdash \texttt{new}\ T \rightarrow \texttt{new}\ U}$$

$$\text{(TR-CSELECT)}\ \frac{\Gamma \vdash t \rightarrow u}{\Gamma \vdash t.L \rightarrow u.L}$$

$$\text{(TR-CEXTENDS)}\ \frac{\Gamma \vdash T \rightarrow U}{\Gamma \vdash t.C :: T \rightarrow t.C :: U}$$

Figure 2.15: Reduction of objects ($\Gamma \vdash t \rightarrow u$) and templates ($\Gamma \vdash T \rightarrow U$) in SCALETTA

$$\text{(L-ROOT)}\ \frac{(d\ \texttt{in Root}) \in \Gamma}{\Gamma \vdash \texttt{this} \ni d} \qquad \text{(L-NEW)}\ \frac{d \in \text{members}_\Gamma(V)}{\Gamma \vdash \texttt{new}\ V \ni d}$$

Figure 2.16: Declaration lookup ($\Gamma \vdash t \ni d$) in SCALETTA

$$\text{(OR-SUPER)}\ \frac{t = \texttt{new}\ V \quad (\texttt{object}\ l = u) \in \text{membersSuper}_\Gamma(C,\ V)}{\Gamma \vdash t.\texttt{super}[C].l \rightarrow u[\texttt{this}\backslash t]}$$

Figure 2.17: Super-selections in SCALETTA

makes it easy to define an expressive type system that can be implemented efficiently. We can consider that this work has already been done in Philippe Altherr's thesis [2] because it contains a typed-intermediate language for SCALA that can be seen as a typed version of SCALETTA. The second step of validation would then be to prove that the type system is sound. In the next chapter we approach this question by focusing on the proof that virtual types, an essential feature of the SCALA type system, are safe.

2.5.2 Related Work

2.5.2.1 Comparing MoC and ς-calculus

In our model of computation we make a syntactic difference between objects and templates. An even simpler calculus can be obtained by unifying both concepts: this would imply to remove the construct `new` T and to only have one kind of labels, one kind of declarations, one kind of selections, and one rule for reducing a selection. The resulting calculus is very similar to the ς-calculus, the basic untyped calculus of objects described in the book "A Theory of Objects" [1]. We briefly remind its syntax and semantics in order to compare it with MoC.

Labels	l			
Terms	t	$::=$	x	variable
		$\mid$	$[\overline{d}]$	object formation
		$\mid$	$t.l$	field selection/method invocation
		$\mid$	$t.l \leftarrow \varsigma(x)\, u$	field update/method update
Fields	d	$::=$	$l = \varsigma(x)\, t$	

There are two redexes in the ς-calculus : field selections and field updates. In the reduction rule for field updates, $\overline{d}\backslash l$ represents the set of fields $\overline{d}$, minus the field associated with label l, when it exists.

$$
\begin{array}{llll}
t.l & \rightarrow & u[x\backslash t] & \text{if } t = [\overline{d}] \text{ and } (l = \varsigma(x)\, u) \in \overline{d} \\
t.l \leftarrow \varsigma(x)\, u & \rightarrow & [\overline{d}\backslash l,\, l = \varsigma(x)\, u] & \text{if } t = [\overline{d}]
\end{array}
$$

One contribution of our work w.r.t. the ς-calculus is to separate the concepts of objects and templates at the level of the syntax. We think it is confusing to unify both concepts. The unification leads to a model for prototype-based languages, like SELF [24]. But this model does not extend naturally to take into account mutable fields, because mutable fields are conceptually part of objects, not templates. The following confusing questions arise if we unify both concepts: what happens if we extend a prototype that has mutable fields? Do we also inherit the values of these fields, do we have the right to assign them new values? A classical error in SELF is to forget to clone an object before extending it.

We can trivially encode the ς-calculus in a variant of MoC where templates and objects are unified: an object $[l_1 = \varsigma(x)\, t_1, \cdots, l_n = \varsigma(x)\, t_n]$ becomes the MoC object/template $\{x \mid \texttt{object}\ l_1 = t_1, \cdots, \texttt{object}\ l_n = t_n\}$, a variable stays a variable, a field selection stays a field selection, and a field update $t.l \leftarrow \varsigma(x)\, u$ becomes a compound object/template $t \,\&\, \{x \mid \texttt{object}\ l = u\}$. As we can see, a field update in the ς-calculus corresponds to a very restrictive

use of the template combination $T \& U$ in MoC, where the first template T is always a selection and the second template U is always an atomic template with just one field.

The authors of the ς-calculus suggest an encoding of static super-calls as found in JAVA, but they do not try to model a semantics of dynamic super-calls, as we do with MoC. In order to define a semantics of dynamic super-calls as they exist in SCALA, it is convenient that declarations can be organized in groups. This way, a super-call inside a group simply means a call whose lookup starts from the next group in some dynamic linearization of groups. MoC has this possibility of grouping declarations through atomic templates $\{x \mid \overline{d}\}$. The ς-calculus is able to express one group with the construct $[\overline{d}]$, but it fails to express a linearization of groups, because the only way of combining groups in the ς-calculus is through a method update and method updates just contribute singleton groups to the linearization.

2.5.2.2 Lambda-calculi with records

Another tradition consists in encoding objects and classes in an extension of the lambda-calculus with records and record selections. The best-known encodings are called the *self-application semantics* and the *recursive-record semantics*.

The self-application semantics corresponds to the traditional implementation of object-oriented languages, like JAVA for instance: an object is seen as a record of functions. Each function of this record corresponds to a method of the object and takes an extra parameter that represents the object itself. The call $t.m(\overline{t})$ of a method m with arguments $\overline{t}$ on a receiver object t is transformed in the function application $(t.m)(t, \overline{t})$, i.e. an application of the function $t.m$ with first argument t itself, followed by the method arguments $\overline{t}$. This encoding is very simple and is all what is necessary for an implementation. However, it is not a good candidate for the design of a type system because it places the current object in a contravariant position when encoding methods. Consequently it forbids the overriding of a method in a subclass. In the recursive-record semantics, an object is seen as a recursive record $\mu(x)\, r$. We remind that recursive terms like $\mu(x)\, t$ can be encoded in the lambda-calculus after defining a fixed-point operator.

The various semantics for classes and objects based on the lambda-calculus suffer the same problem as MoC, our general model of computation for object-oriented languages: because classes are not a primitive of the language, they cannot be used in the definition of a type system. It means that class types must be encoded by the combination of several different type constructs, like recursive types, existential types and universal types, which have complicated theories separately and a fortiori when they are used together. [5] is a good overview of the existing encodings. In this thesis we present a class-based calculus, called SCALETTA, which is supposed to simplify the definition and the proof of a type system.

Chapter 3

A Soundness Proof of Virtual Types

3.1 Introduction

In the effort of adding genericity to object-oriented languages, virtual types have been proposed as a replacement for type parameterization [21]. And indeed, there exists a general encoding that lets one express all kinds of parameterized types as virtual types [2]. A virtual type is a class member holding a type. As for virtual methods, whose exact implementation depends on the object on which they are called, the interpretation of a virtual type depends on the actual object on which it is selected. Virtual types are considered the most natural foundation for type abstraction in the SCALA programming language [20]. This chapter is about a completely formal proof of type safety for a calculus with virtual types. We resisted the temptation to design a very expressive and complex calculus whose soundness would rest only on informal arguments and we instead concentrate on a smaller calculus with a limited expressiveness but whose safety cannot be questioned. In the design of the calculus and the proof of soundness of its type system we have been guided by the objective of ultimately implementing our work in the COQ proof assistant [4]. Some unusual choices of representations in the calculus and the abnormally high level of detail reached in the proof must be understood in the light of this objective.

3.1.1 Extending Featherweight Java with Virtual Types

Except for the syntax, that we wanted close to SCALA, the calculus on which we develop our soundness proof resembles Featherweight Java [17] (FJ) on many points. FJ is a formalization of JAVA that captures some central aspects of the whole language and that is frequently used to model and prove the validity of its extensions. Like FJ, our calculus is a stateless class-based object-oriented calculus with fields, methods, single inheritance, and top-level classes only. In both calculi, the value of a field cannot refer to the current instance of the class to which it belongs, rather it should be possible to compute the value of a field before invoking the creation of an object. Furthermore in both cases the semantics is given as a small-step operational semantics without modeling of

the heap, which means objects have no identity.

Our calculus adds the concept of virtual types to FJ by several simple syntax extensions. FJ has only concrete types, called *class types*, which are of the form C and that denote all instances of a class C. In our calculus, a type can also be a *virtual type* $p.L$, also called abstract type, which represents the value of the type field L held by the object denoted by the path p. *Paths* are the subcategory of terms that are allowed inside types. Roughly speaking, they are composed of variables and field selections performed on terms that are themselves paths. The introduction of paths is motivated by their property of being strongly normalizable and of always evaluating to the same object value; both properties are needed if we want type soundness to hold.

There are also two additional kinds of member declarations that are related to virtual types: a declaration $\texttt{type}\ L >: T <: U$ defines a new virtual type L with a lower-bound T and an upper-bound U, a declaration $\texttt{type}\ L = T$ assigns the value T to the virtual type L in all instances of the current class.

Amongst terms, the calculus adds the possibility of introducing local immutable variables with the construct $\texttt{let}{:}T = t\ \texttt{in}\ u$ which defines a new variable of type T, initialized with value t, and that can be used in u using its de Bruijn's index [12]. The adding of local variables is not just a matter of convenience: they are used to obtain valid prefixes of virtual types. Remember that the prefix p of a virtual type $p.L$ has to be a path; as it will be explained latter this constraint propagates to the receiver object of a method call and to the term on which a field is selected. With local variables, an ill-formed chaining of method calls like $\texttt{this}.f().g()$ can be replaced with the valid term $\texttt{let}{:}T = \texttt{this}.f()\ \texttt{in}\ 0.g()$ in which all method calls are performed either on the current instance `this` or on the local variable 0, which are paths by definitions. They are various approaches to represent the concept of bound variables in a mathematical way. The approach that consists in regarding terms modulo alpha-conversion of the names they contain is usually prefered in a semi-formal context, like a scientific paper, but if we want to be completely formal, de Bruijn's indices tends to be more practical.

There are only two features of FJ that we do not keep in our calculus: method overriding and type casts. There are also some other small dissimilarities like our formal treatment of the current instance of a class – which is a reserved word and not a variable –, and our treatment of variables and method parameters.

3.1.2 Comparison with Featherweight Generic Java

In the previous paragraph we have compared our calculus with FJ. Now we compare it with Featherweight Generic Java [17] (FGJ), the extension of FJ with generic classes and generic methods. The goal is to show that both calculi are not directly comparable. First of all, our calculus is unable to encode parameterized class types as found in FGJ. For instance, in order to encode a parameterized type like `List<Int>`, which is the FGJ type of integer lists, our calculus lacks the possibility of enriching class types with constraints on virtual types. With such a mechanism, the above type would be encoded by the class type with refinements `List{type Elem = Int}`, which represents the set of instances of `List` in which the virtual type `Elem` has the value `Int`.

But there are also programs of our calculus that are not translatable in FGJ,

like the one below. The usual translation scheme from virtual types to genericity consists in transforming virtual type declarations into type parameters [6]. But it does not work in this case. It is like if the class `List` below was parameterized by an infinity of types, one for each element in the list. Let us see why: if we transform `X` as a type parameter of class `List`, then the occurrence of `List` in the type of `tail` must receive a type argument. As we do not know it, because it is left abstract, we must pass it from outside, i.e. add another type parameter `X2` to the class, etc.

```
class List {
  type X
  val head: this.X
  val tail: List
}
```

We could argue that the occurrence of `List` in the type of `tail` could be translated by `List[?]` in an extension of FGJ with type wildcards [23]. But it is an open question if this trick can be applied in all situations. We consider a more complicated example where a virtual type is selected on a field selection [1].

```
class List {
  type X
  val head       : this.X
  val tail       : List
  val elemOfTail : this.tail.X
}
```

In order to translate this program, there are certain choices that are imposed to us. For instance, for translating the type `this.tail.X`, which is a type abstraction, we have no other choice than using a type parameter `Y`. This type parameter can not stay free in the class `List`, it must become a type parameter of this class. At the end the class `List` must declare at least two type parameters: one for simulating the virtual type `X`, as in our previous example, and one for representing the type `this.tail.X`. The occurrence of `List` in the type of `tail` must consequently receive two type arguments. The first one is of course `Y` because it has been created for that. For the second one it is sufficient to use a wildcard. The final result of the translation is the following code.

```
class List[X,Y] {
  val head       : X
  val tail       : List[Y,?]
  val elemOfTail : Y
}
```

There is one main lesson we can draw from this example. The translation scheme forces us to have more type parameters than we had virtual types. Actually, to one virtual type corresponds as many type parameters as occurrences

[1] Note that types like `this.tail.X` are not formalized in our basic calculus for which we have a formal soundness proof, they are left as future work.

of this virtual type with different prefixes in the program. For instance in our example we had two occurrences of the virtual types `X` with different prefixes: `this.X` and `this.tail.X`. And we ended up with two type parameters: `X` and `Y`. This observation makes the encoding unsatisfactory because it forces us to pollute the interface of classes with synthetic type parameters. It is also inherently not compositional because it requires a global analysis of the program in order to find all the occurrences of a virtual type. In conclusion it is an open question if we can translate any program with virtual types into a program with just type parameterization and wildcards. But it is a closed question that such a translation cannot be compositional.

The conclusion of comparing FGJ and our calculus is that both calculi are not directly comparable in terms of expressiveness, i.e. it is not possible to encode simply one calculus into the other. This justifies our investment in the proof of a calculus with virtual types, otherwise we could always be reproached that there already exist convincing type safety proofs for more general calculi.

3.1.3 Overview

The rest of this chapter is structured as follows. Section 3.2 explains the subtleties behind the behind the design of subtyping rules for virtual types and present informally our solutions to have a safe type system. Section 3.3 introduces the syntax of our calculus, Section 3.4 its semantics, Section 3.5 its typing rules, and Section 3.6 the well-formedness of its programs. In Section 3.7 we present the concept of structured subtyping in detail. In Section 3.8 we discuss the compatibility that should exist between the two bounds of a virtual type. Section 3.9 presents briefly the proof of soundness, which is completely detailed in Appendix A. Finally, Section 3.10 presents future and related work and concludes.

3.2 Sound Subtyping

In this section we informally explain the subtleties behind the design of subtyping rules in presence of virtual types. In particular, we show that the naive rules that immediately come to mind are unfortunately unsound in the sense that they are intrinsically incompatible with type safety. After two successive refinements of these rules we end up with a solution that is twofold: first we impose a well-founded relation $\prec$ on type symbols that is consistent with type declarations, second we constrain the transitivity rule for subtyping such that the intermediate type is smaller, with respect to the relation $\prec$, that the types of the conclusion. We call the resulting subtyping relation *structured subtyping*.

3.2.1 Subtyping and Subclassing

The design of a type system requires to make a certain number of choices. Such choices are guided by the objective of eliminating programs that potentially endanger type safety. For an object-oriented calculus the potential runtime errors that must be prevented are: the selection on an object of a field that does not exist in the object, the call on an object of a method that does not exist in the object, and the call of a method with a wrong number of arguments.

Our approach for designing a safe type system consists in characterizing some general classes of problems and in finding solutions to these problems, with the hope that we have covered this way all dangerous cases. We think that, in the context of a class-based calculus with virtual types, most of the problems come down to the following question: *does subtyping implies subclassing?* More precisely, it should be true that if two classes A and B are such that the class type A is a *subtype* of the class type B, then A is also a *subclass* of B. This fact is trivially true in a calculus like FJ in which all types are class types and in which, consequently, subtyping and subclassing are not even distinguished. But this fact is less obvious in a setting with virtual types, in which the proof that A is a subtype of B might go through a virtual type $p.L$ (for instance if L has A as lower-bound and B as upper-bound).

But first, let us see why a type system where subtyping does not imply subclassing is inevitably unsafe. This is illustrated by the following example.

```
class A { }
class B {
  def m(): B = new B()
}
new A().m()
```

In this example we have two independent classes A and B such that one, B, declares a method m, while the other, A, does not. Suppose then that the type system lets us deduce that A is a subtype of B. The expression `new` $A().m()$ is well-typed because method m is declared in class B, and B is a super-type of the type A of the receiver object `new` $A()$. Although the expression `new` $A().m()$ is well-typed, it causes an error at runtime because no method m is actually inherited from A.

Note that we still have a problem if we change the hypothesis from "A is a subtype of B" to "there exists a type T such that A is a subtype of T and T is a subtype of B", in a context where subtyping is not transitive for all types. The construction of a problematic example is less direct, but still simple. It consists in assembling the two subtyping steps, from A to T and from T to B, by composing two coercion functions, `id1` and `id2`, that individually perform just one of theses steps. The resulting function `id`, defined below, lets us see any instance of class A as an instance of class B. As a consequence, the unsafe expression $id($`new` $A()).m()$, which reduces to the stuck evaluation state `new` $A().m()$, is well-typed.

```
def id1(x: A): T = x
def id2(x: T): B = x
def id (x: A): B = id2(id1(x))
```

Now we are going to use these observations to see what kinds of programs must be eliminated by the type system.

3.2.2 Graph of Symbols

In the course of our argumentation, we will consider several program examples containing declarations of classes and of virtual types, and we will base our analysis on the relations that exist between their associated symbols. In order

to make clearer the relation that links type symbols and class symbols, we reason on abstract representations of programs as finite directed graphs. In such graphs, the nodes are the type and class symbols that appear in the program and they are represented by circles. There is an arrow (directed edge) from one symbol to another if there is a declaration that links both symbols together. An arrow is labeled by the kind of the associated declaration: $<:$ is used between a type symbol and the symbol of its upper-bound, or between a class symbol and its superclass; $>:$ is used between a type symbol and the symbol of its lower-bound; finally $=$ is used between a type symbol and the symbol of the type it is an alias for. Note that there can be more than one edge labeled with $=$ starting from a same type symbol since a type symbol can be assigned different values in different classes.

As an illustration of this view of programs, consider the graph representation of the following program in Figure 3.1.

```
class B {
  type T <: B
}
class A extends B {
  type T = A
}
```

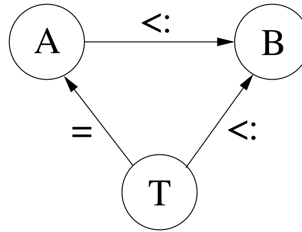


Figure 3.1: Graph representation of a program

Note that, for the sake of clarity, we do not write the prefix of a virtual type in the graph: in our example the node T of the graph actually represents all the types of the form $p.T$, and in particular the type `new` $C().T$. In the rest of this section, we often refer to *the symbol of a type*, it is defined as follows: the symbol of a class type C is C, and the symbol of a virtual type $p.L$ is L.

3.2.3 Naive Subtyping Rules

In this section, we present some natural inference rules for subtyping. In the next sections we are going to demonstrate they are actually unsound using well-chosen examples and we will introduce our solutions for making them sound.

A subtyping judgment has the form $\Gamma \vdash T <: U$ where Γ is a typing context that we do not need to explicit here. The first subtyping rule, (S-UP), states that a virtual type $p.L$ is a subtype of its declared upper-bound, provided its prefix p has for type the class C that contains the declaration of L.

$$\text{(S-UP)}\ \frac{\Gamma \vdash p : C \qquad (\texttt{type}\ L <: T) \in C}{\Gamma \vdash p.L <: T[p]}$$

Of course, we must be careful here to reinterpret the upper-bound T in the context of the prefix p. This is expressed by the type expression $T[p]$ which represents the substitution of p for the self reference `this` in type T. As we do not aim at being completely formal here, we omit the symmetric rule, (S-DOWN), that links a virtual type to its declared lower-bound.

Now, let us look at to the semantics of a type assignment. Conceptually, if a type assignment `type` $L = T$ is visible from an object p, the virtual type $p.L$ is an alias for the type $T[p]$. In other words, both types can be regarded as equivalent. It follows that $p.L$ can be considered as both a subtype and a supertype of the type it is an alias for. The first fact can be deduced from the subtyping rule (S-ALIAS-RIGHT) below, and there is a symmetric rule (S-ALIAS-LEFT), that we omitted, for deducing the other fact.

$$\text{(S-ALIAS-RIGHT)}\ \frac{\Gamma \vdash p : C \qquad (\texttt{type}\ L = T) \in C}{\Gamma \vdash T[p] <: p.L}$$

In this first approach, we also omit the rule (S-CLASS) (resp. the rule (S-VIRTUAL)) which states that a class type C (resp. a virtual type $p.L$) is a subtype of itself. Both rules make subtyping a reflexive relation. We conclude this presentation of subtyping rules with the transitivity rule (S-TRANS) that makes subtyping a transitive relation.

$$\text{(S-TRANS)}\ \frac{\Gamma \vdash T <: S \qquad \Gamma \vdash S <: U}{\Gamma \vdash T <: U}$$

3.2.4 Naive Rules are Unsound

Some programs, if they are accepted by the type system, imply that subtyping is not consistent with subclassing. In Section 3.2.1, we have seen that such inconsistency is synonymous with unsafety. Consequently, it is crucial for our type system to reject such programs. In this section, we show that with the naive subtyping rules, we are able to exhibit such a program. The simplest case of dangerous program is the one of Figure 3.2, where A and B are two unrelated classes.

For this program to be well-formed, it must hold that the type A assigned to T is a subtype of the bound B of T. Using the naive rules for subtyping we are able to demonstrate this fact: `new` $C().T$ is an alias for A, so A is a subtype of `new` $C().T$ (rule (S-ALIAS-RIGHT)); T is bounded by B, so `new` $C().T$ is a subtype of B (rule (S-UP)); by transitivity of subtyping, A is a subtype of B (rule (S-TRANS)).

Now we have a well-formed program and we can show that, in the context of this well-formed program, A is a subtype of B, actually using the same reasoning we have used to show the program is well-formed. By the observation of Section 3.2.1, it means we can build an expression that is well-typed in the context of this program but whose evaluation goes wrong.

```
class A { }
class B { }
class C {
  type T <: B
  type T = A
}
```

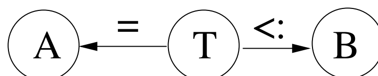


Figure 3.2: Oracle's example

Note that the reasoning we have done here amounts to have an oracle that says that we must pass by `new` $C().T$ in order to go from A to B. Of course this reasoning is nonsensical, it should be impossible to use a type assignment while trying to show it is well-formed. We could argue that the wrong move – the one that only an oracle could do – consists in taking the arrow from T to A in the wrong direction.

3.2.5 Backward Moves are Wanted for Type Aliases

In order to reject the program of the previous example, we can establish the following policy: for proving that a type T is a subtype of another type U, it must be possible to follow a path from (the symbol of) T to (the symbol of) U in the graph of symbols. We can see that it forbids to deduce that A is a subtype of B in the case of the previous program because there is no path from A to B that follows the edges; in particular the move from A to T is forbidden since it would require going backward along an alias arrow.

Such a policy is certainly sound, however it turns out to be far too restrictive, as illustrated by the program of Figure 3.3.

This program is perfectly safe and is actually accepted by the SCALA compiler. But the assignment of type U in class A will be rejected by our policy. For this assignment to be well-formed, it should be possible to show that in the context of the class A, A is a subtype of the upper-bound `this`.T of U. The strong law of never taking an arrow in the wrong direction prevents us from directly concluding that A is a subtype of `this`.T just by going through the symbol U, as in the previous example, and this is good. However, it also forbids us to deduce the result by following the path $A \stackrel{<:}{\longrightarrow} B \stackrel{=}{\longleftarrow} T$, which is bad. Clearly, in this case, we want to follow the edge corresponding to the assignment `type` $T = B$ in the direction $T \rightarrow B$. So, we have to be more subtle and refine our policy in order to making it more fine-grained.

3.2.6 Transitivity by Confluence

The two previous programs can be managed by defining a convergence policy that lets us deduce that a type is a subtype of another only when their symbols both converge towards a same symbol when following arrows in the graph. Such mechanism can be formalized by: (1) inlining transitivity in all subtyping rules except the transitivity rule, and (2) constraining the transitivity rule such that

```
class B {
  type T <: B
  type U <: this.T
}
class A extends B {
  type T = B
  type U = A
}
```

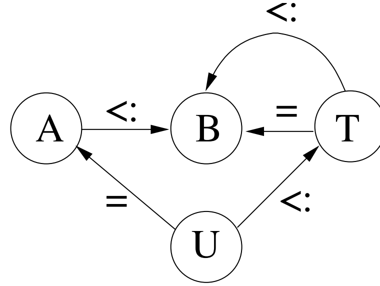


Figure 3.3: Example needing a backward alias move

the symbol of the intermediate type S is reachable from the symbols of the types T and U at both ends.

Inlining transitivity in all subtyping rules except the transitivity rule leads to the following:

$$\text{(S-UP)}\ \frac{\Gamma \vdash p : C \quad (\texttt{type}\ L <: T) \in C \quad \Gamma \vdash T[p] <: S}{\Gamma \vdash p.L <: S} \qquad \text{(S-ALIAS-RIGHT)}\ \frac{\Gamma \vdash p : C \quad (\texttt{type}\ L = T) \in C \quad \Gamma \vdash S <: T[p]}{\Gamma \vdash S <: p.L}$$

The new transitivity rule is written below. In this rule we write $U \prec T$ if there exists a path in the graph from the symbol of T to the symbol of U.

$$\text{(S-TRANS-S)}\ \frac{\Gamma \vdash T <: S \qquad \Gamma \vdash S <: U \qquad S \prec T \qquad S \prec U}{\Gamma \vdash T <: U}$$

The combination of these two modifications results in a new set of subtyping rules, which we call *structured subtyping* because the modifications of the rules impose a structure to the shape of subtyping derivations. Let us see now how these new definitions are able to reject the first program and to accept the second one. In program of Figure 3.2, we can see that the new policy forbids to deduce that A is a subtype of B because: (1) according to the direction of arrows we have $A \prec T$ and $B \prec T$, and (2) for applying the modified transitivity rule we need the opposite relations, namely that $T \prec A$ and $T \prec B$. In Program of Figure 3.3, we can check that we now accept the subtyping judgment $\texttt{this} : C \vdash A <: \texttt{this}.T$: we have A subtype of B (rule (S-EXTENDS)), and B subtype of $\texttt{this}.T$ (rule (S-ALIAS-RIGHT)). To conclude by applying rule (S-TRANS-S) we need to have $B \prec A$ and $B \prec T$, which is the case according to the direction of arrows.

3.2.7 Cycles

Our convergence policy works with the two previous examples, however it fails to reject the program of Figure 3.4. As we will see, accepting this program causes a breach in type safety since it becomes then possible to declare a coercion function from the class type A to the the class type B whereas classes A and B are totally unrelated.

```
class A {}
class B {}
class C {
  type U <: B
  type T <: this.U
  type U = this.T
  type T = A
}
```

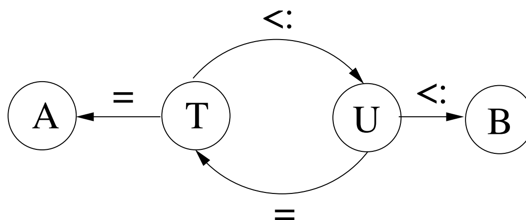


Figure 3.4: Cyclic example

Here is the detail of the proof that the program is accepted by the convergence policy. We start by showing that in this example all declarations are well-formed. Class declarations and type declarations are trivially well-formed. For showing that type assignments are well-formed too we need to prove that the type of the right-hand side of each type assignment is a subtype of the declared bound of the assigned type symbol.

For the assignment $\texttt{type}\ U = \texttt{this}.T$, we have to show that, in the context of class C, the type $\texttt{this}.T$ is a subtype of B. This can be done by two successive applications of the rule (S-UP) followed by the reflexivity rule for class types (S-CLASS):

$$\text{(S-UP)}\ \frac{\text{(S-UP)}\ \dfrac{\text{(S-CLASS)}\ \dfrac{}{\texttt{this} : C \vdash B <: B} \quad (\texttt{type}\ U <: B) \in C \quad \texttt{this} : C \vdash \texttt{this} : C}{\texttt{this} : C \vdash \texttt{this}.U <: B} \quad (\texttt{type}\ T <: \texttt{this}.U) \in C \quad \texttt{this} : C \vdash \texttt{this} : C}{\texttt{this} : C \vdash \texttt{this}.T <: B}$$

For the assignment $\texttt{type}\ T = A$, we have to show that, in the context of class C, the type A is a subtype of $\texttt{this}.U$. This can be done by two successive applications of (S-ALIAS-RIGHT) followed by the reflexivity rule for class types (S-CLASS):

$$\text{(S-Class)}\ \frac{}{\texttt{this} : C \vdash\ A <: A}$$

$$\text{(S-Alias-Right)}\ \frac{(\texttt{type}\ T = A) \in C \qquad \texttt{this} : C \vdash\ \texttt{this} : C}{\texttt{this} : C \vdash\ A <: \texttt{this}.T}$$

$$\text{(S-Alias-Right)}\ \frac{(\texttt{type}\ U = \texttt{this}.T) \in C \qquad \texttt{this} : C \vdash\ \texttt{this} : C}{\texttt{this} : C \vdash\ A <: \texttt{this}.U}$$

Note that in both cases we do not even need the full transitivity rule (S-Trans-S) for proving the well-formedness of the type assignment. Now that we have shown that the program is well-formed, we can easily prove that A is a subtype of $\texttt{new}\ C().T$ and that $\texttt{new}\ C().T$ is a subtype of B. By the second observation of Section 3.2.1 this implies that it is possible to build a coercion function from A to B, which is completely unsafe. The interesting point in this example is that it is unsafe even though it is not possible to deduce that A is a subtype of B. In order to prove such a subtyping judgment we would have to apply the transitivity rule with the two premises $\vdash\ A <: \texttt{new}\ C().T$ and $\vdash\ \texttt{new}\ C().T <: B$. But the confluence check associated with this rule, namely that $T \prec A$ and $T \prec B$, does not succeed. Somehow, we managed to bypass the confluence check by introducing a loop between symbols T and U in the graph of symbols.

Interestingly, this example is similar to the one used for showing that the local confluence of a notion of reduction does not imply its global confluence. If we unfold the graph of symbols, we get an infinite graph that closes at infinity, i.e. never (See Figure 3.5).

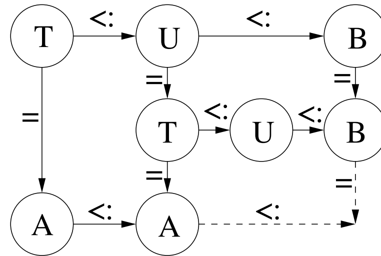


Figure 3.5: Unfolded cyclic example

In this example, the source of the problem is the cycle between symbols T and U in the graph of symbols. The last refinement of the policy that we propose is then to forbid cycles in the graph of symbols.

3.2.8 Well-founded Relation

The anti-cycle policy can be formalized by saying that the relation between symbols in the graph is *well-founded*, in other words there must not exist an infinite path in this graph. Since the graph is derived from the declarations of the program, we end up with the following simple constraint for preventing cycles: there must exist a well-founded relation $\prec$ over class and type symbols that is consistent with class and type declarations.

We can check that such a relation makes the type system reject the program of Figure 3.4. Suppose by contradiction that the program is well-formed. Since $\prec$ is consistent with type declarations, `type` T <: `this`.U implies $U \prec T$, and `type` U = `this`.T implies $T \prec U$. With $U \prec T$ and $T \prec U$ we can construct the infinite descending chain of symbols $\cdots \prec U \prec T \prec U$. This implies that $\prec$ is not well-founded, which contradicts the well-formedness of the program.

3.2.9 Incompatible bounds

In Figure 3.6, we present the last example of this section. It broaches the problem of incompatible bounds in the declaration of a virtual type.

```
class A {}
class B {}
class C {
  type T >: A <: B
}
```

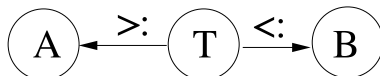


Figure 3.6: Incompatible bounds

This program looks dangerous because for an instance v of class C, it lets us deduce that A is a subtype of $v.T$ (by rule (S-DOWN)) and that $v.T$ is a subtype of B (by rule (S-UP)), whereas A and B are two unrelated classes. According to the second observation of Section 3.2.1, this looks like a breach in type safety. What saves us here is that class C and all its potential subclasses are impossible to instantiate since it is impossible to assign a type value to T. Our reasoning for showing that the program is unsafe was correct except that we have assumed the existence of an instance v of C that actually cannot exist. This is explained in more detail in Section 3.8.

3.2.10 Conclusion

In this section, we have explained why it is necessary for type safety that subtyping implies subclassing. To help us in the discussion, we have introduced a view of a program as a graph of symbols. After presenting naive rules for subtyping, we have discovered that such rules consider as well-formed some particular unsafe program. We have suggested one possibility for rejecting similar programs: avoiding following an arrow along the wrong direction in the graph of symbols. However, this policy turns out to be too strong since it rejects also some correct programs. So, we had to be more subtle. We have discovered that the two considered programs can be managed by defining a policy that lets us deduce that a type is a subtype of another if their symbols both converge to a same symbol. Formally, it amounts to inline transitivity in all the rules except the transitivity rule and to add constraints on the applicability of the transitivity rule. Unfortunately, we have observed that the safety of such technique is broken if there are cycles in the graph of symbols, as demonstrated by another example wrongly accepted by the type system. To prevent cycles in the graph,

we have imposed the condition that the relation derived from the graph should be well-founded. Summarizing, the result of all this analysis is the definition of two theoretical tools, two techniques that together ensure a sound subtyping relation: a restricted subtyping relation, called structured subtyping, and a well-founded relation on symbols. This section has just described the philosophy of these tools, their precise definitions and the properties they satisfy are the main subject of the rest of this chapter.

3.3 Syntax

Our proof of type safety for virtual types is based on a simple class-based object-oriented calculus. We adopt the traditional approach of defining programs of the calculus in two steps: first we define the set of *pre-programs*, which roughly corresponds in a compiler to programs that are syntactically correct and where name analysis has been resolved. Then we define the set of *well-formed programs* as the subset of pre-programs that obey some typing discipline expressed as inference rules. For pre-programs, we give two views: in the first one, that we call the *programmer's view*, pre-programs are defined entirely in terms of abstract syntax. In the second one, that we call by contrast the *mathematician's view*, pre-programs are defined as a set of access functions that return the attributes attached to each name. Both views are complementary: the programmer's view is very close to the concrete syntax of programs and is useful for writing examples, whereas the mathematician's view is convenient for formal theoretical developments about the calculus.

3.3.1 Programmer's View

The abstract syntax of the calculus is defined in Figure 3.7. There are six kinds of symbols corresponding to the different kinds of names that can occur in a program: classes, virtual types, fields, methods, method parameters and local variables.

A program $\overline{D}\ t$ consists of a set $\overline{D}$ of class definitions and a main term t to be evaluated in the context of these classes. A class declaration has the form `class` C `extends` C_{opt} $\{\ \overline{d}\ \}$; it defines a new class C with an optional superclass C_{opt} and member declarations $\overline{d}$. C_{opt} is either another class symbol C' or `none`, a construct that is used to express the absence of superclass; it means that our calculus supports only single inheritance. The special class `Object` which is the root of the FJ class hierarchy can be emulated here by defining a class `Object` with no super-class and such that the following conditions are satisfied: all other classes have a super-class, and all top-level classes inherits from `Object`.

There are two kinds of declarations inside a class: those that *define* a new symbol and those that *assign a value* to an existing symbol. For emphasizing this difference of role, we call the former *declarations* and the later *valuations*. A type declaration `type` $L >: T_{opt} <: U$ defines a new virtual type L with a type upper-bound U and an optional lower-bound T_{opt}. We could also have made the upper-bound optional, but a virtual type without upper-bound is similar to a virtual type with `Object` as upper-bound. A type valuation `type` $L = T$ in a class C makes L an alias for T in all instances of C. A method declaration `def` $m(\overline{x} : \overline{T}) : T$ defines a new method m with formal parameters $\overline{x}$ bounded

by $\overline{T}$ and result type T. At the point of its declaration the method is still abstract because it misses a body; the only way of making it concrete is to write a method valuation. A method valuation `def` $m = t$ in a class C states that m has body t in all instances of C. A method valuation inherits the parameters of the method's declaration, so that they can be referred to inside t. Fields are declared in classes through the syntax `val` $f : T$; following FJ tradition, the only way of assigning a value to a field is through an instance creation expression.

The concepts that are described by the terms of our calculus are standard for an object-oriented calculus, but some points require an explanation. Compared to FJ there are design choices that are imposed by virtual types. We already explained in the introduction the need for local definitions. A local definition `let`: $T = t$ `in` u introduces a local variable with type T and value t whose scope is limited to the term u. FJ uses the concept of variable to represent both the current instance of a class and a method parameter. In our calculus a variable x always represent a method parameter, and we let the current instance of a class be represented by the special symbol `this`. Local variables are represented even differently by integers using the de Bruijn notation. There are also constructs for selecting a field or calling a method. Fields are selected on a subset of terms, that we call paths, with the construct $p.f$. The reason why we restrict the form of terms that can be used as prefixes of a field selection is explained in the description of paths. The construct $p.m(\overline{x} = \overline{t})$ represents the call of the method m on the receiver object p with arguments $\overline{t}$. Usually the link between an argument and a formal parameter is determined by their respective position. Here the link is made explicit by associating each argument with a parameter. Finally, instances of a class C are created using the construct `new` $C(\overline{f} = \overline{t})$. The arguments $\overline{t}$ of this expression specify the values of the fields $\overline{f}$ they are associated with.

Types are either concrete or abstract. A concrete type C refers to a class symbol C and is called a *class type*. An abstract type $p.L$ refers to a virtual type symbol L and is called a *member type* or simply a virtual type; it is selected on a path p. Such types are called virtual because their actual value depends on the runtime value of the object p on which they are selected; generally, at compile time, we just know the bound of such a type.

As in FJ, the values of our calculus are the subset of terms that have the form `new` $C(\overline{f} = \overline{v})$, i.e. they are represented as instance creation expressions with evaluated fields.

Paths are, by definition, the subset of terms that are allowed in types. For type safety reasons, it is important that such terms do not loop nor evaluate to different values in successive evaluations. It can be shown that our definition of paths satisfies the former property. The latter property is actually satisfied by all terms in our calculus because the calculus can be shown confluent and there is no means of modifying the state of a program by a side-effect. The reason fields can only be selected on paths and not on arbitrary terms is that the type attributed to a field selection depends on its prefix and that types can only depend on paths; our restriction on the terms that can appear in types implies a restriction on the terms that can be used as prefixes of a field selection. The same argumentation justifies that receiver objects of method calls have to be paths. Actually, for the same reasons, *arguments* of method calls would also have to be paths if we had chosen to make parameter types and result types depend on value parameters.

Simple Paths. As a simplification, we start by considering a subset of the calculus where paths cannot be field selections $p.f$. More formally we replace the definition of paths with the following syntax.

$$p ::= \texttt{this} \mid x \mid n \mid v$$

Such a constraint simplifies the theoretical study of the calculus because it implies that all terms occurring in types are irreducible. The reason we keep full paths in the syntax nevertheless is that we conjecture there exists a natural extension of our formalism that can deal with them [2].

Syntactic Sugar. Superclasses and type lower-bounds are optional. When a class has no superclass, we just write `class A { ... }` instead of the complete form `class A extends none { ... }`, and when a virtual type has no lower-bound we write just `type L <: A` instead of `type L >: none <: A`. Sometimes we also write `type T` instead of `type T >: none <: Object`; in this case we assume the definition of a class `Object`, as explained above.

Sequences. Sequences are ubiquitous in our formalism. We write ϵ for an empty sequence of elements. We write $|s|$ for the length of a sequence s. We write also $\overline{x}$ for a finite sequence of elements ranged over by the meta-variable x. In that case, if $0 \leq i < |\overline{x}|$, x_i represents the $(i+1)$-th element of the sequence $\overline{x}$ (i.e. we start counting from index 0).

Definition 3.1 (Occurrence of self and free parameters)
We define some notations to speak of the occurrence of self or some parameters in a path. These definitions extend naturally to types.

1. *We write* $\texttt{this} \in p$ *if* `this` *occurs in path* p.
2. *We write* $x \in fp(p)$ *if parameter* x *occurs in path* p.

3.3.2 Mathematician's View

In the mathematician's view we regard a pre-program as a set of functions that take symbols as arguments and return the attributes attached to such symbols. For instance there is a function fieldOwner that returns the class enclosing the declaration of a particular field symbol, and there is a function fieldType for accessing the declared type of that field. With this presentation the important assumption that to each symbol corresponds a *unique* declaration comes for free, it is contained in the fact that mathematical functions always return the same value when applied to the same argument, by definition. With the definition of pre-programs as abstract syntax trees we would have to define and manipulate extra assumptions about uniqueness of declarations. The access functions also enforce the *existence* of a declaration for every symbol that is used in the program because their domain is by definition the entire set of symbols. This last property places us conceptually after name analysis. In the mathematician's view, a program Π is a tuple:

[2] Such an extension is discussed in Section 3.10.1.

Syntax

Class symbol	A, B, C			
Type symbol	L			
Field symbol	f			
Method symbol	m			
Parameter symbol	x			
Integer	$n, i, j,$ k, N			
Class declaration	D	$::=$	$\texttt{class}\ C\ \texttt{extends}\ C_{opt}\ \{\ \overline{d}\ \}$	
Super class	C_{opt}	$::=$	$C \mid \texttt{none}$	
Member declaration	d	$::=$	$\texttt{type}\ L >: T_{opt} <: U$	type declaration
		$\mid$	$\texttt{type}\ L = T$	type valuation
		$\mid$	$\texttt{val}\ f : T$	field declaration
		$\mid$	$\texttt{def}\ m(\overline{x} : \overline{T}) : T$	method declaration
		$\mid$	$\texttt{def}\ m = t$	method valuation
Term	t, u	$::=$	$\texttt{this}$	current instance
		$\mid$	x	method parameter
		$\mid$	n	local variable
		$\mid$	$p.f$	field selection
		$\mid$	$\texttt{new}\ C(\overline{f} = \overline{t})$	instance creation
		$\mid$	$p.m(\overline{x} = \overline{t})$	method call
		$\mid$	$\texttt{let}: T = t\ \texttt{in}\ u$	local definition
Type	$S, T, U,$ V, W	$::=$	C	class type
		$\mid$	$p.L$	member type
Optional Type	T_{opt}	$::=$	$T \mid \texttt{none}$	
Path	p, q	$::=$	$\texttt{this} \mid x \mid n \mid v \mid p.f$	
Value	v, w	$::=$	$\texttt{new}\ C(\overline{f} = \overline{v})$	
Program	P	$::=$	$\overline{D}\ t$	

Figure 3.7: Calculus Syntax

$$
\begin{array}{rcll}
\Pi & = & \langle & E_C, E_L, E_f, E_m, E_x, \\
 & & & \prec \in (E_C \cup E_L \cup E_f)^2, \\
 & & & \text{classSuper} \quad \in \quad E_C \to E_C \text{ (partial) }, \\
 & & & \text{typeOwner} \quad \in \quad E_L \to E_C, \\
 & & & \text{typeUpperBound} \quad \in \quad E_L \to E_T, \\
 & & & \text{typeLowerBound} \quad \in \quad E_L \to E_T \text{ (partial) }, \\
 & & & \text{typeValue} \quad \in \quad E_C \times E_L \to E_T \text{ (partial) }, \\
 & & & \text{fieldOwner} \quad \in \quad E_f \to E_C, \\
 & & & \text{fieldType} \quad \in \quad E_f \to E_T, \\
 & & & \text{methodOwner} \quad \in \quad E_m \to E_C, \\
 & & & \text{methodType} \quad \in \quad E_m \to E_T, \\
 & & & \text{methodValue} \quad \in \quad E_C \times E_m \to E_t \text{ (partial) }, \\
 & & & \text{paramOwner} \quad \in \quad E_x \to E_m, \\
 & & & \text{paramType} \quad \in \quad E_x \to E_T \\
 & & & \text{main} \quad \in \quad E_t \\
 & & \rangle &
\end{array}
$$

This tuple consists of:

- A set E_C of class symbols, a set E_L of virtual type symbols, a set E_f of field symbols, a set E_m of method symbols and a set E_x of parameter symbols.
- A binary relation $\prec$ on class, type and field symbols. This relation is required to be well-founded in well-formed programs and is used to constrain definitions so as to ensure type safety (See Section 3.2.7).
- A partial function (classSuper) which maps a class C to its superclass C'. Note that this function is partial because in our calculus a class can have no superclass.
- A set of accessors for virtual types: a function (typeOwner) which maps a virtual type L to its enclosing class C, i.e. the class where L is initially declared; two functions (typeUpperBound) and (typeLowerBound) which maps a virtual type L to its declared bounds T and U; a partial function (typeValue) which maps a class C and a virtual type L to the value T given to L in class C.
- A set of accessors for fields: a function (fieldOwner) which maps a field f to its enclosing class C; a function (fieldType) which maps a field f to its declared type T.
- A set of accessors for methods: a function (methodOwner) which maps a method m to its enclosing class C; a function (methodType) which maps a method m to its return type T [3]; a partial function (methodValue) which maps a class C and a method m to the implementation t of m in class C.
- A set of accessors for parameters: a function (paramOwner) which maps a parameter x to its method m; a function (paramType) which maps a parameter x to its declared type T.

[3] Note that parameter types are attached to parameter symbols.

- A term (main) which represents the entry point of the program.

We still rely on the definitions as abstract syntax trees given in the programmer's view for defining the set E_t of terms and the set E_T of types.

Note that what we call the mathematician's view roughly corresponds to the data structures used in a compiler for representing a program after name analysis. For instance, a compiler does not traverse the whole syntax tree each time it needs the type of a symbol, rather it uses access functions (or methods) that directly return this information. We find then strange that this representation is almost never used for describing type systems in the litterature. We think that formal studies of type systems could also benefit from this representation.

3.3.3 Example

We show on an example how the programmer's view of a pre-program can be entirely recovered from its mathematician's view. The following program, written as an abstract syntax tree, has a representation with access functions which is contained in Figure 3.8. The general translation is formalized in Figure 3.9.

```
class A {
  type T <: A
  def foo(x: this.T): this.T
}
class B extends A {
  type T = B
  def foo = this.foo(x.foo(this))
}

let: B = new B() in 0.foo(0)
```

3.3.4 De Bruijn's Notation

Local variables introduce binders. Because it makes the rules more readable, the tradition in the description of calculi is to have an informal treatment of binders, which is based on variables and with the implicit assumption that terms must be considered modulo alpha-renaming. In this chapter we want a formal treatment of binders because we want to be sure of our proof. Among the many ways of representing binders mathematically, we choose the de Bruijn notation which consists in representing variables by integers. Method parameters also are often considered as binders. There are several reasons why we use symbols instead of the de Bruijn notation for parameters: the first one is that we want to be able to define the simultaneous substitution of actual parameters for formal parameters in an extension of this calculus with parameter dependent method types; this simultaneous substitution is easier to define with symbols than with binders. The second reason comes from the well-known isomorphism class/method and field/parameter which suggests that fields and parameters have a lot in common and should consequently share similar representations.

Definition 3.2 (Free variables)

$$
\begin{array}{lcl}
E_C & = & \{\, A, B \,\} \\
E_L & = & \{\, T \,\} \\
E_f & = & \emptyset \\
E_m & = & \{\, foo \,\} \\
E_x & = & \{\, x \,\} \\
\prec & = & \{\, (A,B), (B,T), (A,T) \,\} \\
\text{classSuper} & = & \{\, B \mapsto A \,\} \\
\text{typeOwner} & = & \{\, T \mapsto A \,\} \\
\text{typeUpperBound} & = & \{\, T \mapsto A \,\} \\
\text{typeLowerBound} & = & \emptyset \\
\text{typeValue} & = & \{\, (B,T) \mapsto B \,\} \\
\text{fieldOwner} & = & \emptyset \\
\text{fieldType} & = & \emptyset \\
\text{methodOwner} & = & \{\, foo \mapsto A \,\} \\
\text{methodType} & = & \{\, foo \mapsto \texttt{this}.T \,\} \\
\text{methodValue} & = & \{\, (B, foo) \mapsto \texttt{this}.foo(x.foo(\texttt{this})) \,\} \\
\text{paramOwner} & = & \{\, x \mapsto foo \,\} \\
\text{paramType} & = & \{\, x \mapsto \texttt{this}.T \,\} \\
\text{main} & = & \texttt{let}\!: B = \texttt{new}\ B(\ )\texttt{in}\ 0.foo(0)
\end{array}
$$

Figure 3.8: Mathematician's view of a program

$$
\begin{array}{lcl}
\text{parameters}_\Pi(m) & = & \{x : T \mid \\
 & & \quad \Pi.\text{paramOwner}(x) = m, \\
 & & \quad \Pi.\text{paramType}(x) = T\} \\
\text{members}_\Pi(C) & = & \{\texttt{val}\ f : T \mid \\
 & & \quad \Pi.\text{fieldOwner}(f) = C, \\
 & & \quad \Pi.\text{fieldType}(f) = T\} \\
 & \cup & \{\texttt{def}\ m(\overline{x} : \overline{T}) : T \mid \\
 & & \quad \Pi.\text{methodOwner}(m) = C, \\
 & & \quad \text{parameters}_\Pi(m) = \{\, \overline{x} : \overline{T} \,\}, \\
 & & \quad \Pi.\text{methodType}(m) = T\} \\
 & \cup & \{\texttt{def}\ m = t \mid \Pi.\text{methodValue}((C, m)) = t\} \\
 & \cup & \{\texttt{type}\ L >: T_{opt} <: U \mid \\
 & & \quad \Pi.\text{typeOwner}(L) = C, \\
 & & \quad \Pi.\text{typeUpperBound}(L) = U, \\
 & & \quad T_{opt} = \begin{cases} T \text{ if } \Pi.\text{typeLowerBound}(L) = T \\ \texttt{none} \text{ otherwise} \end{cases} \quad \} \\
 & \cup & \{\texttt{type}\ L = T \mid \Pi.\text{typeValue}((C, L)) = T\} \\
\text{classes}_\Pi & = & \{\texttt{class}\ C\ \texttt{extends}\ C_{opt}\ \{\ \overline{d}\ \} \mid \\
 & & \quad C_{opt} = \begin{cases} C' \text{ if } \Pi.\text{classSuper}(C) = C' \\ \texttt{none} \text{ otherwise} \end{cases} \quad , \\
 & & \quad \text{members}_\Pi(C) = \{\, \overline{d} \,\}\} \\
\text{program}_\Pi & = & \overline{D}\ t \text{ s.t. } \text{classes}_\Pi = \{\, \overline{D} \,\},\ \Pi.\text{main} = t
\end{array}
$$

Figure 3.9: Recovering the Programmer's View

The free variables of a term t, noted $fv(t)$, is defined using an auxiliary function $fv_k(t)$ that returns the free variables of t at depth k.

$$
\begin{array}{rcl}
fv_k(n) & = & \begin{cases} \emptyset & \text{if} \quad n < k \\ \{\, n-k \,\} & \text{if} \quad n \geq k \end{cases} \\
fv_k(x) & = & \emptyset \\
fv_k(\texttt{let}{:}T = t \ \texttt{in}\ u) & = & fv_k(T) \cup fv_k(t) \cup fv_{k+1}(u) \\
fv_k(\cdots) & = & \cdots \\
 & & \\
fv(t) & = & fv_0(t)
\end{array}
$$

Definition 3.3 (Closed terms and closed types)

We say that a term t is closed if: $\texttt{this} \notin t$ and $fp(t) \equiv \emptyset$ and $fv(t) \equiv \emptyset$. A similar definition applies to types.

Definition 3.4 (Lifting)

We define the lifting $t{\uparrow}^n_k$ of the term t at depth $k \geq 0$ with amplitude $n \geq 0$. We also define $t{\uparrow}^n$ as a shortcut for $t{\uparrow}^n_0$. These definitions naturally extend to types.

$$
\begin{array}{rcl}
i{\uparrow}^n_k & = & \begin{cases} i & \text{if} \quad i < k \\ i+n & \text{if} \quad i \geq k \end{cases} \\
x{\uparrow}^n_k & = & x \\
(\texttt{let}{:}T = t \ \texttt{in}\ u){\uparrow}^n_k & = & \texttt{let}{:}T{\uparrow}^n_k = t{\uparrow}^n_k \ \texttt{in}\ u{\uparrow}^n_{k+1} \\
\cdots{\uparrow}^n_k & = & \cdots \\
 & & \\
t{\uparrow}^n & = & t{\uparrow}^n_0
\end{array}
$$

Definition 3.5 (Substitution)

We write $t[k := u]$ ($k \geq 0$) the substitution in t of u for the variable k. This definition extends to types naturally.

$$
\begin{array}{rcl}
n[k := t] & = & \begin{cases} n & \text{if} \quad n < k \\ t{\uparrow}^n & \text{if} \quad n = k \\ n-1 & \text{if} \quad n > k \end{cases} \\
x[k := t] & = & x \\
(\texttt{let}{:}T = t_1 \ \texttt{in}\ t_2)[k := t] & = & \texttt{let}{:}T[k := t] = t_1[k := t] \ \texttt{in}\ t_2[k+1 := t] \\
\cdots[k := t] & = & \cdots
\end{array}
$$

Note that in addition to substituting a term for a variable, this function also decreases all free variables by one.

Definition 3.6 (Dropping)

We write $t{\downarrow}_k$ the action of decrementing all free variables by one at depth $k \geq 0$. We also define $t{\downarrow}$ as a shortcut for $t{\downarrow}_0$ [4]. These definitions naturally extend to types.

---

[4] Note that we could alternatively have defined $t{\downarrow}$ as $t[0 := 0]$.

$$
\begin{array}{rcl}
i{\downarrow_k} & = & \left\{ \begin{array}{lll} i & \mathit{if} & i < k \\ \mathit{pred}(i) & \mathit{if} & i \geq k \end{array} \right. \\
x{\downarrow_k} & = & x \\
(\mathbf{let}{:}T = t \ \mathbf{in}\ u){\downarrow_k} & = & \mathbf{let}{:}T{\downarrow_k} = t{\downarrow_k}\ \mathbf{in}\ u{\downarrow_{k+1}} \\
\cdots{\downarrow_k} & = & \cdots \\
 & & \\
t{\downarrow} & = & t{\downarrow_0}
\end{array}
$$

where $\mathit{pred}(i) = i - 1$ *if* $i > 0$ *and* 0 *otherwise.*

3.4 Semantics

The computational meaning of a program is defined by a call-by-value small-step operational semantics. More precisely we define the reduction of a term in the context of a program Π. The evaluation of a program then consists in reducing the main term of the program in the context of the class declarations until reaching a value. The reduction relation needs a subclassing relation between class symbols. Both relations are summarized in Figure 3.10 and Figure 3.11. They are implicitly parameterized by a program Π [5]. We also use some notations to alleviate the rules. For instance we write $(\texttt{def}\ m = t) \in C$ as an abbreviation for $(\texttt{def}\ m = t) \in \text{members}_\Pi(C)$, which in turn is equivalent to $\Pi.\text{methodValue}((C, m)) = t$. Similarly we write $\texttt{class}\ C\ \texttt{extends}\ C'\ \{\ \overline{d}\ \}$ as a shortcut for $\texttt{class}\ C\ \texttt{extends}\ C'\ \{\ \overline{d}\ \} \in \text{classes}_\Pi$.

The evaluation rules are standard and do not deserve much explanation. In rule (R-Call), we substitute in the body t of the method m the receiver object v for the current instance `this` and the actual arguments $\overline{v}$ for the formal parameters $\overline{x}$. The substitution in a term t of a path p for the current instance `this` is written $t[p]$, and the substitution in a term t of paths $\overline{p}$ for parameters $\overline{x}$ is written $t[\overline{x}\backslash\overline{p}]$. As we always substitute values, i.e. closed terms, the substitutions for self and parameters can be defined by straightforward replacement without taking into account possible name captures.

Rules (R-Prefix) and (R-Receive) respectively allow the reduction of the prefix in a field selection and of the receiver object in a method call. Actually these rules are never applicable with our syntactical restriction that paths cannot be field selections (See Section 3.3.1), because this restriction ensures that paths are always in normal form: a path is either an abstract value (self, parameter and variable) or a concrete value. We keep nevertheless rules (R-Prefix) and (R-Receive) in the semantics in order to be general enough for accepting an extension with reducible paths.

3.5 Typing

The rules that decide if a program is well-formed are based on the concept of type. A type is an abstract interpretation of a term: it is impossible to statically know the exact result of evaluating a term but we can always approximate its value by saying it belongs to a certain set, types are a means of describing these

[5] To be completely rigorous we should have written $\Pi \vdash t \to u$ and $\Pi \vdash C \triangleleft C'$ instead of $t \to u$ and $C \triangleleft C'$.

$$\text{(R-PREFIX)}\ \frac{p \rightarrow q}{p.f \rightarrow q.f} \qquad \text{(R-SELECT)}\ \frac{}{\texttt{new}\ C(\overline{f} = \overline{v}).f_i \rightarrow v_i}$$

$$\text{(R-FIELD)}\ \frac{t \rightarrow u}{\texttt{new}\ C(\overline{f} = \overline{v}, t, \overline{t}) \rightarrow \texttt{new}\ C(\overline{f} = \overline{v}, u, \overline{t})}$$

$$\text{(R-RECEIVE)}\ \frac{p \rightarrow q}{p.m(\overline{x} = \overline{t}) \rightarrow q.m(\overline{x} = \overline{t})}$$

$$\text{(R-ARG)}\ \frac{t \rightarrow u}{v.m(\overline{x} = \overline{v}, t, \overline{t}) \rightarrow v.m(\overline{x} = \overline{v}, u, \overline{t})}$$

$$\text{(R-CALL)}\ \frac{v \equiv \texttt{new}\ C(\overline{f} = \overline{w}) \qquad C \lhd C' \qquad (\texttt{def}\ m = t) \in C'}{v.m(\overline{x} = \overline{v}) \rightarrow t[v][\overline{x} \backslash \overline{v}]}$$

$$\text{(R-LOCAL)}\ \frac{t \rightarrow t'}{\texttt{let}\!:T = t\ \texttt{in}\ u \rightarrow \texttt{let}\!:T = t'\ \texttt{in}\ u}$$

$$\text{(R-LET)}\ \frac{}{\texttt{let}\!:T = v\ \texttt{in}\ t \rightarrow t[0 := v]}$$

Figure 3.10: Term reduction ($t \rightarrow u$)

$$\text{(SC-EXTENDS)}\ \frac{\texttt{class}\ C\ \texttt{extends}\ C'\ \{\ \overline{d}\ \} \qquad C' \lhd C''}{C \lhd C''}$$

$$\text{(SC-CLASS)}\ \frac{}{C \lhd C}$$

Figure 3.11: Subclassing ($C \lhd C'$)

sets of values. There are two judgments that are related to types in our calculus: a typing judgment $\Gamma \vdash t : T$ means that t can be approximated by the type T, and a subtyping judgment $\Gamma \vdash T <: U$ means that type T represents a smaller set of values than type U.

In our calculus the typing relation and the subtyping relation have mutually recursive definitions. In an object-oriented calculus, it is natural that typing depends on subtyping: for instance in order to check that a selection $p.f$ is well-typed, we check that the type of p is a *subtype* of a class type C such that class C declares the field f. However, it is less common that subtyping depends on typing, for instance it is not the case in FJ and in FGJ. In our calculus, it comes from the fact that types syntactically depend on terms (more precisely on paths). For instance, in order to check that a virtual type $p.L$ is a subtype of another type U, we usually need to find the more precise declaration of L visible from p. We perform this operation by *assigning a type* T to p and finding a super-type of T that contains a declaration for L. This cyclicity among the definitions is what makes the proof of soundness for virtual types so difficult.

The typing relations (typing and subtyping) are summarized in Figures 3.13 and 3.14. They are parameterized by a typing context Γ. A typing context (See Definition 3.7) has the form $\Gamma_o;\ \overline{x} : \overline{T};\ \overline{U}$ where Γ_o denotes either the current enclosing class C or the top-level context `root`, $\overline{x}$ denotes the parameters of the current method with their declared types $\overline{T}$, and $\overline{U}$ represents the types of the visible local variables.

Definition 3.7 (Typing context)

Owner context	Γ_o	::=	$C \mid$ `root`
Method context	Γ_m	::=	$\Gamma_o;\ \overline{x} : \overline{T}$
Full context	Γ	::=	$\Gamma_m;\ \overline{U}$

Most of the rules are straightforward. We concentrate our explanations on the few points that we think could confuse the reader. First of all it is worth noting that there is no typing rule for subsumption, that is, there is no rule that allows to deduce that a term t has type U under the assumptions that t has type T and that T is a subtype of U. One common way of doing without subsumption rule is to inline it when needed, another way is to make it an admissible rule and that is what we do in our calculus (See Lemma A.1). By definition, a rule is admissible in a system of inference rules if its conclusion holds whenever its premises hold. The two ingredients that make subsumption admissible is our general transitivity rule for subtyping (S-TRANS) and the fact that each typing rule contains a subtyping premise for opening the type that will be assigned. For instance, by one application of the rule (T-THIS) we can directly deduce that `this` is an instance of all super-classes of the class where it is used.

The rule for typing variables is standard when using the de Bruijn notation: in a context $\overline{U}$ for local variables, the type of the variable n is the $(n + 1)$-th element of the sequence $\overline{U}$ when starting from the right (for instance the type of 0 is the last element of the sequence). This type has to be lifted by an amplitude that is equal to the number of types that follow it in the typing environment in order to be still able to refer to variables that are defined before it in the typing environment.

The rule (T-CALL) for method calls contains the essence of virtual types. What is classical is that the arguments of the methods must conform to the declared types of the parameters. What is less standard is that the declared types of the parameters $\overline{T}$ must be reinterpreted in the context of the receiver object p. This reinterpretation is expressed by substituting p for `this` in the types $\overline{T}$, what we note $\overline{T}[p]$. Similarly, the type given to the method call corresponds to a reinterpretation of the declared result type of the method.

When creating an instance of a class C in rule (T-NEW) we must check that the values given to the fields of the instance conform to the declared bounds of these fields. Once again we must reinterpret these bounds in the context of the instance we are creating. Actually for checking this conformance we abstract over the particular instance we are creating and place ourselves in a typing context $\texttt{this} : C$. However we must be careful here because there are two typing contexts in play: the context of the instance creation expression that is also the one of the field values $\overline{t}$ and the context that we created for the conformance check. We perform the transition between both contexts by imposing that the types of field values that will be considered for the conformance checks are closed. Note that a closed type is not necessarily a class type, it can also be a virtual type with a closed prefix, like $\texttt{new}\ A(f = \texttt{new}\ B()).L$. The rule (T-NEW) also resorts to an auxiliary predicate for checking the completeness of the instance that is being created. This predicate is defined in Figure 3.12. Intuitively, the predicate $\text{isComplete}(C, \overline{f})$ is true if all instances of the class C that contain values for the fields $\overline{f}$ are complete. By complete, we mean that all fields that are visible from C are initialized in the instance (Condition 1), that all methods that are visible from C are implemented in a superclass of C (Condition 2), and that all virtual types that are visible from C are assigned a value in a superclass of C (Condition 3). By definition, a member declaration is visible from a class if it is contained in a superclass of this class.

Subtyping rules are very intuitive. Rules (S-CLASS) and (S-VIRTUAL) make the subtyping relation reflexive. One may just wonder why rule (S-VIRTUAL) needs premises. Amongst the premises of rule (S-VIRTUAL), we check that the prefix p of the virtual type $p.L$ is well-typed; we actually need this property for proving Lemmas A.3 and A.4, which state that the free parameters and variables appearing in a subtyping judgment are in the typing context. Rule (S-EXTENDS) imports the subclass hierarchy into the subtyping relation. Rules (S-UP) and (S-DOWN) allow to approximate a virtual type by its lower or upper bound. As for the return type of a function, these bounds are reinterpreted in the context of the virtual type's prefix by means of a substitution. Rules (S-ALIAS-LEFT) and (S-ALIAS-RIGHT) translate the idea that a virtual type is just an alias for the type it contains. They are very similar to rules (S-UP) and (S-DOWN); actually, as far as the subtyping relation is concerned a type valuation $\texttt{type}\ L = T$ is treated like a type declaration $\texttt{type}\ L >: T <: T$. Finally, we already mentioned the rule (S-TRANS) that makes the subtyping rule transitive. In Section 3.7 we present a restricted version of this rule, we call the resulting subtyping relation *structured subtyping* because it imposes a good structure on the shape of derivations. This structure is then used to prove crucial properties of the system [6].

[6] The attentive reader will have noticed that structured subtyping is actually already used in rule (T-NEW).

$$\frac{\begin{array}{c}(1)\ \forall\, C', f, T.\quad C \triangleleft C' \text{ and } (\texttt{val}\ f : T) \in C' \text{ implies } \exists\, i.\quad f_i \equiv f \\[1em] (2)\ \forall\, C', m, \overline{x}, \overline{T}, T.\quad C \triangleleft C' \text{ and } (\texttt{def}\ m(\overline{x} : \overline{T}) : T) \in C' \text{ implies} \\ \exists\, A, t.\quad C \triangleleft A \text{ and } (\texttt{def}\ m = t) \in A \\[1em] (3)\ \forall\, C', L, T_{opt}, U.\quad C \triangleleft C' \text{ and } (\texttt{type}\ L >: T_{opt} <: U) \in C' \text{ implies} \\ \exists\, A, S.\quad C \triangleleft A \text{ and } (\texttt{type}\ L = S) \in A\end{array}}{\text{isComplete}(C, \overline{f})}$$

Figure 3.12: Instance Completeness (isComplete$(C, \overline{f})$)

We conclude this section with the description of some intuitive abbreviations for typing contexts, which are used in definitions and proofs. If J is a judgment, then

$$\begin{array}{rcr} \vdash J & \text{stands for} & \texttt{root};\ \epsilon;\ \epsilon \vdash J \\ \texttt{this} : C \vdash J & \text{stands for} & C;\ \epsilon;\ \epsilon \vdash J \\ \overline{U} \vdash J & \text{stands for} & \texttt{root};\ \epsilon;\ \overline{U} \vdash J \end{array}$$

3.6 Well-formedness

The definition of well-formed programs is based on the relations of type well-formedness, member well-formedness and class well-formedness, which are summarized in Figures 3.15, 3.16 and 3.17. These relations are themselves defined in terms of the typing and subtyping relations of the previous section. The inference rules are not of particular interest except for the premises that have been added to prevent cycles in type declarations and class declarations. For instance, when checking the well-formedness of a type declaration in rule (WF-Type-Def) we check that all symbols that appear in the upper-bound T of the type symbol L are actually smaller than L with respect to the well-founded relation $\prec$ on symbols. We write this condition $\mathrm{I}(T) \prec_{mul} \{\, L \,\}$ where $\mathrm{I}(T)$ represents the multiset of symbols contained in T and $\prec_{mul}$ the multiset extension of $\prec$. Here is the formal definition of the interpretation function $\mathrm{I}(\cdot)$.

Definition 3.8 (Multiset interpretation of paths and types)
We define the interpretation $I(p)$ (resp. $I(T)$) of a path p (resp. a type T) as the multiset of symbols that compose it. We write $\{\ \}$ for the empty multiset, $\{\, e_1, \cdots, e_n \,\}$ for the multiset composed of the elements $e_1, \cdots, e_n$ and $M_1 \uplus M_2$ for the union of two multisets M_1 and M_2.

$$\begin{array}{lcl} I(\textbf{\textit{this}}, x, n, v) & = & \{\ \} \\ I(p.f) & = & I(p) \uplus \{\, f \,\} \\ I(C) & = & \{\, C \,\} \\ I(p.L) & = & I(p) \uplus \{\, L \,\} \end{array}$$

The discussion about the impact of the cyclicity checks on the soundness of the type system is deferred until Section 3.2.7. For now, we are ready to define what a well-formed program is.

$$\text{(T-This)}\ \frac{\Gamma \equiv C;\ \overline{x} : \overline{T};\ \overline{U} \qquad \Gamma \vdash C <: S}{\Gamma \vdash \texttt{this} : S}$$

$$\text{(T-Var)}\ \frac{\Gamma \equiv \Gamma_m;\ \overline{U} \qquad N = |\overline{U}| \quad n < N \qquad \Gamma \vdash U_{(N-1-n)}\uparrow^{n+1} <: S}{\Gamma \vdash n : S}$$

$$\text{(T-Param)}\ \frac{\Gamma \equiv \Gamma_o;\ \overline{x} : \overline{T};\ \overline{U} \qquad \Gamma \vdash T_i <: S}{\Gamma \vdash x_i : S}$$

$$\text{(T-Select)}\ \frac{\Gamma \vdash p : C \qquad (\texttt{val}\ f : T) \in C \qquad \Gamma \vdash T[p] <: S}{\Gamma \vdash p.f : S}$$

$$\text{(T-Call)}\ \frac{\Gamma \vdash p : C \quad (\texttt{def}\ m(\overline{x} : \overline{T}) : T) \in C \qquad \Gamma \vdash \overline{t} : \overline{T}[p] \quad \Gamma \vdash T[p] <: S}{\Gamma \vdash p.m(\overline{x} = \overline{t}) : S}$$

$$\text{(T-New)}\ \frac{\begin{array}{c}\overline{f}\ \text{disjoint} \quad \Gamma \vdash \overline{t} : \overline{U} \quad \overline{U}\ \text{closed} \\ \forall i.\ (\texttt{val}\ f_i : T)\ \text{implies}\ \texttt{this} : C \vdash_{\texttt{struct}} U_i <: T \\ \text{isComplete}(C, \overline{f}) \quad \Gamma \vdash C <: S\end{array}}{\Gamma \vdash \texttt{new}\ C(\overline{f} = \overline{t}) : S}$$

$$\text{(T-Let)}\ \frac{\Gamma \equiv \Gamma_m;\ \overline{T} \qquad \Gamma \vdash t : T \quad \Gamma_m;\ \overline{T}, T \vdash u : U \quad 0 \notin \text{fv}(U) \quad \Gamma \vdash U\downarrow <: S}{\Gamma \vdash \texttt{let}{:} T = t\ \texttt{in}\ u : S}$$

Figure 3.13: Typing ($\Gamma \vdash t : T$)

$$\text{(S-Class)}\ \frac{}{\Gamma \vdash C <: C}$$

$$\text{(S-Extends)}\ \frac{\texttt{class}\ C\ \texttt{extends}\ C'\ \{\ \overline{d}\ \} \qquad \Gamma \vdash C' <: T}{\Gamma \vdash C <: T}$$

$$\text{(S-Up)}\ \frac{\Gamma \vdash p : C \quad (\texttt{type}\ L >: T_{opt} <: U) \in C \qquad \Gamma \vdash U[p] <: S}{\Gamma \vdash p.L <: S}$$

$$\text{(S-Down)}\ \frac{\Gamma \vdash p : C \quad (\texttt{type}\ L >: T <: U) \in C \qquad \Gamma \vdash S <: T[p]}{\Gamma \vdash S <: p.L}$$

$$\text{(S-Alias-Left)}\ \frac{\Gamma \vdash p : C \qquad (\texttt{type}\ L = T) \in C \qquad \Gamma \vdash T[p] <: S}{\Gamma \vdash p.L <: S}$$

$$\text{(S-Alias-Right)}\ \frac{\Gamma \vdash p : C \qquad (\texttt{type}\ L = T) \in C \qquad \Gamma \vdash S <: T[p]}{\Gamma \vdash S <: p.L}$$

$$\text{(S-Virtual)}\ \frac{\Gamma \vdash p : C \qquad (\texttt{type}\ L >: T_{opt} <: U) \in C}{\Gamma \vdash p.L <: p.L}$$

$$\text{(S-Trans)}\ \frac{\Gamma \vdash T <: S \qquad \Gamma \vdash S <: U}{\Gamma \vdash T <: U}$$

Figure 3.14: Subtyping ($\Gamma \vdash T <: U$)

$$\text{(K-Class)}\ \frac{}{\Gamma \vdash C \text{ WF}} \qquad \text{(K-Virtual)}\ \frac{\Gamma \vdash p : C \quad (\texttt{type}\ L >: T_{opt} <: U) \in C}{\Gamma \vdash p.L \text{ WF}}$$

Figure 3.15: Type Well-formedness ($\Gamma \vdash T$ WF)

$$\text{(WF-Field-Def)}\ \frac{\texttt{this} : C \vdash T \text{ WF} \qquad \mathrm{I}(T) \prec_{mul} \{\, f \,\}}{C \vdash \texttt{val}\ f : T \text{ WF}}$$

$$\text{(WF-Type-Def)}\ \frac{\texttt{this} : C \vdash U \text{ WF} \qquad \mathrm{I}(U) \prec_{mul} \{\, L \,\} \quad T_{opt} \equiv T \text{ implies } \texttt{this} : C \vdash T \text{ WF and } \mathrm{I}(T) \prec_{mul} \{\, L \,\}}{C \vdash \texttt{type}\ L >: T_{opt} <: U \text{ WF}}$$

$$\text{(WF-Type-Val)}\ \frac{\texttt{this} : C \vdash S \text{ WF} \qquad \mathrm{I}(S) \prec_{mul} \{\, L \,\} \quad C \triangleleft C' \qquad (\texttt{type}\ L >: T_{opt} <: U) \in C' \quad \texttt{this} : C \vdash_{\texttt{struct}} S <: U \quad T_{opt} \equiv T \text{ implies } \texttt{this} : C \vdash_{\texttt{struct}} T <: S}{C \vdash \texttt{type}\ L = S \text{ WF}}$$

$$\text{(WF-Meth-Def)}\ \frac{\texttt{this} : C \vdash \overline{T}, T \text{ WF}}{C \vdash \texttt{def}\ m(\overline{x} : \overline{T}) : T \text{ WF}}$$

$$\text{(WF-Meth-Val)}\ \frac{C \triangleleft C' \qquad (\texttt{def}\ m(\overline{x} : \overline{T}) : T) \in C' \quad C;\ \overline{x} : \overline{T};\ \epsilon \vdash t : T}{C \vdash \texttt{def}\ m = t \text{ WF}}$$

Figure 3.16: Member Well-formedness ($C \vdash d$ WF)

$$\text{(WF-Class)}\ \frac{C_{opt} \equiv C' \text{ implies } C' \prec C \qquad C \vdash \overline{d} \text{ WF}}{\texttt{class}\ C\ \texttt{extends}\ C_{opt}\ \{\ \overline{d}\ \} \text{ WF}}$$

Figure 3.17: Class Well-formedness (D WF)

Definition 3.9 (Program well-formedness)
A program Π is considered well-formed when all the following properties are satisfied.

1. *$\prec$ is a well-founded relation that satisfies $(\forall\, C, f.\ C \prec f)$ and $(\forall\, C, L.\ C \prec L)$.*
2. *All classes are well-formed: $\forall\, D.\ D \in classes_{\Pi}$ implies D WF.*
3. *The main term is well-typed in the empty context: $\exists\, T.\ \vdash \Pi.main : T$.*
4. *Unicity of method valuations: there is only one valuation of a method visible from one particular class of the program.*

$$\forall\, m, t, u, C, C'.\ \left.\begin{array}{l} (\textbf{\textit{def}}\ m = t) \in C \\ (\textbf{\textit{def}}\ m = u) \in C' \\ C \lhd C' \end{array}\right\}\ \textit{implies}\ t \equiv u\ \textit{and}\ C \equiv C'$$

5. *Unicity of type valuations: there is only one valuation of a virtual type visible from one particular class of the program.*

$$\forall\, L, T, U, C, C'.\ \left.\begin{array}{l} (\textbf{\textit{type}}\ L = T) \in C \\ (\textbf{\textit{type}}\ L = U) \in C' \\ C \lhd C' \end{array}\right\}\ \textit{implies}\ T \equiv U\ \textit{and}\ C \equiv C'$$

Note that 4 ensures determinism of reduction and 5 is required for type safety.

Separate compilation. The reader could wrongly infer that requiring the existence of a well-founded relation on symbols forbids the possibility of separate compilation. Actually we do not ask the programmer to explicitly give the relation $\prec$ in his program. We show on the following example how to implement the inference of such a relation with the additional constraint of separate compilation.

```
class Object {}
class A {
  type T <: Object
  type U <: Object
}
class B extends A {
  type T = this.U
}
```

We assume we compile both classes separately. The idea is to build $\prec$ incrementally. After compilation of `A`, $\prec$ is composed of the pairs $(Object, T)$, $(Object, U)$, (A, T) and (A, U). After compilation of class `B`, $\prec$ has been extended with the pairs (A, B), (B, T), (B, U) and (U, T). As we do not require $\prec$ to be a total order, when compiling class `A` nothing forces us to choose between $T \prec U$ and $U \prec T$, which would prevent further extensions of `A`, like the class `B` if we had chosen $T \prec U$.

Refining the mechanism of symbol ordering. A global well-founded relation on symbols is sometimes too restrictive. We illustrate this point with the following example which looks perfectly valid [7] but which is rejected by our type system because it generates the inconsistent set of constraints $\{ U \prec T, T \prec U \}$.

```
class A {
  type T
  type U
}
class B1 extends A {
  type T = this.U
}
class B2 extends A {
  type U = this.T
}
```

To avoid cyclicity in type declarations the SCALA compiler locks a type symbol when analyzing its bounds or the type it is an alias for. It does the same when using its bounds or its value for the first time in a subtyping check. For instance in our example, it locks the symbol T when analyzing the type $\texttt{this}.U$ in class B_1. The analysis of this type succeeds without using the locked symbol T. At this point the lock on T is released. Then, the compiler locks U when analyzing the type $\texttt{this}.T$ in class B_2. Intuitively, this mechanism of locking enjoys similar induction principle as our relation $\prec$ because the set of unlocked symbols reduces each time we follow the bound or the value of a virtual type. We did not have time to formalize this mechanism for the calculus presented in this chapter, but we developped a similar mechanism for Featherweight-Scala [10], another SCALA calculus. Also, to simplify the formalization and the proofs we prefer to restrict ourselves to our simple solution based on a global well-founded relation.

3.7 Structured Subtyping

In rule (T-NEW), to check that the value associated with a field has a type U_i which conforms to the declared type T of the field, we have written $\texttt{this} : C \vdash_{\texttt{struct}} U_i <: T$ instead of $\texttt{this} : C \vdash U_i <: T$. The latter judgment is an instance of the subtyping relation we have already presented; the former is an instance of a restricted and more regular form of subtyping that we call *structured subtyping* [8]. In this section we give the definition of this relation, then we explain the motivations behind its introduction, and finally we show on examples that it actually corresponds to the typechecking steps performed by a compiler.

3.7.1 Definition

The only difference between general subtyping and structured subtyping is that with the latter we force in every application of the transitivity rule the interme-

[7] It is actually a valid SCALA program.

[8] It has nothing to do with structural subtyping; what is structured here is the form of subtyping derivations, not the way of comparing two types structurally.

diate type to be smaller, with respect to a type ordering $\prec_T$, than the types at both ends. Here is the formal definition.

Definition 3.10 (Structured typing and structured subtyping)
The inference rules for structured typing ($\Gamma \vdash_{\texttt{struct}} t : T$) and structured subtyping ($\Gamma \vdash_{\texttt{struct}} T <: U$) are obtained from the general ones by replacing the (S-Trans) *subtyping rule with the following* (S-Trans-S) *rule.*

$$(\text{S-Trans-S}) \quad \frac{\Gamma \vdash_{\texttt{struct}} T <: S \qquad \Gamma \vdash_{\texttt{struct}} S <: U \qquad S \prec_T T \qquad S \prec_T U}{\Gamma \vdash_{\texttt{struct}} T <: U}$$

In this definition, the ordering $\prec_T$ on types is the natural extension to types of the well-founded relation $\prec$ on symbols. Here is its precise definition.

Definition 3.11 (Type ordering)
We define the ordering $\prec_T$ on types as the multiset extension of the relation $\prec$ on symbols via the interpretation function $I(\cdot)$:

$$\frac{I(T) \prec_{mul} I(U)}{T \prec_T U}$$

3.7.2 Motivations

The concept of structured subtyping emerged from our discussion of Section 3.2 about the subtleties behind the design of a safe subtyping relation in presence of virtual types. By an inspection of several representative examples, we have presented structured subtyping as a safe definition of subtyping. Here, we want to give more technical justifications for structured subtyping, essentially by showing where it is used in the formal proof of soundness and by explaining how it makes this proof possible or simplifies it. These additional justifications for introducing structured subtyping are based on the following three observations.

Point 1. *We need the property that subtyping implies subclassing in the empty context.*

In Section 3.2.1 we have already shown that a type system where subtyping does not imply subclassing is necessarily unsafe, by exhibiting a counter-example. Here, we explain where and how this property is used in the proof.

An important part of the soundness proof consists in proving that every well-typed term that is not a value is reducible (See Theorem A.29 [Progress]). The proof of this property essentially relies on the fact that in an empty typing context the restriction of subtyping to class types is equivalent to subclassing, i.e. each time two class types are in a *subtype* relationship their corresponding classes are in a *subclass* relationship. To see why, let us look at the case (T-Select) in the progress proof (See page 123). In the hypotheses associated with this case we have a well-typed field selection $v.f$ where v is a value $\texttt{new}\ A(\overline{f} = \overline{v})$, and we want to show that the entire field selection $v.f$ can be reduced to one of the v_i's. By definition of typing for field selections, v has type C where C is the class enclosing the declaration of the field f. This implies trivially that A is a subtype

of C. By definition of typing for instance creations, v is complete, which means that it provides a value v_i for each field whose declaration is accessible from class A through class inheritance. With the property that subtyping implies subclassing in empty context, we can derive from the fact that A is a subtype of C that A is a subclass of C. Consequently, v provides a value v_i for the field f. We conclude that $v.f$ reduces to v_i by the rule (R-SELECT).

Point 2. *Considering that subtyping implies subclassing in all contexts is not generalizable.*

It might be true that in our simple calculus the property that subtyping implies subclassing in *all* contexts holds. However there is a counter-example that shows that this property would not be true in a more evolved calculus where types can be empty, namely a simple extension of our calculus with intersection types. As we want our soundness proof to be general enough, we do not want to prove a theorem that would be trivially false in some natural extension. The idea behind the counter-example is to apply the transitivity rule in a context that is not instantiable. Assume A and B are two unrelated classes that assign different and contradictory values to a virtual type T: let us say `String` for the former and `Int` for the later.

```
class A { type T = String }
class B { type T = Int }
```

Now consider the context $\Gamma = \texttt{this} : A \,\&\, B$ where $A \,\&\, B$ is an intersection type composed of types A and B. By applying the subtyping rules of Figure 3.14 we get the following subtyping derivation.

$$
\dfrac{
\dfrac{
\dfrac{\dfrac{(\texttt{this} : A \,\&\, B) \in \Gamma}{\vdots}}{\Gamma \vdash \texttt{this} : A}\,(\text{T-THIS}) \quad (\texttt{type}\ T = \texttt{String}) \in A
}{\Gamma \vdash \texttt{String} <: \texttt{this}.T}\,(\text{S-ALIAS-RIGHT})
\qquad
\dfrac{
\dfrac{\dfrac{(\texttt{this} : A \,\&\, B) \in \Gamma}{\vdots}}{\Gamma \vdash \texttt{this} : B}\,(\text{T-THIS}) \quad (\texttt{type}\ T = \texttt{Int}) \in B
}{\Gamma \vdash \texttt{this}.T <: \texttt{Int}}\,(\text{S-ALIAS-LEFT})
}{\Gamma \vdash \texttt{String} <: \texttt{Int}}\,(\text{S-TRANS})
$$

We see that in an inconsistent typing environment, namely $\texttt{this} : A \,\&\, B$, we have been able to derive that two unrelated classes, `String` and `Int`, are in the subtype relationship. Note that it does not imply unsoundness of the type system, it just says that if we provide a witness of the type $A \,\&\, B$, which is impossible in a calculus with single inheritance, then `String` can be considered a subtype of `Int` [9]. Of course, if the same counter-example was reproducible in an empty context, we would have a problem of type safety, as explained in Section 3.2.1.

The lesson of this observation is that we are not going to prove the property of Point 1 in an arbitrary typing context, but only in the empty context.

[9] By analogy with logic formalisms, the ability of deriving false deductions from inconsistent hypotheses does not imply inconsistency of the formalism.

Point 3. *Structured subtyping has the expected property of Point 1 and coincides with general subtyping in the empty context.*

The previous point has shown it is not a good idea to prove the property that subtyping implies subclassing in all contexts because it would not be generalizable to natural extensions of our calculus. So, we want to prove the property just in the empty context. We take an indirect approach that consists in defining a new subtyping relation, namely structured subtyping, that trivially satisfies the first requirement, actually in an arbitrary context (See Lemma A.21), and that coincides with the general subtyping relation in the empty context (See Lemmas A.28 and A.23). The difficult part in all these properties consists in showing that a subtyping derivation can be structured in the empty context (Lemma A.28). This result is a corollary of the most interesting lemma of this work which states that the following rule is an admissible rule of structured subtyping (See Lemma A.27).

$$\frac{\vdash_{\texttt{struct}} T <: S \qquad \vdash_{\texttt{struct}} S <: U}{\vdash_{\texttt{struct}} T <: U}$$

Compared to the transitivity rule (S-Trans-S) of structured subtyping, we observe that this rule does not impose any special condition on the intermediate type S; in compensation, all typing contexts must be empty.

In conclusion, by limiting the situations in which a transitivity step can be performed structured subtyping imposes a structure on the shape of typing derivations. This structure can then be exploited to prove important properties of the calculus, whose most important are Lemma A.21 (each time two class types are in a subtype relationship their corresponding classes are in a subclass relationship), and the strengthening lemma (Lemma A.24).

3.7.3 Implementation

From the previous section we could infer that structured subtyping is just a proof technique for demonstrating properties on the general subtyping relation. However, structured subtyping is more than a proof technique because it is also used in the definition of typing rules, namely in rules (T-New) and (WF-Type-Val). As structured subtyping is a restricted version of general subtyping, it is legitimate to wonder if its use in the typing rules does not reduce the expressiveness of the calculus. Our answer to this question is that, in practice, this is not the case. More precisely, we show in this section that a natural implementation of a type-checker for our calculus would generate only applications of the restricted transitivity rule for subtyping. To do that we start by describing the algorithm we have in mind [10].

Subtyping plays two roles. In order to convince ourselves that only the restricted transitivity rule is used in a type-checking algorithm, it is sufficient to imagine all situations where subtyping is needed. If we do that we will see that subtyping plays in fact two different roles.

Both roles are best illustrated by considering the typing rule (T-Call-alg) below, which is the algorithmic version of rule (T-Call) for typing method

[10] We actually wrote a type-checker for our calculus.

calls. This rule makes use of the judgments $\Gamma \vdash_{\texttt{alg}} t : T$ and $\Gamma \vdash_{\texttt{alg}} T <: U$, which are the algorithmic equivalents of the standard typing and subtyping judgments. When checking that the type U of the prefix p is a subtype of the class C that contains the declaration of the method m, subtyping is used to express a mechanism of *lookup*. When checking that the types $\overline{U}$ of the actual arguments of the method call conform to the expected types $\overline{T}[p]$ of the method, subtyping is used to *check the conformance* between two types that are already known at that point of the analysis.

$$\text{(T-CALL-ALG)}\quad \frac{\Gamma \vdash_{\texttt{alg}} p : U \qquad \Gamma \vdash_{\texttt{alg}} U <: C \qquad (\texttt{def}\ m(\overline{x} : \overline{T}) : T) \in C \qquad \Gamma \vdash_{\texttt{alg}} \overline{t} : \overline{U} \qquad \Gamma \vdash_{\texttt{alg}} \overline{U} <: \overline{T}[p]}{\Gamma \vdash_{\texttt{alg}} p.m(\overline{x} = \overline{t}) : T[p]}$$

In the first case, we know a type and we want to find a supertype that satisfies a certain predicate (being a class type that contains a particular member); in the second case we have two types and we want to verify that one is more precise than the other. This difference of role is expressed with a different implementation for each kind of algorithmic subtyping: the algorithmic subtyping premise $\Gamma \vdash_{\texttt{alg}} T <: C$ is implemented as an instance of a lookup relation $\Gamma \vdash_{\texttt{lookup}} T <: U$, while the algorithmic subtyping premise $\Gamma \vdash_{\texttt{alg}} \overline{U} <: \overline{T}[p]$ is implemented as an instance of another relation $\Gamma \vdash_{\texttt{check}} T <: U$. The relation $\Gamma \vdash_{\texttt{lookup}} T <: U$ is typically implemented as a function that takes a type T and returns an optional type U such that T is a subtype of U. The relation $\Gamma \vdash_{\texttt{check}} T <: U$ is typically implemented as a function that takes two types, T and U, and that returns the boolean value TRUE if T is a subtype of U and FALSE otherwise.

Now, we informally explain how $\Gamma \vdash_{\texttt{lookup}} T <: U$ and $\Gamma \vdash_{\texttt{check}} T <: U$ can be implemented just with structured subtyping, i.e. using only the restricted transitivity rule (S-TRANS-S). The implementations of both relations are more easily formulated with the concepts of type expansion and type lowering.

Type expansion. A *type expansion* step consists in taking the most direct supertype of a given type. For instance, if the direct superclass of a class C is another class C', the one-step expansion of the class type C is the class type C'. And the one-step expansion of a virtual type $p.L$ is obtained either by taking the value attached to this type in the current typing context or the declared bound of the type otherwise. More formally, we say that T expands to U in the context Γ, and we write $\Gamma \vdash T \xrightarrow{<:} U$, if the subtyping judgment $\Gamma \vdash T <: U$ can be derived by successive applications of the rules (S-CLASS), (S-VIRTUAL), (S-EXTENDS), (S-UP) and (S-ALIAS-LEFT). Such a relation can be easily formalized with inference rules; as an example we just give one rule, namely the rule that lets us expand a virtual type using its bound.

$$\text{(E-UP)}\quad \frac{\Gamma \vdash_{\texttt{alg}} p : C \qquad \Gamma \vdash C \xrightarrow{<:} C' \qquad (\texttt{type}\ L >: T_{opt} <: U) \in C' \qquad \Gamma \vdash U[p] \xrightarrow{<:} S}{\Gamma \vdash p.L \xrightarrow{<:} S}$$

Type lowering. A *type lowering* step consists in taking the most direct subtype of a given type. Similarly to type expansion, type lowering can be characterized by subtyping derivations that correspond to successive applications of the rules (S-CLASS), (S-VIRTUAL), (S-DOWN) and (S-ALIAS-RIGHT). Note that a class type cannot be lowered further since the rule (S-EXTENDS) is not part of the rules that can be used to lower a type. We write $\Gamma \vdash U \overset{<:}{\longleftarrow} T$ for denoting that T can be lowered to U. As for type expansion, such a relation can be formalized by a set of inference rules; as an example we just give the rule that lets us lower a virtual type using a type alias.

$$\text{(L-ALIAS)}\ \frac{\Gamma \vdash_{\texttt{alg}} p : C \qquad \Gamma \vdash C \overset{<:}{\longrightarrow} C' \qquad (\texttt{type}\ L = T) \in C' \qquad \Gamma \vdash S \overset{<:}{\longleftarrow} T[p]}{\Gamma \vdash S \overset{<:}{\longleftarrow} p.L}$$

Lookup implementation. A lookup can be performed using only type expansion: from a given type T, we expand it until reaching a type U that satisfies a given predicate.

$$\text{(LOOKUP)}\ \frac{\Gamma \vdash T \overset{<:}{\longrightarrow} U}{\Gamma \vdash_{\texttt{lookup}} T <: U}$$

Conformance check implementation. Checking that a type T is more precise than a type U can be performed by using type expansion and type lowering: more precisely, we expand type T until we reach type U. If this fails we compute a type U' through a one-step type lowering from U and we expand type T until we reach type U', etc. At the end, either we found a type S such that T expands to S and U can be lowered to S, or we failed.

$$\text{(CONFORMANCE)}\ \frac{\Gamma \vdash T \overset{<:}{\longrightarrow} S \qquad \Gamma \vdash S \overset{<:}{\longleftarrow} U}{\Gamma \vdash_{\texttt{check}} T <: U}$$

The following lemma states that type expansion and type lowering are instances of structured subtyping, and that both relations decrease the measure on types.

Lemma 3.1 (Type expansion and type lowering)

$$\begin{array}{lll} \Gamma \vdash T \overset{<:}{\longrightarrow} U & \textit{implies} & \Gamma \vdash_{\texttt{struct}} T <: U \textit{ and } (U \equiv T \textit{ or } U \prec_T T) \\ \Gamma \vdash U \overset{<:}{\longleftarrow} T & \textit{implies} & \Gamma \vdash_{\texttt{struct}} U <: T \textit{ and } (U \equiv T \textit{ or } U \prec_T T) \end{array}$$

Proof: The proof that type expansion and type lowering correspond to structured subtyping is trivial since their respective definitions forbid the use of the transitivity rule. The fact that both relations decrease the measure on types can be shown by an easy induction on the relations.

Qed.

We can now prove our initial claim that the subtyping steps performed by a natural type-checking algorithm are all instances of structured subtyping.

Lemma 3.2 (Algorithmic rules use structured subtyping)

$$\begin{array}{lll} \Gamma \vdash_{\texttt{lookup}} T <: U & \textit{implies} & \Gamma \vdash_{\texttt{struct}} T <: U \\ \Gamma \vdash_{\texttt{check}} T <: U & \textit{implies} & \Gamma \vdash_{\texttt{struct}} T <: U \end{array}$$

Proof: The proof is easy assuming the previous lemma. For the lookup relation, by definition $\Gamma \vdash_{\texttt{lookup}} T <: U$ implies that $\Gamma \vdash T \xrightarrow{<:} U$, and we conclude directly by Lemma 3.1. For the conformance relation, by definition $\Gamma \vdash_{\texttt{check}} T <: U$ implies that $\Gamma \vdash T \xrightarrow{<:} S$ and $\Gamma \vdash S \xleftarrow{<:} U$. We distinguish then two cases. Either the intermediate type S is equal to T (resp. to U) and in this case we conclude using the property of Lemma 3.1 on type lowering (resp. type expansion). Or S is different from both T and U. By the previous lemma, S is also smaller than both T and U with respect to the type ordering $\prec_T$, so we can apply the restricted rule (S-TRANS-S) to conclude that T is a subtype of U.

Qed.

Summary. What has been explained until now in this section can be summarized by the diagrams of Figure 3.18. The diagram presents the two roles of subtyping with their implementation: a left-to-right arrow represents a type expansion and a right-to-left arrow represents a type lowering. The fact that arrows are always descending emphasizes the property that both relations decrease the measure on types.

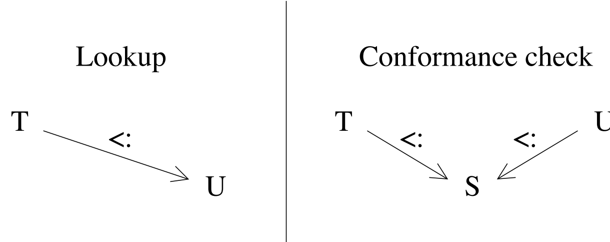


Figure 3.18: Roles of subtyping in a type-checker

Example. Let us illustrate all this development with a simple example. In the following example, if we want to verify that type alias `type` $U = B$ is well-formed, we must check that, in the context of class C, type B is a subtype of `this`.T.

```
class A
class B extends A
class C {
  type T
  type U <: this.T
  type T = A
  type U = B
}
```

Type B can be expanded to A using rule (S-EXTENDS) followed by rule (S-CLASS).

$$\text{(S-EXTENDS)}\ \frac{\texttt{class}\ B\ \texttt{extends}\ A\ \{\ \}\qquad \text{(S-CLASS)}\ \dfrac{}{\texttt{this} : C \vdash_{\texttt{struct}} A <: A}}{\texttt{this} : C \vdash_{\texttt{struct}} B <: A}$$

Type $\texttt{this}.T$ can be lowered to type A using rule (S-ALIAS-RIGHT) followed by rule (S-CLASS).

$$\text{(S-ALIAS-RIGHT)}\ \frac{\texttt{this} : C \vdash_{\texttt{struct}} \texttt{this} : C \qquad (\texttt{type}\ T = A) \in C \qquad \text{(S-CLASS)}\ \dfrac{}{\texttt{this} : C \vdash_{\texttt{struct}} A <: A}}{\texttt{this} : C \vdash_{\texttt{struct}} A <: \texttt{this}.T}$$

At this point we would like to apply rule (S-TRANS-S) to derive that B is a subtype of $\texttt{this}.T$:

$$\text{(S-TRANS-S)}\ \frac{\begin{array}{cc} A \prec_T B & A \prec_T \texttt{this}.T \\ \texttt{this} : C \vdash_{\texttt{struct}} B <: A & \texttt{this} : C \vdash_{\texttt{struct}} A <: \texttt{this}.T \end{array}}{\texttt{this} : C \vdash_{\texttt{struct}} B <: \texttt{this}.T}$$

We still need to check that $A \prec_T B$ and $A \prec_T \texttt{this}.T$. We already said that, by construction, expanding or lowering a type always results in a smaller type w.r.t. $\prec_T$. We observe that it is actually the case in this example. We have $A \prec_T B$ since B is a subclass of A and since well-formedness of class declarations (rule WF-CLASS) implies $A \prec B$. We have $A \prec_T \texttt{this}.T$ because the well-formedness of the type alias $\texttt{type}\ T = A$ (rule WF-TYPE-VAL) enforces $\mathrm{I}(A) \prec_{mul} \{T\}$ (note also that according to Point 1 of Definition 3.9, it is always the case that $A \prec T$).

Conclusion. The main idea we want to convey in this section is that it is not natural for a type-checker to use the full transitivity rule. The restricted transitivity rule that we propose forbids absurd actions, like for instance passing through a virtual type in order to show that a class type is a subtype of another class type. The presence of structured subtyping in the general typing rules, namely in rules (T-NEW) and (WF-TYPE-VAL) remains to be justified. Actually we have no good reasons here, apart that it makes the proofs of admissibility of transitivity simpler and that it does not diminish in practice the expressiveness of the calculus.

3.8 Compatibility of Bounds

Virtual types can be declared both with a lower-bound and an upper-bound. In this section we examine what properties of these bounds are needed for proving the soundness of our type system. We also discuss two alternative ways of enforcing these properties in the type system and we justify our choice of one of them.

For deciding what relation should statically exist between the lower-bound and the upper-bound of a virtual type we adopt a pragmatic approach; we start writing the proof with an open mind and see what hypotheses are needed to complete it. In the way we organize our proof, the interaction between an upper-bound and the corresponding lower-bound appears in the case (S-Down, S-Up) of the *admissibility of transitivity for structured subtyping* proof (Lemma A.27, page 119). In this case we must show that a type T is a subtype of a type U under the following set of hypotheses.

$$\boxed{\begin{array}{c} \vdash_{\texttt{struct}} p : C \quad (\texttt{type}\ L >: T' <: U') \in C \\ \vdash_{\texttt{struct}} T <: T'[p] \quad \vdash_{\texttt{struct}} U'[p] <: U \end{array}}$$

For proving our goal there are now two natural possibilities we can think of. The first one consists in requiring in the well-formedness of the type declaration of L that $\texttt{this} : C \vdash_{\texttt{struct}} T' <: U'$. With such an assumption we can apply a substitution lemma in order to obtain $\vdash_{\texttt{struct}} T'[p] <: U'[p]$. Then, $T'[p]$ and $U'[p]$ being smaller that $p.L$ w.r.t. the type ordering $\prec_T$ we can apply several times the induction hypothesis and prove our goal $\vdash_{\texttt{struct}} T <: U$. The second possibility consists in requiring for each class that is used for creating an instance the existence of a type valuation for each visible type declaration; if such type valuation $\texttt{type}\ L = S$ exists in a class A, its well-formedness implies that $\texttt{this} : A \vdash_{\texttt{struct}} T' <: S$ and $\texttt{this} : A \vdash_{\texttt{struct}} S <: U'$. By applying the substitution lemma and the induction hypothesis we are able to finish the proof of our goal as we did with the first possibility. Let us examine now the difference between these two approaches.

Pessimistic approach. In the first case we check explicitly for the compatibility between bounds at the type declaration site. The advantage of this approach is that it allows an early detection of classes that are impossible to instantiate due to an inconsistency in the bounds of one of their type members. We illustrate this scenario with the following example where it is impossible to assign a value to T because there is no type both supertype of B and subtype of A.

```
class B
class A extends B
class C {
  type T >: B <: A
}
```

This is the approach adopted by the Scala compiler. What is important is to prevent the existence of a path expression p of type C. If such an expression exists, we can deduce that B is a subtype of $p.T$ and that $p.T$ is a subtype of A, and according to the second observation of Section 3.2.1, such a situation is unsafe. Note also that such an expression, namely `null`, was at the basis of the Scala compiler bug explained in Section 1.1 of the introductory chapter. Rejecting totally the class C, as the Scala compiler does, is a rough but efficient means of achieving the goal that such an expression p cannot exist.

Another advantage of this approach is that, in principle, it is not necessary to forbid the instantiation of classes with abstract type members. The reason

is that the compiler can always insert a type valuation when one is missing by choosing either the declared lower-bound or the declared upper-bound. This type valuation would be automatically well-formed since the bounds are compatible. Note that the SCALA compiler does not implement this optimization. The reason is that SCALA implements F-bounded polymorphism [8], i.e. it allows the bound of a type member `T` to depend of itself, like in `type T <: List[T]`. By assigning its bound to the type member `T`, we would end up with a recursive type definition `type T = List[T]`. In principle, recursive types are not problematic, but they would certainly complicate the type system, the arguments for its soundness and the design of a terminating type-checking algorithm.

Optimistic approach. In the second approach we also check for the compatibility between bounds but we do it indirectly by supplying a value for the virtual type. As the value must be both a subtype of the upper-bound and a supertype of the lower-bound the compatibility between the bounds is automatically enforced. The advantage of such a solution is that it makes the type system more expressive: there are programs that cannot be typed with the first approach but that can be typed with the second one. The following program is an example of such a scenario. It reuses the classes A and B of the previous example. What is important to note is that it is impossible to check for compatibility of the bounds of T in class C because they are both abstract. Actually we do not care about the compatibility of bounds at this point because class C has still abstract members and cannot be instantiated. Bounds must only be proved compatible in concrete classes, as in class D where all type members have been assigned a value. The compatibility of bounds L (lower) and U (upper) is indirectly enforced by the fact that the value A assigned to T is both a subtype of U and a supertype of L.

```
class C {
  type L
  type U
  type T >: Low <: Up
}
class D extends C {
  type L = A
  type U = B
  type T = A
}
```

In our type system we chose the second approach that enforces bound compatibility indirectly through the requirement that no instantiated class can have abstract type members (See premise isComplete$(C, \overline{f})$ in rule (T-NEW)). The reason of this choice is that it makes the type system more expressive, as we just showed. But taking the other approach would require only a minor modification of our proof; the beginning of this section informally explains what proof steps must be taken in this case.

Finally it is interesting to remark that there is no concept of type equality in our type system, we only have subtyping. Actually a type valuation `type` $L = T$ is used in the subtyping rules as an additional declaration of L with lower-bound and upper-bound both equal to T. From this consideration we could design a

type system where type valuations would be replaced with the possibility of overriding the bounds of a virtual type in a subclass one or more times. Once again we did not make this design choice because it was not consistent with our treatment of methods where overriding is forbidden.

3.9 Soundness Proof

Following a now well-established approach [25] for proving type safety of calculi with small-step operational semantics we decompose our proof of soundness into two lemmas: progress and subject-reduction. Progress ensures that a well-typed term that is not a value is reducible, subject-reduction ensures that the type of a term is preserved by reduction. All lemmas and proofs are collected in Appendix A. Every lemma is immediately followed by its proof. Furthermore the proof of a lemma only refers to lemmas that have been previously defined (and proved).

3.10 Conclusion

3.10.1 Summary and Future Work

In this chapter we present a completely formal, although not machine-checked, proof of soundness for a simple extension of FJ with virtual types. Contrary to the classical representation of programs as abstract syntax trees, our proof is based on a more abstract representation of programs as sets of functions over symbols; this representation is more appropriate to the mechanization of the proof in a proof assistant. We identify an interesting example that shows that a cyclicity between type declarations may lead to a subtyping relation which is no longer consistent with the class hierarchy; this breaks type safety. Our solution to this problem is to impose the existence of a global well-founded relation on symbols which is consistent with type and class declarations. We also define a restricted subtyping relation, called structured subtyping, which prevents absurd uses of the rule of transitivity of subtyping. This relation makes proofs simpler and is shown equivalent to the general subtyping relation in the empty typing context.

To our knowledge, this work represents the first completely formal argument in favor of the soundness of virtual types. However, it cannot be considered as a proof of soundness of SCALA virtual types, because in this language, virtual types are used in a more general setting. In order to generalize our proof to the use of virtual types in SCALA, we would have to consider the following extensions.

Reducible paths. First, we must release the limitation that virtual types can only be selected on irreducible terms. This forbids interesting examples where a type is selected on a filed member. For instance, suppose we have the following class `ListModule` which represents the lists of integers as an abstract data type.

```
class ListModule {
  type T
  def head(xs: this.T): Int
```

```
    def tail(xs: this.T): this.T
    def isEmpty(xs: this.T): Boolean
    def nil(): this.T
    def cons(x: Int, xs: this.T): this.T
  }
```

In this class, the abstract type `T` is the type of integer lists, the abstract method `head` takes as input a list, i.e. an element of `this.T`, and returns the first element of this list, etc. Suppose also we have several different implementations of a list module. Later, we may want to implement a `sort` function that is not bound to a particular list module implementation. Parameterizing the sort function by a list module implementation can be achieved by enclosing the function `sort` in a class `Sort` that declares a field `lm` of type `ListModule`, as shown below. In this case, both the argument type and the return type of the method `sort` are the selection of a type member `T` on the field `this.lm`.

```
  class Sort {
    val lm: ListModule
    def sort(xs: this.lm.T): this.lm.T = ...
  }
```

Having reducible terms embedded inside types complicates the type system. With sufficient conditions on the use of field symbols we think it is possible to prove a lemma that states that typing derivations can be normalized, i.e. transformed into equivalent derivations that do no longer contain reducible terms inside types. After this normalization we could in principle apply the proof techniques that we develop in our proof.

Class types with refinements The types of our calculus are not rich enough to express parametric polymorphism. We are able to translate the type parameter X of a class C by a virtual type declaration, but it is of limited interest if we do not extend at the same time the syntax of types so that we can speak of the value of X in an occurrence of a C class type. In SCALA such types are written $C\{\,\texttt{type}\ X >: T <: U\,\}$, they represent all instances x of class C such that the value of X in x is both a supertype of T and a subtype of U.

Here we extend the syntax of types so that class types can now contain constraints on virtual types. We call the resulting types *class types with refinements.*

Constraint	R	$::=$	$\texttt{type}\ L >: T_{opt} <: U$
Type	T	$::=$	$C\{\,\overline{R}\,\}\ \mid\ p.L$

New rules must be added to deal with class types with refinements. In Figure 3.19, we present a set of natural typing and subtyping rules. We do not know exactly how to extend the current proof for dealing with refinements in class types, in particular it is not clear how to define the interpretation function $\mathrm{I}(\cdot)$ for class types with refinements.

Inner classes. Contrary to our calculus, SCALA has inner classes. Unfortunately, the concepts of inner classes and virtual types are not orthogonal. A detailed explanation of their interaction and the description of a static analysis

$$(\text{S-EXTENDS})\ \frac{C \lhd D \qquad \text{for all } (\texttt{type } L >: T_{opt} <: U) \in \overline{R}', \quad \Gamma_m;\ \overline{S}, C\{\overline{R}\} \vdash 0.L <: U\uparrow^1 \text{ and} \quad T_{opt} \equiv T \text{ implies } \Gamma_m;\ \overline{S}, C\{\overline{R}\} \vdash T\uparrow^1 <: 0.L}{\Gamma_m;\ \overline{S} \vdash C\{\overline{R}\} <: D\{\overline{R}'\}}$$

$$(\text{S-REF-LEFT})\ \frac{\Gamma \vdash p : C\{\overline{R}\} \quad (\texttt{type } L >: T_{opt} <: U) \in \overline{R} \quad \Gamma \vdash U <: S}{\Gamma \vdash p.L <: S}$$

$$(\text{S-REF-RIGHT})\ \frac{\Gamma \vdash p : C\{\overline{R}\} \quad (\texttt{type } L >: T <: U) \in \overline{R} \quad \Gamma \vdash S <: T}{\Gamma \vdash S <: p.L}$$

$$(\text{T-NEW})\ \frac{\text{(as before)}}{\Gamma \vdash \texttt{new } C(\overline{f} = \overline{t}) : C\{\ \}}$$

Figure 3.19: Rules for class types with refinements

can be found in [3]. Here we just give an idea of the kind of mechanism that is needed to typecheck virtual types in the presence of inner classes. To illustrate this we consider a SCALA version of the example of Section 1.2.1.4 augmented with a type member `T` in class `A` and a field `x` in class `X`.

```
abstract class A {
  type T <: Object
  abstract class X {
    val outerX = A.this
    val x: T              // T  =  A.this.T  =  this.outerX.T
  }
}
class B extends A {
  type T = String
  class Y extends X {
    val outerY = B.this
    val x = "foo"
  }
}
```

The code above and in particular the assignment of field `x` in class `Y` is well-typed. This might be a bit surprising because the field `x` is declared in class `X` with the type `T` and `T` is an abstract type member of class `A`. The exact value of `T` is not known, at least not in class `X`. So, how could one assign `"foo"` to `x` in one of its subclasses? The answer comes from the observation we made in Section 1.2.1.4: for all instances of class `Y`, the fields `outerX` and `outerY` hold the same value. The field `x` is declared with the type `T`. In class `X`, `T` is a shorthand for `A.this.T` and by definition `A.this` is equal to `this.outerX`, so `T`

is equal to `this.outerX.T`. As for any instance of `Y`, `outerX` is equal to `outerY`, we conclude that in class `Y`, the field `x` has the type `this.outerY.T` which is obviously equal to `String`. The assignment of `"foo"` to `x` in class `Y` is therefore well-typed.

This example illustrates the fact that typing virtual types in the presence of inner classes requires some kind of alias analysis on outer fields. In this example, it is possible to establish that `x` has the type `String` in class `Y` only because it was possible to establish that, for any instance of `Y`, the fields `outerX` and `outerY` hold the same value. Without that information, one would only know that `x` has the type `T` and that `T` is bound by `Object`.

3.10.2 Evaluation Criteria

For evaluating our calculus with virtual types, we consider three criteria that we think are the most important for a type system: expressiveness, safety and decidability.

The *expressiveness* of a type system is its ability to accept non-trivial programs. This property cannot be proved formally because there is no formal definition of what a non-trivial program is. The easiest way of gaining confidence in the expressiveness of a type system is to write a type-checker whose implementation follows closely the typing rules and to use it for testing some programs examples. However this is not completely satisfactory because it can happen that, due to a bug in its implementation, the type-checker accepts a program that is actually ill-typed. A simple solution to this problem consists in having the type-checker generate formal typing proofs for the programs it accepts. A standard type-checker either says `Yes` if it accepts the input program or `No` if it rejects it; in the latter case it also signals the violated typing rule. Instead of simply saying `Yes`, a type-checker that generates typing proofs returns also a formal proof that the input program is actually well-typed; such a proof can then be checked by a proof assistant. This approach requires to instrument all analysis functions of the type-checker, so that they additionally build proof trees. Unfortunately, we had no time to write a typing-proof generator for the calculus presented in this chapter but we applied successfully the method on another calculus which formalizes Scala. More details can be found in [11]. Finally, note that expressiveness must not be confused with Turing-completeness. Turing-completeness means that the calculus is sufficient to encode Turing machines, but it says nothing on how easy it is to write interesting examples in a readable and compact way.

By definition, a type system is *safe* if all the programs that are accepted execute correctly. For instance, it cannot happen that, during the execution of an accepted program, a method call is impossible to resolve because the method is not present in the receiver object. The property of safety can be proved formally and we actually proved it for our calculus.

A type system is *decidable* if the set of programs it accepts is computable by an algorithm, i.e. there must exist an algorithm that terminates with `Yes` if the input program is accepted by the type system and that terminates with `No` otherwise. Decidability is a property that can be proved formally. It is of course a good idea to have a decidable type system, but in general it is a mistake to try to prove type safety directly on the decidable type system. It is often necessary to find a more general type system, maybe undecidable, but on which proofs

by induction can be made. This is the same idea as generalizing a theorem in order to make a proof by induction succeed.

Now we can estimate the value our calculus with respect to these three criteria. The weak point of our calculus is definitely its expressiveness. It would become much more expressive if we could extend it with the missing features previously presented: reducible paths, class types with refinements and inner classes. The strong point of our calculus is to have a rigorous proof of safety. Finally, we conjecture that the type system is decidable. In order to prove this statement we would need to formalize the typing algorithm that we have presented in Section 3.7.3 and show that it is equivalent to the typing rules of the calculus.

3.10.3 Related Work

Several works aim at formalizing virtual types. Here is a short overview of these works.

In [18] the authors present some foundations for virtual types. They model objects and classes containing type members in a typed lambda-calculus which is a combination of well-understood and sound features, namely subtyping, type operators, fixed points, dependent functions and dependent records. The authors believe that the interaction between these features should not break type safety, but give no proof. They have no mechanisms similar to ours for preventing cycles in the graph of type symbols and for restricting the application of the transitivity rule. In our calculus such mechanisms are used for proving some properties that are crucial for type safety. However, it does not mean their type system is unsound because it could be that these properties are satisfied for other reasons or simply that their calculus is not expressive enough to encode our counter-examples.

The title of [22] is "Virtual Types are Statically Safe". The paper is a bit disappointing in this regard because it only contains a syntax and a set of typing rules for a calculus with virtual types; the author does not try to formally state nor prove a soundness theorem, only informal arguments are given in favor of type safety. We think the title must be understood in the context of a time where the current belief was that it was not possible to design a type system for virtual types that is both sound and static. The presented calculus is close to ours, in particular it has the restriction that virtual types are only selected on the current instance of a class. In our calculus, we have a similar restriction: virtual types are only selected on terms that are not reducible, namely variables, parameters, values and the current instance.

In [20] the authors present ν-OBJ, a calculus of objects with inner classes, multiple inheritance and type fields, which is far more general than the one we use for our proof of virtual types. They completely formalize its semantics and type system, and present a proof sketch of its soundness. ν-OBJ is indeed a good candidate as a core calculus for SCALA and a good argument in favor of the soundness of SCALA and of virtual types in general. However, there are small points where ν-OBJ fails to be the ultimate answer to these questions. There are two main problems. First, ν-OBJ has a syntax which is completely different from SCALA. For instance, ν-OBJ has no primitive concept of class and represents a program as a term, rather than as a set of classes, like SCALA. The paper informally describes how to encode the main features of SCALA into ν-OBJ. It

is proposed to encode a class by two things: a type declaration and a class template declaration. Since both declarations include the complete signature of the original class, such an encoding is necessarily exponential in the depth of the class nesting. The complexity of the encoding, the complexity of the ν-OBJ syntax and the complexity of the typing rules make that it is unclear if the result of encoding a valid SCALA program is still a valid ν-OBJ term; it is actually not feasible to manually encode and typecheck even trivial SCALA programs. Since no type-checker has ever been implemented for ν-OBJ, contrary to our calculus, the question of the existence of such a type-preserving encoding is still open. Our calculus does not suffer this problem since it is almost a syntactical subset of SCALA. Furthermore, our calculus shares with SCALA the intuitive representation of a class as a fixed entity that can be consulted but never modified during evaluation. Since, in the ν-OBJ encoding, classes are embedded in terms, they may potentially be rewritten and even copied during evaluation. The second problem of ν-OBJ is the degree of formalization of its proof of soundness. Contrary to our case, mechanizing the soundness proof of ν-OBJ is not conceivable in the short term.

Our main source of inspiration for designing the type system and finding the intermediate lemmas of the proof has been [9]. The authors present a proof of soundness for a lambda-calculus with dependent types, subtyping and late-bound overloading in a very rigorous and didactic style. They identify and solve the same problem of cyclicity between definitions we have in our calculus, namely having the typing relation depend on subtyping and subtyping depend on typing. It is not surprising since virtual types are a kind of dependent types: they depend on the object on which they are selected.

In [15], the authors prove the soundness of a calculus with virtual classes called *vc*. Virtual classes are classes that can be defined in a class and overridden in a subclass, they form the basis of the programming language GBETA. Overriding classes automatically inherit from the overridden classes. *vc*'s virtual classes and our virtual types are not directly comparable. The main difference is that virtual types eventually receive a type value, whereas virtual classes stay open to future extensions. It means that no assumption can ever be made on the exact value of a virtual class, which limits the expressiveness of the calculus. The language GBETA has the concept of *final-binding* for classes: a final class cannot be extended further. Such a mechanism is similar to the type assignments of our formalism. Unfortunaly, final-bindings for virtual classes are not part of the *vc* calculus.

In [6, 7], the authors present a safe solution to encode a particular and interesting use of virtual types, namely the possibility of extending several classes that are mutually recursive, in parallel. The solution consists in defining groups of classes and adapting the treatment of class parameters accordingly. This work is not directly comparable to ours because it imposes strong constraints on the way a virtual type can be used. These constraints are necessary to let the authors interpret virtual types as type parameters of classes.

Appendix A

Complete Proof of Soundness

A.1 Miscellaneous

Lemma A.1 (Admissibility of subsumption) $\Gamma \vdash t : T$ *and* $\Gamma \vdash T <: U$ *implies* $\Gamma \vdash t : U$.

Proof: By cases on $\Gamma \vdash t : T$ by applying rule (S-TRANS) each time. We show just one case.
Case (T-THIS). The hypotheses are:

$$t \equiv \mathtt{this} \quad \Gamma \equiv C;\ \overline{x} : \overline{T};\ \overline{U} \quad \Gamma \vdash C <: T$$

1. By rule (S-TRANS), $\Gamma \vdash C <: T$ and $\Gamma \vdash T <: U$ implies $\Gamma \vdash C <: U$.
2. We conclude with rule (T-THIS).

Qed.

Lemma A.2 (Self in self substitution)

1. $\mathtt{this} \in q[p]$ *implies* $\mathtt{this} \in p$.
2. $\mathtt{this} \in T[p]$ *implies* $\mathtt{this} \in p$.

Proof: By mutual induction on q and T.
Case (THIS). The hypothesis is: $q \equiv \mathtt{this}$

1. $q[p] \equiv \mathtt{this}[p] \equiv p$.
2. By hypothesis $\mathtt{this} \in q[p]$, so $\mathtt{this} \in p$.

Case (PARAM). The hypothesis is: $q \equiv x$

1. $q[p] \equiv x[p] \equiv x$.
2. By hypothesis $\mathtt{this} \in q[p]$, so $\mathtt{this} \in x$, which is false.

Case (VAR). Similar to case (PARAM).
Case (SELECT). The hypothesis is: $q \equiv q'.f$

1. $q[p] \equiv q'.f[p] \equiv q'[p].f$.
2. $\texttt{this} \in q'[p].f$ implies $\texttt{this} \in q'[p]$.
3. By (IH), $\texttt{this} \in p$.

Case (Value). The hypothesis is: $\boxed{q \equiv v}$

1. $q[p] \equiv v[p] \equiv v$.
2. By hypothesis $\texttt{this} \in q[p]$, so $\texttt{this} \in v$, which is false.

Case (ClassType). Similar to case (Param).
Case (VirtualType). Similar to case (Select).

Qed.

Lemma A.3 (Free parameters are in the context)

1. $\Gamma_o;\ \overline{x} : \overline{T};\ \overline{U} \vdash t : T$ *implies* $fp(t, T) \subset \{\,\overline{x}\,\}$.
2. $\Gamma_o;\ \overline{x} : \overline{T};\ \overline{U} \vdash T <: U$ *implies* $fp(T, U) \subset \{\,\overline{x}\,\}$.
3. $\Gamma_o;\ \overline{x} : \overline{T};\ \overline{U} \vdash T \text{ WF}$ *implies* $fp(T) \subset \{\,\overline{x}\,\}$.

Proof: Property 1 and 2 by mutual induction on the typing and subtyping relations using when necessary that $\mathrm{fp}(T{\uparrow^k}) \equiv \mathrm{fp}(T)$. Property 3 is proved by cases on the kinding relation using Property 1.

Qed.

Lemma A.4 (Free variables are in the context)

1. $\Gamma_m;\ \overline{U} \vdash t : T$ *implies* $\forall\, k \in fv(t, T).\ k < |\overline{U}|$.
2. $\Gamma_m;\ \overline{U} \vdash T <: U$ *implies* $\forall\, k \in fv(T, U).\ k < |\overline{U}|$.
3. $\Gamma_m;\ \overline{U} \vdash T \text{ WF}$ *implies* $\forall\, k \in fv(T).\ k < |\overline{U}|$.

Proof: We prove the more general lemma

1. $\Gamma_m;\ \overline{U} \vdash t : T$ implies $\forall\, d \le |\overline{U}|.\ \forall\, k \in \mathrm{fv}_d(t, T).\ k < |\overline{U}| - d$.
2. $\Gamma_m;\ \overline{U} \vdash T <: U$ implies $\forall\, d \le |\overline{U}|.\ \forall\, k \in \mathrm{fv}_d(T, U).\ k < |\overline{U}| - d$.

by mutual induction on $\Gamma_m;\ \overline{U} \vdash t : T$ and $\Gamma_m;\ \overline{U} \vdash T <: U$, and then we take $d = 0$. Property 3 is proved by cases on the kinding relation using Property 1.

Case (T-This). The hypotheses are

$$\boxed{t \equiv \texttt{this} \quad \Gamma_m \equiv C;\ \overline{x} : \overline{T} \quad \Gamma_m;\ \overline{U} \vdash C <: T}$$

1. Let $d \le |\overline{U}|$.
2. $\mathrm{fv}_d(t) = \mathrm{fv}_d(\texttt{this}) = \emptyset$, so $\forall\, k \in \mathrm{fv}_d(t).\ k < |\overline{U}| - d$.
3. By (IH), $\Gamma_m;\ \overline{U} \vdash C <: T$ implies $\forall\, k \in \mathrm{fv}_d(T).\ k < |\overline{U}| - d$.

Case (T-PARAM). Similar to case (T-THIS).
Case (T-VAR). The hypotheses are

$$t \equiv n \quad N = |\overline{U}| \quad n < N \quad \Gamma_m;\ \overline{U} \vdash U_{(N-1-n)}\uparrow^{n+1} <: T$$

1. Let $d \leq |\overline{U}|$.
2. By (IH), $\Gamma_m;\ \overline{U} \vdash U_{(N-1-n)}\uparrow^{n+1} <: T$ implies $\forall\, k \in \mathrm{fv}_d(T).\ k < |\overline{U}|$.
3. We reason by cases on n and d.

   Case $(n < d)$.

   (a) $\mathrm{fv}_d(t) = \mathrm{fv}_d(n) = \emptyset$.
   (b) $\forall\, k \in \emptyset.\ k < |\overline{U}| - d$.

   Case $(n \geq d)$.

   (a) $\mathrm{fv}_d(t) = \mathrm{fv}_d(n) = \{\, n - d \,\}$.
   (b) $k \in \mathrm{fv}_d(t)$ implies $k \equiv n - d$.
   (c) $N = |\overline{U}|$ and $n < N$ implies $n - d < |\overline{U}| - d$.

Case (T-SELECT). The hypotheses are

$$t \equiv p.f \quad \Gamma_m;\ \overline{U} \vdash p : C \quad (\mathtt{val}\ f : S) \in C \quad \Gamma_m;\ \overline{U} \vdash S[p] <: T$$

1. Let $d \leq |\overline{U}|$.
2. $\mathrm{fv}_d(t) = \mathrm{fv}_d(p.f) = \mathrm{fv}_d(p)$.
3. By (IH), $\Gamma_m;\ \overline{U} \vdash p : C$ implies $\forall\, k \in \mathrm{fv}_d(p).\ k < |\overline{U}| - d$, so $\forall\, k \in \mathrm{fv}_d(t).\ k < |\overline{U}| - d$.
4. By (IH), $\Gamma_m;\ \overline{U} \vdash S[p] <: T$ implies $\forall\, k \in \mathrm{fv}_d(T).\ k < |\overline{U}| - d$.

Case (T-CALL). The hypotheses are

$$t \equiv p.m(\overline{x} = \overline{t}) \quad \Gamma_m;\ \overline{U} \vdash p : C \quad (\mathtt{def}\ m(\overline{x} : \overline{S}) : S) \in C$$
$$\Gamma_m;\ \overline{U} \vdash \overline{t} : \overline{S}[p] \quad \Gamma_m;\ \overline{U} \vdash S[p] <: T$$

1. Let $d \leq |\overline{U}|$.
2. $\mathrm{fv}_d(t) = \mathrm{fv}_d(p.m(\overline{x} = \overline{t})) = \mathrm{fv}_d(p) \cup \mathrm{fv}_d(\overline{t})$.
3. By (IH), $\Gamma_m;\ \overline{U} \vdash p : C$ implies $\forall\, k \in \mathrm{fv}_d(p).\ k < |\overline{U}| - d$.
4. By (IH), $\Gamma_m;\ \overline{U} \vdash \overline{t} : \overline{S}[p]$ implies $\forall\, k \in \mathrm{fv}_d(\overline{t}).\ k < |\overline{U}| - d$.
5. By (IH), $\Gamma_m;\ \overline{U} \vdash S[p] <: T$ implies $\forall\, k \in \mathrm{fv}_d(T).\ k < |\overline{U}| - d$.

Case (T-NEW). Similar to case (T-CALL).
Case (T-LET). The hypotheses are

$$t \equiv \mathtt{let}{:}\, T_1 = t_1\ \mathtt{in}\ t_2 \quad \Gamma_m;\ \overline{U} \vdash t_1 : T_1$$
$$\Gamma_m;\ \overline{U}, T_1 \vdash t_2 : T_2 \quad 0 \notin \mathrm{fv}(T_2) \quad \Gamma_m;\ \overline{U} \vdash T_2{\downarrow} <: T$$

1. Let $d \leq |\overline{U}|$.
2. $\text{fv}_d(t) = \text{fv}_d(\texttt{let:} T_1 = t_1 \texttt{ in } t_2) = \text{fv}_d(T_1) \cup \text{fv}_d(t_1) \cup \text{fv}_{d+1}(t_2)$.
3. By (IH), $\Gamma_m;\ \overline{U} \vdash t_1 : T_1$ implies $\forall\, k \in \text{fv}_d(t_1) \cup \text{fv}_d(T_1).\ k < |\overline{U}| - d$.
4. By (IH), $\Gamma_m;\ \overline{U}, T_1 \vdash t_2 : T_2$ implies $\forall\, d' \leq |\overline{U}| + 1.\ \forall\, k \in \text{fv}_{d'}(t_2).\ k < |\overline{U}| + 1 - d'$.
5. $d \leq |\overline{U}|$ implies $d + 1 \leq |\overline{U}| + 1$.
6. By taking $d' = d + 1$ in the previous result, $\forall\, k \in \text{fv}_{d+1}(t_2).\ k < |\overline{U}| + 1 - (d + 1)$, i.e. $k < |\overline{U}| - d$.
7. By (IH), $\Gamma_m;\ \overline{U} \vdash T_2{\downarrow} <: T$ implies $\forall\, k \in \text{fv}_d(T).\ k < |\overline{U}| - d$.

Case (S-Class). The hypotheses are $\boxed{T \equiv C \quad U \equiv C}$

1. Let $d \leq |\overline{U}|$.
2. $\text{fv}_d(T) = \text{fv}_d(C) = \emptyset$ and $\text{fv}_d(U) = \text{fv}_d(C) = \emptyset$, so $\forall\, k \in \text{fv}_d(T) \cup \text{fv}_d(U).\ k < |\overline{U}| - d$.

Case (S-Extends). The hypotheses are

$$\boxed{T \equiv C \quad \texttt{class}\ C\ \texttt{extends}\ C'\ \{\ \overline{d}\ \} \quad \Gamma_m;\ \overline{U} \vdash C' <: U}$$

1. Let $d \leq |\overline{U}|$.
2. $\text{fv}_d(T) = \text{fv}_d(C) = \emptyset$, so $\forall\, k \in \text{fv}_d(T).\ k < |\overline{U}| - d$.
3. By (IH), $\Gamma_m;\ \overline{U} \vdash C' <: U$ implies $\forall\, k \in \text{fv}_d(U).\ k < |\overline{U}| - d$.

Case (S-Virtual). The hypotheses are

$$\boxed{T \equiv p.L \quad U \equiv p.L \quad \Gamma_m;\ \overline{U} \vdash p : C \quad (\texttt{type}\ L >: T_{opt} <: U) \in C}$$

1. Let $d \leq |\overline{U}|$.
2. $\text{fv}_d(T) = \text{fv}_d(U) = \text{fv}_d(p.L) = \text{fv}_d(p)$.
3. By (IH), $\Gamma_m;\ \overline{U} \vdash p : C$ implies $\forall\, k \in \text{fv}_d(p).\ k < |\overline{U}| - d$.

Case (S-Trans). The hypotheses are

$$\boxed{\Gamma_m;\ \overline{U} \vdash T <: S \quad \Gamma_m;\ \overline{U} \vdash S <: U \quad S \prec_T T, U}$$

1. Let $d \leq |\overline{U}|$.
2. By (IH), $\Gamma_m;\ \overline{U} \vdash T <: S$ implies $\forall\, k \in \text{fv}_d(T).\ k < |\overline{U}| - d$.
3. By (IH), $\Gamma_m;\ \overline{U} \vdash S <: U$ implies $\forall\, k \in \text{fv}_d(U).\ k < |\overline{U}| - d$.

Case (S-Alias-Left). The hypotheses are

$$\boxed{\begin{array}{c} T \equiv p.L \quad \Gamma_m;\ \overline{U} \vdash p : C \\ (\texttt{type}\ L = S) \in C \quad \Gamma_m;\ \overline{U} \vdash S[p] <: U \end{array}}$$

1. Let $d \leq |\overline{U}|$.
2. $\mathrm{fv}_d(T) = \mathrm{fv}_d(p.L) = \mathrm{fv}_d(p)$.
3. By (IH), $\Gamma_m;\ \overline{U} \vdash\ p : C$ implies $\forall\, k \in \mathrm{fv}_d(p).\ k < |\overline{U}| - d$.
4. By (IH), $\Gamma_m;\ \overline{U} \vdash\ S[p] <: U$ implies $\forall\, k \in \mathrm{fv}_d(U).\ k < |\overline{U}| - d$.

Case (S-Alias-Right), (S-Up), (S-Down). Similar to case (S-Alias-Left).

Qed.

Lemma A.5 (Values are irreducible) *v irreducible.*

Proof: By induction on v.

Qed.

Lemma A.6 (Some terms are not typable in empty context)

$$\vdash\ t : T \textit{ implies } t \not\equiv \texttt{this} \textit{ and } t \not\equiv x \textit{ and } t \not\equiv n$$

Proof: By simple case analysis on t.

Qed.

Lemma A.7 (Paths typable in empty context are values)

$$\vdash\ p : T \textit{ implies } p \equiv v$$

Proof: By definition, a path p is either `this`, x, n or a value v. Lemma A.6 tells us that only the last alternative is possible.

Qed.

Lemma A.8 (Chaining self substitutions)

1. $t[p][v] \equiv t[p[v]]$.

2. $T[p][v] \equiv T[p[v]]$.

Proof: By mutual induction on t and T.

Qed.

Lemma A.9 (Chaining self and parameter substitutions)

1. $\overline{x} \notin fp(t)$ *implies* $t[p][\overline{x}\backslash\overline{v}] \equiv t[p[\overline{x}\backslash\overline{v}]]$.

2. $\overline{x} \notin fp(T)$ *implies* $T[p][\overline{x}\backslash\overline{v}] \equiv T[p[\overline{x}\backslash\overline{v}]]$.

Proof: By mutual induction on t and T.

Case (This). $\boxed{t \equiv \texttt{this}}$

1. On one hand $t[p][\overline{x}\backslash\overline{v}] \equiv \texttt{this}[p][\overline{x}\backslash\overline{v}] \equiv p[\overline{x}\backslash\overline{v}]$.
2. On the other hand $t[p[\overline{x}\backslash\overline{v}]] \equiv \texttt{this}[p[\overline{x}\backslash\overline{v}]] \equiv p[\overline{x}\backslash\overline{v}]$.

Case (Var). $\boxed{t \equiv n}$

1. On one hand $n[p][\overline{x}\backslash\overline{v}] \equiv n$.

2. On the other hand $n[p[\overline{x}\backslash\overline{v}]] \equiv n$.

Case (PARAM). $\boxed{t \equiv x}$

1. $\overline{x} \notin \mathrm{fp}(t)$ implies $x \notin \overline{x}$.
2. On one hand $t[p][\overline{x}\backslash\overline{v}] \equiv x[p][\overline{x}\backslash\overline{v}] \equiv x[\overline{x}\backslash\overline{v}] \equiv x$ (since $x \notin \overline{x}$).
3. On the other hand $t[p[\overline{x}\backslash\overline{v}]] \equiv x[p[\overline{x}\backslash\overline{v}]] \equiv x$.

Case (SELECT, CALL, NEW, LET, CLASSTYPE, VIRTUALTYPE). By applying (IH) on case hypotheses followed by the application of the same rule.

Qed.

Lemma A.10 (Chaining self and variable substitutions)

1. *fv*$(t) = \emptyset$ *implies* $t[p][k := v] \equiv t[p[k := v]]$.
2. *fv*$(T) = \emptyset$ *implies* $T[p][k := v] \equiv T[p[k := v]]$.

Proof: By mutual induction on t and T.

Case (THIS). $\boxed{t \equiv \texttt{this}}$

1. On one hand $t[p][k := v] \equiv \texttt{this}[p][k := v] \equiv p[k := v]$.
2. On the other hand $t[p[k := v]] \equiv \texttt{this}[p[k := v]] \equiv p[k := v]$.

Case (VAR). $\boxed{t \equiv n}$

1. Case impossible because $\mathrm{fv}(n) = \{\, n \,\}$ and by hypothesis $\mathrm{fv}(t) = \emptyset$.

Case (PARAM). $\boxed{t \equiv x}$

1. On one hand $t[p][k := v] \equiv x[p][k := v] \equiv x[k := v] \equiv x$.
2. On the other hand $t[p[k := v]] \equiv x[p[k := v]] \equiv x$.

Case (SELECT, CALL, NEW, LET, CLASSTYPE, VIRTUALTYPE). By applying (IH) on case hypotheses followed by the application of the same rule.

Qed.

A.2 Type Ordering

Lemma A.11 (Interpretation and substitution)

1. *$\mathtt{this} \in q$ implies $I(q[p]) \equiv I(q) \uplus I(p)$.*
2. *$\mathtt{this} \in T$ implies $I(T[p]) \equiv I(T) \uplus I(p)$.*

Proof: Property 1 by induction on q, then Property 2 by cases on T using Property 1.

Case ($q \equiv \mathtt{this}$).

1. On one hand, $\mathrm{I}(q[p]) \equiv \mathrm{I}(\mathtt{this}[p]) \equiv \mathrm{I}(p)$.
2. On the other hand, $\mathrm{I}(q) \uplus \mathrm{I}(p) \equiv \mathrm{I}(\mathtt{this}) \uplus \mathrm{I}(p) \equiv \{\ \} \uplus \mathrm{I}(p) \equiv \mathrm{I}(p)$.

Case ($q \equiv x, n, v$).

1. Cases impossible because $\mathtt{this} \notin q$.

Case ($q \equiv q'.f$).

1. $\mathtt{this} \in q$ implies $\mathtt{this} \in q'$.
2. By (IH), $\mathtt{this} \in q'$ implies $\mathrm{I}(q'[p]) \equiv \mathrm{I}(q') \uplus \mathrm{I}(p)$.
3. On one hand, $\mathrm{I}(q[p]) \equiv \mathrm{I}(q'.f[p]) \equiv \mathrm{I}(q'[p].f) \equiv \mathrm{I}(q'[p]) \uplus \{ f \} \equiv \mathrm{I}(q') \uplus \mathrm{I}(p) \uplus \{ f \}$.
4. On the other hand, $\mathrm{I}(q) \uplus \mathrm{I}(p) \equiv \mathrm{I}(q'.f) \uplus \mathrm{I}(p) \equiv \mathrm{I}(q') \uplus \{ f \} \uplus \mathrm{I}(p)$.
5. We conclude using associativity of multiset union.

Case ($T \equiv C$).

1. Cases impossible because $\mathtt{this} \notin T$.

Case ($T \equiv q.L$).

1. $\mathtt{this} \in T$ implies $\mathtt{this} \in q$.
2. By Property 1, $\mathtt{this} \in q$ implies $\mathrm{I}(q[p]) \equiv \mathrm{I}(q) \uplus \mathrm{I}(p)$.
3. On one hand, $\mathrm{I}(T[p]) \equiv \mathrm{I}(q.L[p]) \equiv \mathrm{I}(q[p].L) \equiv \mathrm{I}(q[p]) \uplus \{ L \} \equiv \mathrm{I}(q) \uplus \mathrm{I}(p) \uplus \{ L \}$.
4. On the other hand, $\mathrm{I}(T) \uplus \mathrm{I}(p) \equiv \mathrm{I}(q.L) \uplus \mathrm{I}(p) \equiv \mathrm{I}(q) \uplus \{ L \} \uplus \mathrm{I}(p)$.
5. We conclude using associativity of multiset union.

Qed.

Lemma A.12 (Interpretation and substitution of a value)

$$I(T[v]) \equiv I(T)$$

Proof: By cases on $\mathtt{this} \in T$.

Case (YES).

1. By Lemma A.11, $\mathtt{this} \in T$ implies $\mathrm{I}(T[v]) \equiv \mathrm{I}(T) \uplus \mathrm{I}(v)$.
2. $\mathrm{I}(v) \equiv \{\ \}$ implies $\mathrm{I}(T) \uplus \mathrm{I}(v) \equiv \mathrm{I}(T)$.

Case (NO).

1. $\mathtt{this} \notin T$ implies $T[v] \equiv T$, implies $\mathrm{I}(T[v]) \equiv \mathrm{I}(T)$.

Qed.

Lemma A.13 (Declarations and type ordering)

1.	$(\textit{\textbf{type}}\ L >: T_{opt} <: U)$	*implies*	$T_{opt}[p], U[p] \prec_T p.L$
2.	$(\textit{\textbf{type}}\ L = T)$	*implies*	$T[p] \prec_T p.L$
3.	$(\textit{\textbf{class}}\ C\ \textit{\textbf{extends}}\ C'\ \{\ \overline{d}\ \})$	*implies*	$C' \prec_T C$

Proof: We start with the proof of Property 1.

1. By well-formedness of declarations, the declaration $(\mathtt{type}\ L >: T_{opt} <: U)$ implies $\mathrm{I}(U) \prec_{mul} \{ L \}$.
2. $\mathrm{I}(p.L) \equiv \mathrm{I}(p) \uplus \{ L \}$.
3. We prove $\mathrm{I}(U[p]) \prec_{mul} \mathrm{I}(p.L)$ by cases on $\mathtt{this} \in p$.

   Case (YES).

   (a) By Lemma A.11, $\mathtt{this} \in p$ implies $\mathrm{I}(U[p]) \equiv \mathrm{I}(U) \uplus \mathrm{I}(p)$.
   (b) $\mathrm{I}(U) \prec_{mul} \{ L \}$ implies $\mathrm{I}(U) \uplus \mathrm{I}(p) \prec_{mul} \mathrm{I}(p) \uplus \{ L \}$.

   Case (NO).

   (a) $\mathtt{this} \notin p$ implies $U[p] \equiv U$.
   (b) $\mathrm{I}(U[p]) \equiv \mathrm{I}(U)$.
   (c) $\mathrm{I}(U) \prec_{mul} \{ L \}$ implies $\mathrm{I}(U) \prec_{mul} \mathrm{I}(p) \uplus \{ L \}$.

4. By definition of type ordering, $\mathrm{I}(U[p]) \prec_{mul} \mathrm{I}(p.L)$ implies $U[p] \prec_T p.L$.
5. If $T_{opt} \equiv T$, there is a completely similar proof that $T[p] \prec_T p.L$.

Proof of Property 2 is completely similar to proof of Property 1.

We conclude with proof of Property 3.

1. By well-formedness of declarations, $(\mathtt{class}\ C\ \mathtt{extends}\ C'\ \{\ \overline{d}\ \})$ implies $C' \prec C$.
2. $\mathrm{I}(C) \equiv \{ C \}$ and $\mathrm{I}(C') \equiv \{ C' \}$.
3. $C' \prec C$ implies $\{ C' \} \prec_{mul} \{ C \}$.
4. By definition of type ordering, $\mathrm{I}(C') \prec_{mul} \mathrm{I}(C)$ implies $C' \prec_T C$.

Qed.

Lemma A.14 (Type ordering and substitution of a value) *$T \prec_T U$ implies $T[v] \prec_T U[v]$.*

Proof:

1. $T \prec_T U$ implies $\mathrm{I}(T) \prec_{mul} \mathrm{I}(U)$.
2. By Lemma A.12, $\mathrm{I}(T[v]) \equiv \mathrm{I}(T)$ and $\mathrm{I}(U[v]) \equiv \mathrm{I}(U)$.
3. $\mathrm{I}(T[v]) \prec_{mul} \mathrm{I}(U[v])$ implies $T[v] \prec_T U[v]$.

Qed.

Lemma A.15 (Facts about type ordering)

1. *$T \prec_T C$ implies $T \equiv C'$.*
2. *$C \prec_T p.L$.*

Proof: We prove Property 1 by cases on T.
Case ($T \equiv C'$). Done.
Case ($T \equiv p.L$).

1. $p.L \prec_T C$ implies $\mathrm{I}(p.L) \prec_{mul} \mathrm{I}(C)$.
2. $\mathrm{I}(p.L) \equiv \mathrm{I}(p) \uplus \{ L \}$, $\mathrm{I}(C) \equiv \{ C \}$.
3. $\mathrm{I}(p) \uplus \{ L \} \prec_{mul} \{ C \}$ implies $L \prec C$.
4. By well-formedness of programs, $C \prec L$.
5. There exists an infinite decreasing chain of symbols $\cdots \prec L \prec C \prec L \prec C$, which contradicts well-foundedness of $\prec$.

Proof of Property 2.

1. $\mathrm{I}(C) \equiv \{ C \}$, $\mathrm{I}(p.L) \equiv \mathrm{I}(p) \uplus \{ L \}$.
2. By well-formedness of programs, $C \prec L$.
3. $C \prec L$ implies $\{ C \} \prec_{mul} \mathrm{I}(p) \uplus \{ L \}$.
4. $\mathrm{I}(C) \prec_{mul} \mathrm{I}(p.L)$ implies $C \prec_T p.L$.

Qed.

Lemma A.16 (Multiset extension preserves well-foundedness) *The multiset extension R_{mul} of a well-founded relation R is well-founded.*

Proof: Standard lemma.

Lemma A.17 (Type ordering is well-founded) *The relation $\prec_T$ is well-founded.*

Proof: We reason by absurd.

1. Suppose we have an infinite sequence of types $(T_i)_i$ that is decreasing for the type ordering $\prec_T$.

2. By definition of $\prec_T$ we have also an infinite sequence of type multisets $(\mathrm{I}(T_i))_i$ that is decreasing for the multiset extension $\prec_{mul}$ of symbol ordering $\prec$.

3. By Lemma A.16, the relation $\prec_{mul}$ is well-founded, so such sequence cannot exist.

Qed.

Lemma A.18 (The multiset extension of type ordering is well-founded) *The multiset extension $\prec_{T\,mul}$ of type ordering $\prec_T$ is well-founded.*

Proof: By Lemma A.17, the relation $\prec_T$ is well-founded. By Lemma A.16, its multiset extension is also well-founded.

Qed.

A.3 Subclassing

Lemma A.19 (Transitivity of subclassing) *$C \triangleleft C'$ and $C' \triangleleft C''$ implies $C \triangleleft C''$.*

Proof: By induction on $C \triangleleft C'$.

Qed.

Lemma A.20 (Subclassing defines a hierarchy)

$$C \triangleleft C_1 \textit{ and } C \triangleleft C_2 \textit{ implies } C_1 \triangleleft C_2 \textit{ or } C_2 \triangleleft C_1$$

Proof: By induction on $C \triangleleft C_1$. In case (SC-Class), $C \equiv C_1$, so $C \triangleleft C_2$ implies $C_1 \triangleleft C_2$. In case (SC-Extends), $\texttt{class}\ C\ \texttt{extends}\ C'\ \{\ \overline{d}\ \}$ and $C' \triangleleft C_1$. We reason now by cases on $C \triangleleft C_2$. In case (SC-Class), $C \equiv C_2$, so $C \triangleleft C_1$ implies $C_2 \triangleleft C_1$. In case (SC-Extends), $\texttt{class}\ C\ \texttt{extends}\ C'\ \{\ \overline{d}\ \}$ and $C' \triangleleft C_2$. By induction hypothesis, $C_1 \triangleleft C_2$ or $C_2 \triangleleft C_1$.

Qed.

Lemma A.21 (Subtyping implies subclassing) *$\Gamma \vdash_{\texttt{struct}} C <: C'$ implies $C \triangleleft C'$.*

Proof: By induction on $\Gamma \vdash_{\texttt{struct}} C <: C'$. The only applicable rules are (S-Class), (S-Extends) and (S-Trans-S). Case (S-Class) is resolved with (SC-Class). Case (S-Extends) is a straightforward application of the induction hypothesis followed by (SC-Extends). For (S-Trans-S), we know that the intermediate type S satisfied $S \prec_T C, C'$. By Lemma A.15 (Property 1), it follows that S is necessarily a class type C''. By applying twice the induction hypothesis we get $C \triangleleft C''$ and $C'' \triangleleft C'$. We conclude using transitivity of subclassing (Lemma A.19).

Qed.

Lemma A.22 (Subclassing implies subtyping) *$C \triangleleft C'$ implies $\Gamma \vdash_{\texttt{struct}} C <: C'$.*

Proof: By induction on $C \triangleleft C'$ using rules (S-Class) and (S-Extends).

Qed.

A.4 Admissibility of Transitivity

Lemma A.23 (Unstructuring derivations)

1. $\Gamma \vdash_{\texttt{struct}} t : T$ *implies* $\Gamma \vdash t : T$.
2. $\Gamma \vdash_{\texttt{struct}} T <: U$ *implies* $\Gamma \vdash T <: U$.

Proof: By simple mutual induction on $\Gamma \vdash_{\texttt{struct}} t : T$ and $\Gamma \vdash_{\texttt{struct}} T <: U$. The idea is just to replace every application of rule (S-Trans-S) with an application of rule (S-Trans).

Qed.

Lemma A.24 (Strengthening)

1. $\texttt{this} : C \vdash_{\texttt{struct}} A <: T$ *and* $\texttt{this} \notin T$ *implies* $\vdash_{\texttt{struct}} A <: T$.
2. $\texttt{this} : C \vdash_{\texttt{struct}} v : T$ *and* $\texttt{this} \notin T$ *implies* $\vdash_{\texttt{struct}} v : T$.

Proof: By mutual induction on $\texttt{this} : C \vdash_{\texttt{struct}} A <: T$ and $\texttt{this} : C \vdash_{\texttt{struct}} v : T$.

Case (S-Trans-S). The hypotheses are:

$$\texttt{this} : C \vdash_{\texttt{struct}} A <: S \quad \texttt{this} : C \vdash_{\texttt{struct}} S <: T \quad S \prec_T A, T$$

1. By Lemma A.15 (Property 1), $S \prec_T A$ implies $S \equiv B$.
2. By (IH), $\texttt{this} : C \vdash_{\texttt{struct}} A <: B$ and $\texttt{this} \notin B$ implies $\vdash_{\texttt{struct}} A <: B$.
3. By (IH), $\texttt{this} : C \vdash_{\texttt{struct}} B <: T$ and $\texttt{this} \notin T$ implies $\vdash_{\texttt{struct}} B <: T$.
4. By rule (S-Trans-S), $\vdash_{\texttt{struct}} A <: T$.

Case (S-Class). The hypothesis is: $\boxed{T \equiv A}$

1. By rule (S-Class), $\vdash_{\texttt{struct}} A <: A$.

Case (S-Extends). The hypotheses are:

$$\texttt{class}\ A\ \texttt{extends}\ A' \ \{\ \overline{d}\ \} \quad \texttt{this} : C \vdash_{\texttt{struct}} A' <: T$$

1. By (IH), $\texttt{this} : C \vdash_{\texttt{struct}} A' <: T$ and $\texttt{this} \notin T$ implies $\vdash_{\texttt{struct}} A' <: T$.
2. By rule (S-Extends), $\vdash_{\texttt{struct}} A <: T$.

Case (S-Virtual, S-Up, S-Alias-Left). These cases are impossible because the left-hand type in the subtyping judgment should be both a class type and a virtual type.

Case (S-Alias-Right). The hypotheses are:

$$T \equiv p.L \quad \texttt{this} : C \vdash_{\texttt{struct}} p : C'$$
$$(\texttt{type}\ L = U) \in C' \quad \texttt{this} : C \vdash_{\texttt{struct}} A <: U[p]$$

1. $T \equiv p.L$ and $\texttt{this} \notin T$ implies $\texttt{this} \notin p$.
2. By Lemma A.2, $\texttt{this} \notin p$ implies $\texttt{this} \notin U[p]$.

3. By (IH), $\texttt{this} : C \vdash_{\texttt{struct}} A <: U[p]$ and $\texttt{this} \notin U[p]$ implies $\vdash_{\texttt{struct}} A <: U[p]$.

4. By definition of paths, p is either $\texttt{this}$, a parameter x, a variable n or a value v. $\texttt{this} \notin p$ implies p is not $\texttt{this}$. p well-typed in a context $(\texttt{this} : C)$ without parameter nor variable bindings implies p is not a parameter nor a variable. So necessarily p is a value v.

5. By (IH), $\texttt{this} : C \vdash_{\texttt{struct}} v : C'$ and $\texttt{this} \notin C'$ implies $\vdash_{\texttt{struct}} v : C'$.

6. By rule (S-ALIAS-RIGHT), $\vdash_{\texttt{struct}} A <: v.L$.

Case (S-DOWN). Similar to case (S-ALIAS-RIGHT).

For the typing of $\texttt{this} : C \vdash_{\texttt{struct}} v : T$, the only applicable rule is the following.
Case (T-NEW). The hypotheses are:

$$v \equiv \texttt{new}\ B(\overline{f} = \overline{v}) \quad \overline{f} \text{ disjoint} \quad \texttt{this} : C \vdash_{\texttt{struct}} \overline{v} : \overline{U} \quad \overline{U} \text{ closed}$$
$$\forall\, i, S.\ (\texttt{val}\ f_i : S) \text{ implies } \texttt{this} : B \vdash_{\texttt{struct}} U_i <: S$$
$$\text{isComplete}(B, \overline{f}) \quad \texttt{this} : C \vdash_{\texttt{struct}} B <: T$$

1. $\overline{U}$ closed implies $\texttt{this} \notin \overline{U}$.

2. By (IH), $\texttt{this} : C \vdash_{\texttt{struct}} \overline{v} : \overline{U}$ and $\texttt{this} \notin \overline{U}$ implies $\vdash_{\texttt{struct}} \overline{v} : \overline{U}$.

3. By (IH), $\texttt{this} : C \vdash_{\texttt{struct}} B <: T$ and $\texttt{this} \notin T$ implies $\vdash_{\texttt{struct}} B <: T$.

4. By rule (T-NEW), $\vdash_{\texttt{struct}} \texttt{new}\ B(\overline{f} = \overline{v}) : T$.

Qed.

Lemma A.25 (Admissibility of transitivity for class type subtyping)
$\Gamma \vdash_{\texttt{struct}} A <: B$ *and* $\Gamma' \vdash_{\texttt{struct}} B <: C$ *implies* $\Gamma'' \vdash_{\texttt{struct}} A <: C$.

Proof:

1. By Lemma A.21, $\Gamma \vdash_{\texttt{struct}} A <: B$ implies $A \triangleleft B$.

2. By Lemma A.21, $\Gamma' \vdash_{\texttt{struct}} B <: C$ implies $B \triangleleft C$.

3. By transitivity of subclassing (Lemma A.19), $A \triangleleft B$ and $B \triangleleft C$ implies $A \triangleleft C$.

4. By Lemma A.22, $A \triangleleft C$ implies $\Gamma'' \vdash_{\texttt{struct}} A <: C$.

Qed.

Lemma A.26 (Substitution for self in structured derivations)
Suppose $\vdash_{\texttt{struct}} v : C$.

1. $\texttt{this} : C \vdash_{\texttt{struct}} p : A$ *implies* $\vdash_{\texttt{struct}} p[v] : A$.

2. $\texttt{this} : C \vdash_{\texttt{struct}} T <: U$ *implies* $\vdash_{\texttt{struct}} T[v] <: U[v]$.

Proof: [1]
We first prove Property 1 by cases on p.

Case (This). The hypotheses are: $p \equiv \texttt{this} \quad \texttt{this} : C \vdash_{\texttt{struct}} C <: A$

1. $p[v] \equiv \texttt{this}[v] \equiv v$.
2. By hypothesis, $\vdash_{\texttt{struct}} v : C$.
3. $\vdash_{\texttt{struct}} v : C$ implies

   $v \equiv \texttt{new}\ B(\overline{f} = \overline{v}) \quad \overline{f}$ disjoint $\quad \vdash_{\texttt{struct}} \overline{v} : \overline{U} \quad \overline{U}$ closed
   $\forall\, i, S.\ (\texttt{val}\ f_i : S)$ implies $\texttt{this} : B \vdash_{\texttt{struct}} U_i <: S$
   isComplete$(B, \overline{f}) \quad \vdash_{\texttt{struct}} B <: C$

4. By admissibility of transitivity for class types (Lemma A.25), $\vdash_{\texttt{struct}} B <: C$ and $\texttt{this} : C \vdash_{\texttt{struct}} C <: A$ implies $\vdash_{\texttt{struct}} B <: A$.
5. By rule (T-New), $\vdash_{\texttt{struct}} v : A$.

Case (Value). The hypothesis is: $p \equiv w$

1. By definition of substitution, $p[v] \equiv w[v] \equiv w$ because $\texttt{this} \notin w$.
2. By strengthening lemma A.24, $\texttt{this} : C \vdash_{\texttt{struct}} w : A$ implies $\vdash_{\texttt{struct}} w : A$.

Case (Param). The hypothesis is: $p \equiv x$

1. Case impossible because x does not appear in the typing context.

Case (Var). The hypothesis is: $p \equiv n$

1. Case impossible because the typing context for variables is empty, which means we should have $n < 0$.

Now we prove Property 2 by induction on $\texttt{this} : C \vdash_{\texttt{struct}} T <: U$.

Case (S-Trans-S). The hypotheses are:

$\texttt{this} : C \vdash_{\texttt{struct}} T <: S \quad \texttt{this} : C \vdash_{\texttt{struct}} S <: U \quad S \prec_T T, U$

1. By (IH), $\vdash_{\texttt{struct}} T[v] <: S[v]$ and $\vdash_{\texttt{struct}} S[v] <: U[v]$.
2. By Lemma A.14, $S \prec_T T, U$ implies $S[v] \prec_T T[v], U[v]$.
3. By rule (S-Trans-S), $\vdash_{\texttt{struct}} T[v] <: U[v]$.

[1] Note that for proving a more general lemma where A is replaced with an arbitrary type T, we need the admissibility of transitivity. But for proving the latter we need the former. With the present version of the lemma there is no such circularity because we just need admissibility of transitivity for class types, which can be proved independently (Lemma A.25).

Case (S-CLASS). Straightforward.
Case (S-EXTENDS). Straightforward application of the (IH).
Case (S-VIRTUAL). Straightforward application of the (IH).
Case (S-ALIAS-LEFT).
The hypotheses are:

$$T \equiv p.L \quad \texttt{this} : C \vdash_{\texttt{struct}} p : C' \\ (\texttt{type}\ L = S) \in C' \quad \texttt{this} : C \vdash_{\texttt{struct}} S[p] <: U$$

1. $(p.L)[v] \equiv p[v].L$.
2. By Property 1, $\texttt{this} : C \vdash_{\texttt{struct}} p : C'$ implies $\vdash_{\texttt{struct}} p[v] : C'$.
3. By (IH), $\texttt{this} : C \vdash_{\texttt{struct}} S[p] <: U$ implies $\vdash_{\texttt{struct}} S[p][v] <: U[v]$.
4. By Lemma A.8, $S[p][v] \equiv S[p[v]]$.
5. By rule (S-ALIAS-LEFT), $\vdash_{\texttt{struct}} p[v].L <: U[v]$.

Case (S-ALIAS-RIGHT, S-DOWN, S-UP). Similar to case (S-ALIAS-LEFT).
Qed.

Lemma A.27 (Admissibility of transitivity for structured subtyping)

$\vdash_{\texttt{struct}} T <: S$ *and* $\vdash_{\texttt{struct}} S <: U$ *implies* $\vdash_{\texttt{struct}} T <: U$.

Proof: By induction on the multiset extension $\prec_{T mul}$ of type ordering, which is well-founded after Lemma A.18. We reason by cases on the last subtyping rules used to derive $\vdash_{\texttt{struct}} T <: S$ and $\vdash_{\texttt{struct}} S <: U$ with the hypothesis that the property is true for every type multiset smaller than $\{ T, S, U \}$.

Case (S-EXTENDS,?). The hypotheses are:

$$T \equiv C \quad \texttt{class}\ C\ \texttt{extends}\ C'\ \{\ \overline{d}\ \} \quad \vdash_{\texttt{struct}} C' <: S$$

1. By Lemma A.13 (Property 3), $(\texttt{class}\ C\ \texttt{extends}\ C'\ \{\ \overline{d}\ \})$ implies $C' \prec_T C$.
2. $C' \prec_T C$ implies $\{ C', S, U \} \prec_{T mul} \{ C, S, U \}$.
3. By (IH), $\vdash_{\texttt{struct}} C' <: S$ and $\vdash_{\texttt{struct}} S <: U$ implies $\vdash_{\texttt{struct}} C' <: U$.
4. By (S-EXTENDS), $\vdash_{\texttt{struct}} C <: U$.

Case (S-UP,?). The hypotheses are:

$$T \equiv p.L \quad \vdash_{\texttt{struct}} p : C \\ (\texttt{type}\ L >: T_{opt} <: T') \in C \quad \vdash_{\texttt{struct}} T'[p] <: S$$

1. By Lemma A.13, $(\texttt{type}\ L >: T_{opt} <: T')$ implies $T'[p] \prec_T p.L$.
2. $T'[p] \prec_T p.L$ implies $\{ T'[p], S, U \} \prec_{T mul} \{ p.L, S, U \}$.
3. By (IH), $\vdash_{\texttt{struct}} T'[p] <: S$ and $\vdash_{\texttt{struct}} S <: U$ implies $\vdash_{\texttt{struct}} T'[p] <: U$.

4. By (S-UP), $\vdash_{\texttt{struct}} p.L <: U$.

Case (?, S-DOWN). Symmetric to case (S-UP,?).
Case (S-CLASS,?). The hypotheses are: $\boxed{T \equiv C \quad S \equiv C}$

1. $T \equiv C$ and $S \equiv C$ implies $S \equiv T$.
2. By replacement, $\vdash_{\texttt{struct}} S <: U$ implies $\vdash_{\texttt{struct}} T <: U$.

Case (S-VIRTUAL,?). The hypotheses are:

$$\boxed{T \equiv p.L \quad S \equiv p.L \quad \vdash_{\texttt{struct}} p : C \quad (\texttt{type}\ L >: T_{opt} <: T') \in C}$$

1. $T \equiv p.L$ and $S \equiv p.L$ implies $S \equiv T$.
2. By replacement, $\vdash_{\texttt{struct}} S <: U$ implies $\vdash_{\texttt{struct}} T <: U$.

Case (?,S-CLASS). Symmetric to case (S-CLASS,?).
Case (?,S-VIRTUAL). Symmetric to case (S-VIRTUAL,?).
Case (S-ALIAS-LEFT,?). Similar to case (S-UP,?).
Case (?,S-ALIAS-RIGHT). Symmetric to case (S-ALIAS-LEFT,?).
Case (S-ALIAS-RIGHT,S-EXTENDS). Impossible because the intermediate type S should be both a virtual type and a class type.
Case (S-DOWN,S-EXTENDS). Similar to case (S-ALIAS-RIGHT,S-EXTENDS).
Case (S-ALIAS-RIGHT,S-UP). The hypotheses are:

$$\boxed{\begin{array}{c} S \equiv p.L \\ \vdash_{\texttt{struct}} p : C \quad (\texttt{type}\ L = T') \in C \quad \vdash_{\texttt{struct}} T <: T'[p] \\ \vdash_{\texttt{struct}} p : C' \quad (\texttt{type}\ L >: T_{opt} <: U') \in C' \quad \vdash_{\texttt{struct}} U'[p] <: U \end{array}}$$

1. By well-formedness of declarations, $(\texttt{type}\ L = T') \in C$ and $(\texttt{type}\ L >: T_{opt} <: U')$ implies $\texttt{this} : C \vdash_{\texttt{struct}} T' <: U'$.
2. By Lemma A.23, $\vdash_{\texttt{struct}} p : C$ implies $\vdash p : C$.
3. By Lemma A.7, $\vdash p : C$ implies p is a value v.
4. By substitution lemma A.26, $\vdash_{\texttt{struct}} v : C$ and $\texttt{this} : C \vdash_{\texttt{struct}} T' <: U'$ implies $\vdash_{\texttt{struct}} T'[v] <: U'[v]$.
5. By Lemma A.13 (Property 2), $(\texttt{type}\ L = T')$ implies $T'[p] \prec_T p.L$.
6. By Lemma A.13 (Property 1), $(\texttt{type}\ L >: T_{opt} <: U')$ implies $U'[p] \prec_T p.L$.
7. $T'[p], U'[p] \prec_T p.L$ implies $\{T, T'[p], U'[p]\} \prec_{T\,mul} \{T, p.L, U\}$.
8. By (IH), $\vdash_{\texttt{struct}} T <: T'[p]$ and $\vdash_{\texttt{struct}} T'[p] <: U'[p]$ implies $\vdash_{\texttt{struct}} T <: U'[p]$.
9. $U'[p] \prec_T p.L$ implies $\{T, U'[p], U\} \prec_{T\,mul} \{T, p.L, U\}$.
10. By (IH), $\vdash_{\texttt{struct}} T <: U'[p]$ and $\vdash_{\texttt{struct}} U'[p] <: U$ implies $\vdash_{\texttt{struct}} T <: U$.

Case (S-Down,S-Alias-Left). Symmetric to case (S-Alias-Right,S-Up).
Case (S-Alias-Right,S-Alias-Left). The hypotheses are:

$$\begin{array}{ccc} & S \equiv p.L & \\ \vdash_{\texttt{struct}} p : C & (\texttt{type}\ L = T') \in C & \vdash_{\texttt{struct}} T <: T'[p] \\ \vdash_{\texttt{struct}} p : C' & (\texttt{type}\ L = U') \in C' & \vdash_{\texttt{struct}} U'[p] <: U \end{array}$$

1. By Lemma A.23, $\vdash_{\texttt{struct}} p : C$ implies $\vdash\ p : C$.
2. By Lemma A.7, $\vdash\ p : C$ implies p is a value $\texttt{new}\ A(\overline{f} = \overline{v})$.
3. By inversion, $\vdash_{\texttt{struct}} \texttt{new}\ A(\overline{f} = \overline{v}) : C$ implies $\vdash_{\texttt{struct}} A <: C$.
4. By Lemma A.21, $\vdash_{\texttt{struct}} A <: C$ implies $A \triangleleft C$.
5. Similarly, we deduce that $A \triangleleft C'$.
6. By Lemma A.20, $A \triangleleft C$ and $A \triangleleft C'$ implies $C \triangleleft C'$ or $C' \triangleleft C$.
7. It follows by uniqueness of type valuations that $C \equiv C'$ and $T' \equiv U'$, which implies $T'[p] \equiv U'[p]$.
8. By Lemma A.13 (Property 2), $(\texttt{type}\ L = T')$ implies $T'[p] \prec_T p.L$.
9. $T'[p] \prec_T p.L$ implies $\{T, T'[p], U\} \prec_{T\,mul} \{T, p.L, U\}$.
10. By (IH), $\vdash_{\texttt{struct}} T <: T'[p]$ and $\vdash_{\texttt{struct}} T'[p] <: U$ implies $\vdash_{\texttt{struct}} T <: U$.

Case (S-Down,S-Up). The hypotheses are:

$$\begin{array}{ccc} & S \equiv p.L \quad \vdash_{\texttt{struct}} p : C & \\ (\texttt{type}\ L >: T' <: U') \in C & \vdash_{\texttt{struct}} T <: T'[p] & \vdash_{\texttt{struct}} U'[p] <: U \end{array}$$

1. By Lemma A.23, $\vdash_{\texttt{struct}} p : C$ implies $\vdash\ p : C$.
2. By Lemma A.7, $\vdash\ p : C$ implies p is a value v.
3. By inversion, $\vdash_{\texttt{struct}} v : C$ implies

   $$\begin{array}{l} v \equiv \texttt{new}\ A(\overline{f} = \overline{v}) \quad \overline{f} \text{ disjoint} \quad \vdash_{\texttt{struct}} \overline{v} : \overline{U} \quad \overline{U} \text{ closed} \\ \forall\, i, U.\ (\texttt{val}\ f_i : U) \text{ implies } \texttt{this} : B \vdash_{\texttt{struct}} U_i <: U \\ \text{isComplete}(A, \overline{f}) \quad \vdash_{\texttt{struct}} A <: C \end{array}$$

4. By Lemma A.21, $\vdash_{\texttt{struct}} A <: C$ implies $A \triangleleft C$.
5. By completeness of v, $(\texttt{type}\ L >: T' <: U') \in C$ and $A \triangleleft C$ implies there exists C' and S' such that $A \triangleleft C'$ and $(\texttt{type}\ L = S') \in C'$.
6. By well-formedness of type valuations, $(\texttt{type}\ L = S') \in C'$ implies $\texttt{this} : C' \vdash_{\texttt{struct}} S' <: U'$ and $\texttt{this} : C' \vdash_{\texttt{struct}} T' <: S'$.
7. By Lemma A.22, $A \triangleleft C'$ implies $\vdash_{\texttt{struct}} A <: C'$.
8. By rule (T-New), $\vdash_{\texttt{struct}} A <: C'$ implies $\vdash_{\texttt{struct}} v : C'$.
9. By substitution lemma A.26, $\vdash_{\texttt{struct}} v : C'$ and $\texttt{this} : C' \vdash_{\texttt{struct}} S' <: U'$ implies $\vdash_{\texttt{struct}} S'[v] <: U'[v]$.

10. By substitution lemma A.26, $\vdash_{\texttt{struct}} v : C'$ and $\texttt{this} : C' \vdash_{\texttt{struct}} T' <: S'$ implies $\vdash_{\texttt{struct}} T'[v] <: S'[v]$.

11. By Lemma A.13 (Property 2), ($\texttt{type}\ L = S'$) implies $S'[p] \prec_T p.L$.

12. By Lemma A.13 (Property 1), the declaration ($\texttt{type}\ L >: T' <: U'$) implies $T'[p], U'[p] \prec_T p.L$.

13. $T'[p], S'[p] \prec_T p.L$ implies $\{ T, T'[p], S'[p] \} \prec_{Tmul} \{ T, p.L, U \}$.

14. By (IH), $\vdash_{\texttt{struct}} T <: T'[p]$ and $\vdash_{\texttt{struct}} T'[p] <: S'[p]$ implies $\vdash_{\texttt{struct}} T <: S'[p]$.

15. $S'[p], U'[p] \prec_T p.L$ implies $\{ T, S'[p], U'[p] \} \prec_{Tmul} \{ T, p.L, U \}$.

16. By (IH), $\vdash_{\texttt{struct}} T <: S'[p]$ and $\vdash_{\texttt{struct}} S'[p] <: U'[p]$ implies $\vdash_{\texttt{struct}} T <: U'[p]$.

17. $U'[p] \prec_T p.L$ implies $\{ T, U'[p], U \} \prec_{Tmul} \{ T, p.L, U \}$.

18. By (IH), $\vdash_{\texttt{struct}} T <: U'[p]$ and $\vdash_{\texttt{struct}} U'[p] <: U$ implies $\vdash_{\texttt{struct}} T <: U$.

Case (S-TRANS-S,**?**). The hypotheses are:

$$\vdash_{\texttt{struct}} T <: S' \quad \vdash_{\texttt{struct}} S' <: S \quad S' \prec_T T, S$$

1. $S' \prec_T T$ implies $\{ S', S, U \} \prec_{Tmul} \{ T, S, U \}$.

2. By (IH), $\vdash_{\texttt{struct}} S' <: S$ and $\vdash_{\texttt{struct}} S <: U$ implies $\vdash_{\texttt{struct}} S' <: U$.

3. $S' \prec_T S$ implies $\{ T, S', U \} \prec_{Tmul} \{ T, S, U \}$.

4. By (IH), $\vdash_{\texttt{struct}} T <: S'$ and $\vdash_{\texttt{struct}} S' <: U$ implies $\vdash_{\texttt{struct}} T <: U$.

Case (**?**,S-TRANS-S). Symmetric to case (S-TRANS-S,?).

Qed.

Lemma A.28 (Structuring derivations)

1. $\vdash T <: U$ *implies* $\vdash_{\texttt{struct}} T <: U$.

2. $\vdash v : T$ *implies* $\vdash_{\texttt{struct}} v : T$.

Proof: By mutual induction on $\vdash T <: U$ and $\vdash v : T$. For all rules but (S-TRANS) the result follows from a direct application of the current induction hypothesis. The case of rule (S-TRANS) is resolved using the admissibility of transitivity for structured subtyping (Lemma A.27).

Qed.

A.5 Progress

Lemma A.29 (Progress)
$\vdash t : T$ *and* t *not a value implies there exists* u *s.t.* $t \rightarrow u$.

Proof: By induction on $\vdash t : T$.

Case (T-This). The hypotheses contain $t \equiv \texttt{this}$

1. By Lemma A.6, it is impossible that $\vdash \texttt{this} : T$, which contradicts one of the hypotheses.

Case (T-Param). The hypotheses contain $t \equiv x$

1. By Lemma A.6, it is impossible that $\vdash x : T$, which contradicts one of the hypotheses.

Case (T-Var). The hypotheses contain $t \equiv n$

1. By Lemma A.6, it is impossible that $\vdash n : T$, which contradicts one of the hypotheses.

Case (T-New). The hypotheses contain $t \equiv \texttt{new}\ C(\overline{f} = \overline{t}) \quad \vdash \overline{t} : \overline{T}$

1. We reason by cases on all $\overline{t}$ being values.

   Case (Yes). The hypothesis is: $\overline{t} \equiv \overline{v}$.

   (a) After replacement $t \equiv \texttt{new}\ C(\overline{f} = \overline{v})$ is a value, which contradicts one of the hypotheses.

   Case (No). The hypotheses are: $\overline{t} \equiv \overline{v}, t', \overline{t}'$ and t' is not a value

   (a) $\vdash \overline{t} : \overline{T}$ implies there exists i s.t. $\vdash t' : T_i$.

   (b) By (IH), there exists u' s.t. $t' \rightarrow u'$.

   (c) By rule (R-Field), $\texttt{new}\ C(\overline{f} = \overline{v}, t', \overline{t}') \rightarrow \texttt{new}\ C(\overline{f} = \overline{v}, u', \overline{t}')$.

Case (T-Select). The hypotheses are

$t \equiv p.f \quad \vdash p : C \quad (\texttt{val}\ f : S) \in C \quad \vdash S[p] <: T$

1. We reason by cases on p being a value.

   Case (No). The hypothesis is: p is not a value

   (a) By (IH), there exists p' s.t. $p \rightarrow p'$.

   (b) By rule (R-Prefix), $p.f \rightarrow p'.f$.

   Case (Yes). The hypothesis is: $p \equiv \texttt{new}\ B(\overline{f} = \overline{v})$

(a) By inversion, $\vdash$ `new` $B(\overline{f} = \overline{v}) : C$ implies

$$\overline{f} \text{ disjoint} \quad \vdash \overline{v} : \overline{U} \quad \overline{U} \text{ closed}$$
$$\forall\, i, S.\ (\texttt{val}\ f_i : S) \text{ implies } \texttt{this} : B \vdash_{\texttt{struct}} U_i <: S$$
$$\text{isComplete}(B, \overline{f}) \quad \vdash B <: C$$

(b) By Lemma A.28, $\vdash B <: C$ implies $\vdash_{\texttt{struct}} B <: C$.

(c) By Lemma A.21, $\vdash_{\texttt{struct}} B <: C$ implies $B \triangleleft C$.

(d) isComplete$(B, \overline{f})$, $B \triangleleft C$ and $(\texttt{val}\ f : S) \in C$ implies there exists i s.t. $f \equiv f_i$.

(e) By rule (R-Select), $p.f_i \rightarrow v_i$.

Case (T-Call). The hypotheses are:

$$t \equiv p.m(\overline{x} = \overline{t}) \quad \vdash p : C \quad (\texttt{def}\ m(\overline{x} : \overline{S}) : S) \in C$$
$$\vdash \overline{t} : \overline{S}[p] \quad \vdash S[p] <: T$$

1. We reason by cases on p and $\overline{t}$ being values.

Case (1). The hypothesis is: p is not a value

(a) By (IH), there exists p' s.t. $p \rightarrow p'$.

(b) By rule (R-Receive), $p.m(\overline{x} = \overline{t}) \rightarrow p'.m(\overline{x} = \overline{t})$.

Case (2). The hypotheses are: $p \equiv v \quad \overline{t} \equiv \overline{v}, t', \overline{t}' \quad t'$ is not a value

(a) By (IH),there exists u' s.t. $t' \rightarrow u'$.

(b) By rule (R-Arg), $p.m(\overline{x} = \overline{v}, t', \overline{t}') \rightarrow p.m(\overline{x} = \overline{v}, u', \overline{t}')$.

Case (3). The hypotheses are: $p \equiv \texttt{new}\ B(\overline{f} = \overline{v}) \quad \overline{t} \equiv \overline{w}$

(a) By inversion, $\vdash$ `new` $B(\overline{f} = \overline{v}) : C$ implies

$$\overline{f} \text{ disjoint} \quad \vdash \overline{v} : \overline{U} \quad \overline{U} \text{ closed}$$
$$\forall\, i, S.\ (\texttt{val}\ f_i : S) \text{ implies } \texttt{this} : B \vdash_{\texttt{struct}} U_i <: S$$
$$\text{isComplete}(B, \overline{f}) \quad \vdash B <: C$$

(b) By admissibility of transitivity (Lemma A.27), $\vdash B <: C$ implies $\vdash_{\texttt{struct}} B <: C$.

(c) By Lemma A.21, $\vdash_{\texttt{struct}} B <: C$ implies $B \triangleleft C$.

(d) isComplete$(B, \overline{f})$, $B \triangleleft C$ and $(\texttt{def}\ m(\overline{x} : \overline{S}) : S) \in C$ implies there exist A, t' s.t. $B \triangleleft A$ and $(\texttt{def}\ m = t') \in A$.

(e) By rule (R-Call), $p.m(\overline{x} = \overline{w}) \rightarrow t'[p][\overline{x} \backslash \overline{w}]$.

Case (T-Let). The hypotheses are: $t \equiv \texttt{let}{:}T = t'\ \texttt{in}\ t''$

1. We reason by cases on t' being a value.

Case (No). The hypothesis is: t' is not a value

(a) By (IH), there exists u' s.t. $t' \rightarrow u'$.

(b) By rule (R-LOCAL), $\texttt{let:}T = t' \texttt{ in } t'' \rightarrow \texttt{let:}T = u' \texttt{ in } t''$.

Case (YES). The hypothesis is: $\boxed{t' \equiv v}$.

(a) By rule (R-LET), $\texttt{let:}T = v \texttt{ in } t'' \rightarrow t''[0 := v]$.

Qed.

A.6 De Bruijn's Indices

Lemma A.30 (Lifting a closed term or type)

1. *$fv(t) \equiv \emptyset$ implies $t\uparrow^n \equiv t$.*
2. *$fv(T) \equiv \emptyset$ implies $T\uparrow^n \equiv T$.*

Proof: By mutual induction on t and T.

Qed.

Lemma A.31 (Dropping a closed term or type)

1. *$fv(t) \equiv \emptyset$ implies $t\downarrow \equiv t$.*
2. *$fv(T) \equiv \emptyset$ implies $T\downarrow \equiv T$.*

Proof: Same proof as for Lemma A.30.

Qed.

Lemma A.32 (Values are invariant by lifting)

$$v\uparrow^n_k \equiv v$$

Proof: By induction on v.

Qed.

Lemma A.33 (Permuting lifting and self substitution)

1. *$t[v]\uparrow^n \equiv t\uparrow^n[v]$.*
2. *$T[v]\uparrow^n \equiv T\uparrow^n[v]$.*

Proof: We prove $t[v]\uparrow^n_k \equiv t\uparrow^n_k[v]$ and $T[v]\uparrow^n_k \equiv T\uparrow^n_k[v]$ by mutual induction on t and T.
Case (THIS). The hypothesis is $\boxed{t \equiv \texttt{this}}$

1. $t[v]\uparrow^n_k \equiv \texttt{this}[v]\uparrow^n_k \equiv v\uparrow^n_k$.
2. By Lemma A.32, $v\uparrow^n_k \equiv v$.
3. $t\uparrow^n_k[v] \equiv \texttt{this}\uparrow^n_k[v] \equiv \texttt{this}[v] \equiv v$.

Case (VAR). The hypothesis is $\boxed{t \equiv i}$

1. $t[v]\uparrow^n_k \equiv i[v]\uparrow^n_k \equiv i\uparrow^n_k$.
2. $t\uparrow^n_k[v] \equiv i\uparrow^n_k[v]$.
3. We reason by cases on $i < k$. If YES, $i\uparrow^n_k \equiv i$ and $i[v] \equiv i$. If NO, $i\uparrow^n_k \equiv i + n$ and $(i + n)[v] \equiv i + n$.

Case (OTHERS). All other cases are straightforward applications of the induction hypothesis.

Qed.

Lemma A.34 (Permuting lifting and parameter substitution)

1. $t[\overline{x}\backslash\overline{v}]\uparrow^n \equiv t\uparrow^n[\overline{x}\backslash\overline{v}]$.

2. $T[\overline{x}\backslash\overline{v}]\uparrow^n \equiv T\uparrow^n[\overline{x}\backslash\overline{v}]$.

Proof: We prove $t[\overline{x}\backslash\overline{v}]\uparrow^n_k \equiv t\uparrow^n_k[\overline{x}\backslash\overline{v}]$ and $T[\overline{x}\backslash\overline{v}]\uparrow^n_k \equiv T\uparrow^n_k[\overline{x}\backslash\overline{v}]$ by mutual induction on t and T.
Case (Param). The hypothesis is $\boxed{t \equiv x}$

1. $t[\overline{x}\backslash\overline{v}]\uparrow^n_k \equiv x[\overline{x}\backslash\overline{v}]\uparrow^n_k$.
2. $t\uparrow^n_k[\overline{x}\backslash\overline{v}] \equiv x\uparrow^n_k[\overline{x}\backslash\overline{v}] \equiv x[\overline{x}\backslash\overline{v}]$.
3. We reason by cases on $x \in \overline{x}$.

   Case (Yes).

   (a) There exists i s.t. $x[\overline{x}\backslash\overline{v}] \equiv v_i$.
   (b) $x[\overline{x}\backslash\overline{v}]\uparrow^n_k \equiv v_i\uparrow^n_k$.
   (c) By Lemma A.32, $v_i\uparrow^n_k \equiv v_i$.

   Case (No).

   (a) $x[\overline{x}\backslash\overline{v}] \equiv x$.
   (b) $x[\overline{x}\backslash\overline{v}]\uparrow^n_k \equiv x\uparrow^n_k \equiv x$.

Case (Var). The hypothesis is $\boxed{t \equiv i}$

1. $t[\overline{x}\backslash\overline{v}]\uparrow^n_k \equiv i[\overline{x}\backslash\overline{v}]\uparrow^n_k \equiv i\uparrow^n_k$.
2. $t\uparrow^n_k[\overline{x}\backslash\overline{v}] \equiv i\uparrow^n_k[\overline{x}\backslash\overline{v}]$.
3. We reason by cases on $i < k$. If Yes, $i\uparrow^n_k \equiv i$ and $i[\overline{x}\backslash\overline{v}] \equiv i$. If No, $i\uparrow^n_k \equiv i+n$ and $(i+n)[\overline{x}\backslash\overline{v}] \equiv i+n$.

Case (Others). All other cases are straightforward applications of the induction hypothesis.

Qed.

Lemma A.35 (Permuting lifting and variable substitution)

1. $t[k := v]\uparrow^n \equiv t\uparrow^n[k+n := v]$.

2. $T[k := v]\uparrow^n \equiv T\uparrow^n[k+n := v]$.

Proof: We prove

1. $\forall k' \leq k.\ t[k := v]\uparrow^n_{k'} \equiv t\uparrow^n_{k'}[k+n := v]$.
2. $\forall k' \leq k.\ T[k := v]\uparrow^n_{k'} \equiv T\uparrow^n_{k'}[k+n := v]$.

by mutual induction on t and T, and then we take $k' = 0$.
Case (Var). The hypothesis is $\boxed{t \equiv i}$

1. We reason by cases on i and k.
   Case $(i < k)$.
   (a) $i[k := v] \equiv i$.
   (b) We reason by cases on i and k'.
   Case $(i < k')$.
      - $i{\uparrow}^n_{k'} \equiv i$.
      - $i < k$ implies $i < k + n$.
      - $i < k + n$ implies $i[k + n := v] \equiv i$.

      Case $(i \equiv k')$.
      - $i{\uparrow}^n_{k'} \equiv i + n$.
      - $i < k$ implies $i + n < k + n$.
      - $i + n < k + n$ implies $(i + n)[k + n := v] \equiv i + n$.

      Case $(i > k')$.
      - $i{\uparrow}^n_{k'} \equiv i + n$.
      - $i < k$ implies $i + n < k + n$.
      - $i + n < k + n$ implies $(i + n)[k + n := v] \equiv i + n$.

   Case $(i \equiv k)$.
   (a) $i[k := v] \equiv v{\uparrow}^k$.
   (b) By Lemma A.32, $v{\uparrow}^k \equiv v$.
   (c) By Lemma A.32, $v{\uparrow}^n_{k'} \equiv v$.
   (d) We reason by cases on i and k'.
   Case $(i < k')$.
      - $i \equiv k$ and $i < k'$ implies $k < k'$, which contradicts $k' \leq k$.

      Case $(i \equiv k')$.
      - $i \equiv k'$ implies $i \geq k'$.
      - $i \geq k'$ implies $i{\uparrow}^n_{k'} \equiv i + n$.
      - $i \equiv k$ implies $(i+n)[k+n := v] \equiv (k+n)[k+n := v] \equiv v{\uparrow}^{k+n} \equiv v$.

      Case $(i > k')$.
      - $i > k'$ implies $i \geq k'$.
      - $i \geq k'$ implies $i{\uparrow}^n_{k'} \equiv i + n$.
      - $i \equiv k$ implies $(i+n)[k+n := v] \equiv (k+n)[k+n := v] \equiv v{\uparrow}^{k+n} \equiv v$.

   Case $(i > k)$.
   (a) $i[k := v] \equiv i - 1$.
   (b) We reason by cases on i and k'.
   Case $(i < k')$.
      - $i > k$ and $i < k'$ implies $k < k'$, which contradicts $k' \leq k$.

      Case $(i \equiv k')$.

- $i > k$ and $i \equiv k'$ implies $k' > k$, which contradicts $k' \leq k$.

Case $(i > k')$.

- $i > k'$ implies $i - 1 \geq k'$, implies $(i-1)\uparrow^n_{k'} \equiv i - 1 + n$
- $i > k'$ implies $i \geq k'$, implies $i\uparrow^n_{k'} \equiv i + n$.
- $i > k$ implies $i+n > k+n$, implies $(i+n)[k+n := v] \equiv i+n-1$.

Case (LET). The hypothesis is $\boxed{t \equiv \texttt{let}\!:\! T = t' \texttt{ in } t''}$

1. On one hand

$$\begin{array}{rcl} t[k := v]\uparrow^n_{k'} & \equiv & (\texttt{let}\!:\! T = t' \texttt{ in } t'')[k := v]\uparrow^n_{k'} \\ & \equiv & (\texttt{let}\!:\! T[k := v] = t'[k := v] \texttt{ in } t''[k+1 := v])\uparrow^n_{k'} \\ & \equiv & \texttt{let}\!:\! T[k := v]\uparrow^n_{k'} = t'[k := v]\uparrow^n_{k'} \texttt{ in } t''[k+1 := v]\uparrow^n_{k'+1} \end{array}$$

2. On the other hand

$$\begin{array}{rcl} t\uparrow^n_{k'}[k+n := v] & \equiv & (\texttt{let}\!:\! T = t' \texttt{ in } t'')\uparrow^n_{k'}[k+n := v] \\ & \equiv & (\texttt{let}\!:\! T\uparrow^n_{k'} = t'\uparrow^n_{k'} \texttt{ in } t''\uparrow^n_{k'+1})[k+n := v] \\ & \equiv & \texttt{let}\!:\! T\uparrow^n_{k'}[k+n := v] = t'\uparrow^n_{k'}[k+n := v] \texttt{ in } t''\uparrow^n_{k'+1}[k+n+1 := v] \end{array}$$

3. By (IH), $T[k := v]\uparrow^n_{k'} \equiv T\uparrow^n_{k'}[k+n := v]$.
4. By (IH), $t'[k := v]\uparrow^n_{k'} \equiv t'\uparrow^n_{k'}[k+n := v]$.
5. $k' \leq k$ implies $k'+1 \leq k+1$.
6. By (IH), $t''[k+1 := v]\uparrow^n_{k'+1} \equiv t\uparrow^n_{k'+1}[k+1+n := v]$.

Case (OTHERS). All other cases are straightforward applications of the induction hypothesis.

Qed.

Lemma A.36 (Permuting dropping and substitutions)

1. $t[v]\downarrow \equiv t\downarrow[v]$ *and* $T[v]\downarrow \equiv T\downarrow[v]$.
2. $t[\overline{x}\backslash\overline{v}]\downarrow \equiv t\downarrow[\overline{x}\backslash\overline{v}]$ *and* $T[\overline{x}\backslash\overline{v}]\downarrow \equiv T\downarrow[\overline{x}\backslash\overline{v}]$.
3. $0 \notin fv(t)$ *implies* $t[k+1 := v]\downarrow \equiv t\downarrow[k := v]$, *and* $0 \notin fv(T)$ *implies* $T[k+1 := v]\downarrow \equiv T\downarrow[k := v]$.

Proof: Proofs are similar to those of Lemmas A.33, A.34 and A.35.

Qed.

Lemma A.37 (Facts about lifting)

1. $fv(t) \equiv \emptyset$ *implies* $t[k := u] \equiv t$.

Proof: *!!!*

Qed.

A.7 Substitution Lemmas

Lemma A.38 (Weakening)

1. $\overline{S} \vdash t : T$ *implies* $\texttt{root};\ \overline{x} : \overline{T};\ \overline{U}, \overline{S} \vdash t : T$
2. $\overline{S} \vdash T <: U$ *implies* $\texttt{root};\ \overline{x} : \overline{T};\ \overline{U}, \overline{S} \vdash T <: U$

Proof: By mutual induction on typing and subtyping derivations.
Case (T-This). The hypotheses are $t \equiv \texttt{this} \quad \Gamma \equiv \overline{S} \quad \Gamma \vdash C <: T$

1. Case impossible because `this` is not typable in the root context.

Case (T-Param). The hypotheses contain $t \equiv x$

1. Case impossible because x is not typable in a context without bindings for parameters.

Case (T-Var). The hypotheses are

$$t \equiv n \quad \Gamma \equiv \overline{S} \quad N \equiv |\overline{S}| \quad n < N \quad \Gamma \vdash S_{(N-1-n)}\uparrow^{n+1} <: T$$

1. By (IH), $\texttt{root};\ \overline{x} : \overline{T};\ \overline{U}, \overline{S} \vdash S_{(N-1-n)}\uparrow^{n+1} <: T$.
2. Let $\overline{V} \equiv \overline{U}, \overline{S}$ and $N' \equiv |\overline{V}|$.
3. $N \leq N'$ and $n < N$ implies $n < N'$.
4. $n < N$ implies $V_{(N'-1-n)} \equiv S_{(N-1-n)}$.
5. By rule (T-Var), $\texttt{root};\ \overline{x} : \overline{T};\ \overline{U}, \overline{S} \vdash n : T$.

Case (T-Select, T-Call, T-New). By applying (IH) on premises followed by the same typing rule.
Case (T-Let). The hypotheses are

$$t \equiv \texttt{let}{:}T_1 = t_1\ \texttt{in}\ t_2 \quad \Gamma \equiv \overline{S}$$
$$\Gamma \vdash t_1 : T_1 \quad \overline{S}, T_1 \vdash t_2 : T_2 \quad 0 \notin \text{fv}(T_2) \quad \Gamma \vdash T_2\!\downarrow <: T$$

1. By (IH), $\texttt{root};\ \overline{x} : \overline{T};\ \overline{U}, \overline{S} \vdash t_1 : T_1$.
2. By (IH), $\texttt{root};\ \overline{x} : \overline{T};\ \overline{U}, \overline{S}, T_1 \vdash t_2 : T_2$.
3. By (IH), $\texttt{root};\ \overline{x} : \overline{T};\ \overline{U}, \overline{S} \vdash T_2\!\downarrow <: T$.
4. By rule (T-Let), $\texttt{root};\ \overline{x} : \overline{T};\ \overline{U}, \overline{S} \vdash \texttt{let}{:}T_1 = t_1\ \texttt{in}\ t_2 : T$.

Case (S-Trans, S-Class, S-Extends, S-Virtual, S-Up, S-Down, S-Alias-Left, S-Alias-Right). By applying (IH) on case hypotheses followed by the same subtyping rule.

Qed.

For stating and proving the substitution lemmas we need the concept of *well-formed typing context*. Well-formedness of contexts is defined in Figure A.1.

Lemma A.39 (Substitution lemmas)

$$\text{(WF-CTX)}\ \frac{\text{fp}(\overline{T}) \equiv \emptyset \qquad \forall\, i \in [0, |\overline{U}|-1],\ \forall\, k \in \text{fv}(U_i),\ k < i}{\text{wellFormed}(\Gamma_o;\ \overline{x} : \overline{T};\ \overline{U})}$$

Figure A.1: Context Well-formedness (wellFormed(Γ))

1. *Suppose* $\Gamma \equiv C;\ \overline{x} : \overline{T};\ \overline{U}$ *and* $\vdash\ v : C$. *Then,*
   (a) $\Gamma\ \vdash\ t : T$ *implies* $\texttt{root};\ \overline{x} : \overline{T}[v];\ \overline{U}[v]\ \vdash\ t[v] : T[v]$.
   (b) $\Gamma\ \vdash\ T <: U$ *implies* $\texttt{root};\ \overline{x} : \overline{T}[v];\ \overline{U}[v]\ \vdash\ T[v] <: U[v]$.
2. *Suppose* $\Gamma \equiv \texttt{root};\ \overline{x} : \overline{T};\ \overline{U}$, *wellFormed*$(\Gamma)$ *and* $\vdash\ \overline{v} : \overline{T}$. *Then,*
   (a) $\Gamma\ \vdash\ t : T$ *implies* $\overline{U}[\overline{x}\backslash\overline{v}]\ \vdash\ t[\overline{x}\backslash\overline{v}] : T[\overline{x}\backslash\overline{v}]$.
   (b) $\Gamma\ \vdash\ T <: U$ *implies* $\overline{U}[\overline{x}\backslash\overline{v}]\ \vdash\ T[\overline{x}\backslash\overline{v}] <: U[\overline{x}\backslash\overline{v}]$.
3. *Suppose* $\Gamma \equiv \texttt{root};\ \epsilon;\ U, \overline{U}$, *wellFormed*$(\Gamma)$, $N \equiv |\overline{U}|$ *and* $\vdash\ v : U$. *Then,*
   (a) $\Gamma\ \vdash\ t : T$ *implies* $(U_i[i := v])_{i \in [0,N-1]}\ \vdash\ t[N := v] : T[N := v]$.
   (b) $\Gamma\ \vdash\ S_1 <: S_2$ *implies* $(U_i[i := v])_{i \in [0,N-1]}\ \vdash\ S_1[N := v] <: S_2[N := v]$.

Proof: Each property by mutual induction on typing and subtyping derivations. We start with Property 1.
Case (T-THIS). The hypotheses are

$t \equiv \texttt{this} \quad \Gamma \equiv C;\ \overline{x} : \overline{T};\ \overline{U} \quad \Gamma\ \vdash\ C <: T$

1. $t[v] \equiv \texttt{this}[v] \equiv v$.
2. By (IH), $\Gamma\ \vdash\ C <: T$ implies $\texttt{root};\ \overline{x} : \overline{T}[v];\ \overline{U}[v]\ \vdash\ C <: T[v]$.
3. By weakening (Lemma A.38)), $\vdash\ v : C$ implies $\texttt{root};\ \overline{x} : \overline{T}[v];\ \overline{U}[v]\ \vdash\ v : C$.
4. By subsumption (Lemma A.1), the two previous points imply $\texttt{root};\ \overline{x} : \overline{T}[v];\ \overline{U}[v]\ \vdash\ v : T[v]$.

Case (T-VAR). The hypotheses are

$t \equiv n \quad \Gamma \equiv C;\ \overline{x} : \overline{T};\ \overline{U} \quad N \equiv \lvert\overline{U}\rvert \quad \Gamma\ \vdash\ U_{(N-1-n)}\uparrow^{n+1} <: T$

1. $t[v] \equiv n[v] \equiv n$.
2. By (IH), $\texttt{root};\ \overline{x} : \overline{T}[v];\ \overline{U}[v]\ \vdash\ U_{(N-1-n)}\uparrow^{n+1}[v] <: T[v]$.
3. By Lemma A.33, $U_{(N-1-n)}\uparrow^{n+1}[v] \equiv U_{(N-1-n)}[v]\uparrow^{n+1}$.
4. By rule (T-VAR), $\texttt{root};\ \overline{x} : \overline{T}[v];\ \overline{U}[v]\ \vdash\ n : T[v]$.

Case (T-PARAM). The hypotheses are

$t \equiv x_i \quad \Gamma \equiv C;\ \overline{x} : \overline{T};\ \overline{U} \quad \Gamma\ \vdash\ T_i <: T$

1. $t[v] \equiv x_i[v] \equiv x_i$.
2. By (IH), $\texttt{root};\ \overline{x} : \overline{T}[v];\ \overline{U}[v] \vdash T_i[v] <: T[v]$.
3. By rule (T-PARAM), $\texttt{root};\ \overline{x} : \overline{T}[v];\ \overline{U}[v] \vdash x_i : T[v]$.

Case (T-SELECT). The hypotheses are

$$t \equiv p.f \quad \Gamma \equiv C;\ \overline{x} : \overline{T};\ \overline{U} \quad \Gamma \vdash p : A \quad (\texttt{val}\ f : S) \in A \quad \Gamma \vdash S[p] <: T$$

1. $t[v] \equiv p.f[v] \equiv p[v].f$.
2. By (IH), $\texttt{root};\ \overline{x} : \overline{T}[v];\ \overline{U}[v] \vdash p[v] : A$.
3. By (IH), $\texttt{root};\ \overline{x} : \overline{T}[v];\ \overline{U}[v] \vdash S[p][v] <: T[v]$.
4. By Lemma A.8, $S[p][v] \equiv S[p[v]]$.
5. By rule (T-SELECT), $\texttt{root};\ \overline{x} : \overline{T}[v];\ \overline{U}[v] \vdash p[v].f : T[v]$.

Case (T-CALL). Similar to case (T-SELECT).

Case (T-NEW). The hypotheses are

$$t \equiv \texttt{new}\ C(\overline{f} = \overline{t}) \quad \Gamma \equiv C;\ \overline{x} : \overline{T};\ \overline{U} \quad \Gamma \vdash \overline{t} : \overline{U} \quad \overline{U}\ \text{closed} \quad \Gamma \vdash C <: T \quad \cdots$$

1. $t[v] \equiv \texttt{new}\ C(\overline{f} = \overline{t})[v] \equiv \texttt{new}\ C(\overline{f} = \overline{t}[v])$.
2. By (IH), $\texttt{root};\ \overline{x} : \overline{T}[v];\ \overline{U}[v] \vdash \overline{t}[v] : \overline{U}[v]$.
3. $\overline{U}$ closed implies $\overline{U}[v] \equiv \overline{U}$.
4. By (IH), $\texttt{root};\ \overline{x} : \overline{T}[v];\ \overline{U}[v] \vdash C <: T[v]$.
5. By rule (T-SELECT), $\texttt{root};\ \overline{x} : \overline{T}[v];\ \overline{U}[v] \vdash \texttt{new}\ C(\overline{f} = \overline{t}[v]) : T[v]$.

Case (T-LET). The hypotheses are

$$\begin{array}{c} t \equiv \texttt{let}{:}\, S_1 = t_1\ \texttt{in}\ t_2 \quad \Gamma \equiv C;\ \overline{x} : \overline{T};\ \overline{U} \\ \Gamma \vdash t_1 : S_1 \quad C;\ \overline{x} : \overline{T};\ \overline{U}, S_1 \vdash t_2 : S_2 \quad 0 \notin \text{fv}(S_2) \quad \Gamma \vdash S_2{\downarrow} <: T \end{array}$$

1. $t[v] \equiv (\texttt{let}{:}\, S_1 = t_1\ \texttt{in}\ t_2)[v] \equiv \texttt{let}{:}\, S_1[v] = t_1[v]\ \texttt{in}\ t_2[v]$.
2. By (IH), $\texttt{root};\ \overline{x} : \overline{T}[v];\ \overline{U}[v] \vdash t_1[v] : S_1[v]$.
3. By (IH), $\texttt{root};\ \overline{x} : \overline{T}[v];\ \overline{U}[v], S_1[v] \vdash t_2[v] : S_2[v]$.
4. $0 \notin \text{fv}(S_2)$ and v closed implies $0 \notin \text{fv}(S_2[v])$.
5. By (IH), $\texttt{root};\ \overline{x} : \overline{T}[v];\ \overline{U}[v] \vdash (S_2{\downarrow})[v] <: T[v]$.
6. By Lemma A.36 (Property 1), $(S_2{\downarrow})[v] \equiv S_2[v]{\downarrow}$.
7. By rule (T-LET), $\texttt{root};\ \overline{x} : \overline{T}[v];\ \overline{U}[v] \vdash \texttt{let}{:}\, S_1[v] = t_1[v]\ \texttt{in}\ t_2[v] : T[v]$.

Case (S-VIRTUAL). The hypotheses are

$$T \equiv p.L \quad U \equiv p.L \quad \Gamma \vdash p : A \quad (\texttt{type}\ L >: T_{opt} <: S) \in A$$

1. $T[v] \equiv U[v] \equiv p.L[v] \equiv p[v].L$.

2. By (IH), $\texttt{root};\ \overline{x} : \overline{T}[v];\ \overline{U}[v]\ \vdash\ p[v] : A$.

3. By rule (S-VIRTUAL), $\texttt{root};\ \overline{x} : \overline{T}[v];\ \overline{U}[v]\ \vdash\ p[v].L <: p[v].L$.

Case (S-TRANS, S-CLASS, S-EXTENDS). By applying (IH) on case hypotheses followed by the same subtyping rule.
Case (S-UP, S-DOWN, S-ALIAS-LEFT, S-ALIAS-RIGHT). Similar to case (T-SELECT).

Qed.

Proof: We continue with Property 2 of Lemma A.39.

Case (T-THIS). Impossible because the typing environment should start both with a class context and a root context.
Case (T-VAR). The hypotheses are

$$\boxed{t \equiv n \quad \Gamma \equiv \texttt{root};\ \overline{x} : \overline{T};\ \overline{U} \quad N \equiv |\overline{U}| \quad \Gamma\ \vdash\ U_{(N-1-n)}{\uparrow}^{n+1} <: T}$$

1. $t[\overline{x}\backslash\overline{v}] \equiv n[\overline{x}\backslash\overline{v}] \equiv n$.

2. By (IH), $\overline{U}[\overline{x}\backslash\overline{v}]\ \vdash\ U_{(N-1-n)}{\uparrow}^{n+1}[\overline{x}\backslash\overline{v}] <: T[\overline{x}\backslash\overline{v}]$.

3. By Lemma A.34, $U_{(N-1-n)}{\uparrow}^{n+1}[\overline{x}\backslash\overline{v}] \equiv U_{(N-1-n)}[\overline{x}\backslash\overline{v}]{\uparrow}^{n+1}$.

4. By rule (T-VAR), $\overline{U}[\overline{x}\backslash\overline{v}]\ \vdash\ n : T[\overline{x}\backslash\overline{v}]$.

Case (T-PARAM). The hypotheses are

$$\boxed{t \equiv x_i \quad \Gamma \equiv \texttt{root};\ \overline{x} : \overline{T};\ \overline{U} \quad \Gamma\ \vdash\ T_i <: T}$$

1. $t[\overline{x}\backslash\overline{v}] \equiv x_i[\overline{x}\backslash\overline{v}] \equiv v_i$.

2. By (IH), $\overline{U}[\overline{x}\backslash\overline{v}]\ \vdash\ T_i[\overline{x}\backslash\overline{v}] <: T[\overline{x}\backslash\overline{v}]$.

3. wellFormed(Γ) implies $\mathrm{fp}(\overline{T}) \equiv \emptyset$.

4. $\mathrm{fp}(\overline{T}) \equiv \emptyset$ implies $T_i[\overline{x}\backslash\overline{v}] \equiv T_i$.

5. $\vdash\ \overline{v} : \overline{T}$ implies $\vdash\ v_i : T_i$.

6. By weakening (Lemma A.38), $\vdash\ v_i : T_i$ implies $\overline{U}[\overline{x}\backslash\overline{v}]\ \vdash\ v_i : T_i$.

7. By subsumption (Lemma A.1), $\overline{U}[\overline{x}\backslash\overline{v}]\ \vdash\ v_i : T[\overline{x}\backslash\overline{v}]$.

Case (T-SELECT). The hypotheses are

$$\boxed{t \equiv p.f \quad \Gamma \equiv \texttt{root};\ \overline{x} : \overline{T};\ \overline{U} \quad \Gamma\ \vdash\ p : A \quad (\texttt{val}\ f : S) \in A \quad \Gamma\ \vdash\ S[p] <: T}$$

1. $t[\overline{x}\backslash\overline{v}] \equiv p.f[\overline{x}\backslash\overline{v}] \equiv p[\overline{x}\backslash\overline{v}].f$.

2. By (IH), $\overline{U}[\overline{x}\backslash\overline{v}]\ \vdash\ p[\overline{x}\backslash\overline{v}] : A$.

3. By (IH), $\overline{U}[\overline{x}\backslash\overline{v}]\ \vdash\ S[p][\overline{x}\backslash\overline{v}] <: T[\overline{x}\backslash\overline{v}]$.

4. $(\texttt{val}\ f : S) \in A$ well-formed implies $\texttt{this} : A \vdash S\ \textsc{wf}$.

5. By Lemma A.3 (Property 3), $\texttt{this} : A \vdash S\ \textsc{wf}$ implies $\mathrm{fp}(S) \subset \emptyset$.

6. By Lemma A.9, $\overline{x} \notin \mathrm{fp}(S)$ implies $S[p][\overline{x}\backslash\overline{v}] \equiv S[p[\overline{x}\backslash\overline{v}]]$.

7. By rule (T-Select), $\overline{U}[\overline{x}\backslash\overline{v}] \vdash p[\overline{x}\backslash\overline{v}].f : T[\overline{x}\backslash\overline{v}]$.

Case (T-Let). The hypotheses are

$$t \equiv \texttt{let}\colon S_1 = t_1 \;\texttt{in}\; t_2 \quad \Gamma \equiv \texttt{root};\ \overline{x} : \overline{T};\ \overline{U}$$
$$\Gamma \vdash t_1 : S_1 \quad \texttt{root};\ \overline{x} : \overline{T};\ \overline{U}, S_1 \vdash t_2 : S_2 \quad 0 \notin \mathrm{fv}(S_2) \quad \Gamma \vdash S_2{\downarrow} <: T$$

1. $t[\overline{x}\backslash\overline{v}] \equiv (\texttt{let}\colon S_1 = t_1 \;\texttt{in}\; t_2)[\overline{x}\backslash\overline{v}] \equiv \texttt{let}\colon S_1[\overline{x}\backslash\overline{v}] = t_1[\overline{x}\backslash\overline{v}] \;\texttt{in}\; t_2[\overline{x}\backslash\overline{v}]$.
2. By (IH), $\overline{U}[\overline{x}\backslash\overline{v}] \vdash t_1[\overline{x}\backslash\overline{v}] : S_1[\overline{x}\backslash\overline{v}]$.
3. By (IH), $\overline{U}[\overline{x}\backslash\overline{v}], S_1[\overline{x}\backslash\overline{v}] \vdash t_2[\overline{x}\backslash\overline{v}] : S_2[\overline{x}\backslash\overline{v}]$.
4. $0 \notin \mathrm{fv}(S_2)$ and v closed implies $0 \notin \mathrm{fv}(S_2[\overline{x}\backslash\overline{v}])$.
5. By (IH), $\overline{U}[\overline{x}\backslash\overline{v}] \vdash S_2{\downarrow}[\overline{x}\backslash\overline{v}] <: T[\overline{x}\backslash\overline{v}]$.
6. By Lemma A.36 (Property 2), $S_2{\downarrow}[\overline{x}\backslash\overline{v}] \equiv S_2[\overline{x}\backslash\overline{v}]{\downarrow}$.
7. By rule (T-Let), $\overline{U}[\overline{x}\backslash\overline{v}] \vdash \texttt{let}\colon S_1[\overline{x}\backslash\overline{v}] = t_1[\overline{x}\backslash\overline{v}] \;\texttt{in}\; t_2[\overline{x}\backslash\overline{v}] : T[\overline{x}\backslash\overline{v}]$.

Case (S-Virtual). The hypotheses are

$$T \equiv p.L \quad U \equiv p.L \quad \Gamma \vdash p : A \quad (\texttt{type}\ L >: T_{opt} <: S) \in A$$

1. $T[\overline{x}\backslash\overline{v}] \equiv U[\overline{x}\backslash\overline{v}] \equiv p.L[\overline{x}\backslash\overline{v}] \equiv p[\overline{x}\backslash\overline{v}].L$.
2. By (IH), $\overline{U}[\overline{x}\backslash\overline{v}] \vdash p[\overline{x}\backslash\overline{v}] : A$.
3. By rule (S-Virtual), $\overline{U}[\overline{x}\backslash\overline{v}] \vdash p[\overline{x}\backslash\overline{v}].L <: p[\overline{x}\backslash\overline{v}].L$.

Case (T-New). Similar to case (T-New) in the proof of Property 1 using the fact that if $\overline{U}$ are closed then $\overline{U}[\overline{x}\backslash\overline{v}] \equiv \overline{U}$.
Case (S-Trans, S-Class, S-Extends). By applying (IH) on case hypotheses followed by the same subtyping rule.
Case (S-Up, S-Down, S-Alias-Left, S-Alias-Right). Similar to case (T-Select).

Qed.

Proof: We continue with Property 3 of Lemma A.39.

Case (T-This). The hypotheses are

$$t \equiv \texttt{this} \quad \Gamma \equiv U, \overline{U} \quad \Gamma \vdash C <: T$$

1. $t[N := v] \equiv \texttt{this}[N := v] \equiv \texttt{this}$.
2. By (IH), $(U_i[i := v])_i \vdash C <: T[N := v]$.
3. By rule (T-This), $(U_i[i := v])_i \vdash \texttt{this} : T[N := v]$.

Case (T-Var). The hypotheses are

$$t \equiv n \quad \Gamma \equiv U, \overline{U} \quad \overline{V} = U, \overline{U} \quad n < N + 1 \quad \Gamma \vdash V_{(N-n)}{\uparrow}^{n+1} <: T$$

1. By (IH), $(U_i[i := v])_i \vdash V_{(N-n)}{\uparrow}^{n+1}[N := v] <: T[N := v]$.
2. We reason by cases on n.

   Case ($n \equiv N$).

(a) $t[N := v] \equiv N[N := v] \equiv v{\uparrow}^N \equiv v$.
(b) $V_{(N-n)}{\uparrow}^{n+1}[N := v] \equiv V_0{\uparrow}^{N+1}[N := v] \equiv U{\uparrow}^{N+1}[N := v]$.
(c) wellFormed(Γ) implies $\forall\, k \in \text{fv}(U),\ k < 0$, i.e. $\text{fv}(U) \equiv \emptyset$.
(d) By Lemma A.30, $\text{fv}(U) \equiv \emptyset$ implies $U{\uparrow}^{N+1} \equiv U$.
(e) By Lemma A.37, $\text{fv}(U) \equiv \emptyset$ implies $U[N := v] \equiv U$.
(f) The two previous points imply $U{\uparrow}^{N+1}[N := v] \equiv U$.
(g) By weakening (Lemma A.38), $\vdash\ v : U$ implies $(U_i[i := v])_i \vdash\ v : U$.
(h) By subsumption (Lemma A.1), $(U_i[i := v])_i \vdash\ v : T[N := v]$.

Case $(n < N)$.

(a) $t[N := v] \equiv n[N := v] \equiv n$ (since $n < N$).
(b) $n < N$ implies $V_{(N-n)}{\uparrow}^{n+1}[N := v] \equiv U_{(N-1-n)}{\uparrow}^{n+1}[N := v]$.
(c) By Lemma A.35, $U_{(N-1-n)}{\uparrow}^{n+1}[N := v] \equiv U_{(N-1-n)}[N - 1 - n := v]{\uparrow}^{n+1}$.
(d) By rule (T-VAR), $(U_i[i := v])_i \vdash\ n : T[N := v]$.

Case (T-PARAM). The hypotheses are $t \equiv x$

1. Case impossible because x is not typable in context $U, \overline{U}$.

Case (T-SELECT). The hypotheses are

$t \equiv p.f \quad \Gamma \equiv U, \overline{U} \quad \Gamma \vdash\ p : C \quad (\texttt{val}\ f : S) \in C \quad \Gamma \vdash\ S[p] <: T$

1. $t[N := v] \equiv p.f[N := v] \equiv p[N := v].f$.
2. By (IH), $(U_i[i := v])_i \vdash\ p[N := v] : C$.
3. By (IH), $(U_i[i := v])_i \vdash\ S[p][N := v] <: T[N := v]$.
4. $(\texttt{val}\ f : S) \in C$ well-formed implies $\texttt{this} : C \vdash S$ WF.
5. By Lemma A.4 (Property 3), $\texttt{this} : C \vdash S$ WF implies $\text{fv}(S) \equiv \emptyset$.
6. By Lemma A.10, $\text{fv}(S) \equiv \emptyset$ implies $S[p][N := v] \equiv S[p[N := v]]$.
7. By rule (T-SELECT), $(U_i[i := v])_i \vdash\ p[N := v].f : T[N := v]$.

Case (T-LET). The hypotheses are

$t \equiv \texttt{let:} S_1 = t_1\ \texttt{in}\ t_2 \quad \Gamma \equiv U, \overline{U}$
$\Gamma \vdash\ t_1 : S_1 \quad U, \overline{U}, S_1 \vdash\ t_2 : S_2 \quad 0 \notin \text{fv}(S_2) \quad \Gamma \vdash\ S_2{\downarrow} <: T$

1. $t[N := v] \equiv (\texttt{let:} S_1 = t_1\ \texttt{in}\ t_2)[N := v] \equiv \texttt{let:} S_1[N := v] = t_1[N := v]\ \texttt{in}\ t_2[N + 1 := v]$.
2. By (IH), $(U_i[i := v])_i \vdash\ t_1[N := v] : S_1[N := v]$.
3. By (IH), $(U_i[i := v])_i, S_1[N := v] \vdash\ t_2[N + 1 := v] : S_2[N + 1 := v]$.
4. $0 \notin \text{fv}(S_2)$ and v closed implies $0 \notin \text{fv}(S_2[N + 1 := v])$.

5. By (IH), $(U_i[i := v])_i \vdash S_2{\downarrow}[N := v] <: T[N := v]$.
6. By Lemma A.36, $S_2{\downarrow}[N := v] \equiv S_2[N+1 := v]{\downarrow}$.
7. By rule (T-Let), $(U_i[i := v])_i \vdash \texttt{let:} S_1[N := v] = t_1[N := v] \texttt{ in } t_2[N + 1 := v] : T[N := v]$.

Case (T-New, S-Trans, S-Class, S-Virtual, S-Extends, S-Up).
Case (S-Down, S-Alias-Left, S-Alias-Right).

By applying (IH) on case hypotheses followed by the same typing or subtyping rule.

Qed.

A.8 Subject-reduction

Lemma A.40 (Subject reduction) *$\vdash t : T$ and $t \rightarrow u$ implies $\vdash u : T$.*

Proof: By induction on the typing derivation $\vdash t : T$.

Case (T-THIS). The hypotheses contain: $t \equiv \texttt{this}$

1. Case impossible because there is no rule for reducing `this` (or equivalently `this` is not typable in empty context by Lemma A.6).

Case (T-PARAM). The hypotheses contain: $t \equiv x$

1. Case impossible, because x is not reducible.

Case (T-VAR). The hypotheses contain: $t \equiv n$

1. Case impossible, because n is not reducible.

Case (T-SELECT). The hypotheses are:

$t \equiv p.f \quad \vdash p : C \quad (\texttt{val}\ f : S) \in C \quad \vdash S[p] <: T$

1. By Lemma A.7, $\vdash p : C$ implies p is a value v.
2. We reason by cases on the last rule used to derive $t \rightarrow u$.

   Case (R-PREFIX). The hypotheses are: $p \rightarrow p' \quad u \equiv p'.f$

   (a) By Lemma A.5, p value implies p irreducible, so this case is impossible.

   Case (R-SELECT). The hypotheses are:

   $p \equiv \texttt{new}\ B(\overline{f} = \overline{v}) \quad f \equiv f_i \quad u \equiv v_i$

   (a) By inversion, $\vdash v : C$ implies:

   $$\overline{f} \text{ disjoint} \qquad \vdash \overline{v} : \overline{U} \quad \overline{U} \text{ closed}$$
   $$\forall\, i, S.\ (\texttt{val}\ f_i : S) \text{ implies } \texttt{this} : B \vdash_{\texttt{struct}} U_i <: S$$
   $$\text{isComplete}(B, \overline{f}) \quad \vdash B <: C$$

   (b) After replacement, $f \equiv f_i$ implies $(\texttt{val}\ f_i : S)$.

   (c) By hypothesis, $(\texttt{val}\ f_i : S)$ implies $\texttt{this} : B \vdash_{\texttt{struct}} U_i <: S$.

   (d) By Lemma A.23, $\texttt{this} : B \vdash_{\texttt{struct}} U_i <: S$ implies $\texttt{this} : B \vdash U_i <: S$.

   (e) By rule (S-CLASS), $\vdash B <: B$.

   (f) By rule (T-NEW), $\vdash v : B$.

   (g) By substitution Lemma A.39 (Property 1), $\vdash v : B$ and $\texttt{this} : B \vdash U_i <: S$ implies $\vdash U_i <: S[v]$ (since U_i closed).

   (h) By rule (S-TRANS), $\vdash U_i <: S[v]$ and $\vdash S[v] <: T$ implies $\vdash U_i <: T$.

(i) By subsumption Lemma A.1, $\vdash v_i : U_i$ and $\vdash U_i <: T$ implies $\vdash v_i : T$.

Case (T-Let). The hypotheses are:

$t \equiv \texttt{let:} S = t' \texttt{ in } t'' \quad \vdash t' : S \quad S \vdash t'' : S' \quad 0 \notin \text{fv}(S') \quad \vdash S'\!\downarrow <: T$

1. We reason by cases on the last rule used to derive $t \rightarrow u$.

   Case (R-Local). The hypotheses are:

   $t' \rightarrow u' \quad u \equiv \texttt{let:} S = u' \texttt{ in } t''$

   (a) By (IH), $\vdash u' : S$.

   (b) By rule (T-Let), $\vdash \texttt{let:} S = u' \texttt{ in } t'' : T$.

   Case (R-Let). The hypotheses are: $t' \equiv v \quad u \equiv t''[0 := v]$

   (a) By Lemma A.4 (Property 1), $\vdash v : S$ implies $\forall k \in \text{fv}(S),\ k < 0$.

   (b) By rule (WF-Ctx), $\forall k \in \text{fv}(S),\ k < 0$ implies wellFormed(S).

   (c) By substitution Lemma A.39 (Property 3), $\vdash v : S$ and $S \vdash t'' : S'$ and wellFormed(S) implies $\vdash t''[0 := v] : S'[0 := v]$.

   (d) By Lemma A.4, $S \vdash t'' : S'$ implies $(\forall k \in \text{fv}(S').\ k < 1)$.

   (e) $(\forall k \in \text{fv}(S').\ k < 1)$ and $0 \notin \text{fv}(S')$ implies $\text{fv}(S') \equiv \emptyset$.

   (f) By Lemma A.37, $\text{fv}(S') \equiv \emptyset$ implies $S'[0 := v] \equiv S'$.

   (g) By Lemma A.31, $\text{fv}(S') \equiv \emptyset$ implies $S'\!\downarrow \equiv S'$.

   (h) By subsumption Lemma A.1, $\vdash t''[0 := v] : S'$ and $\vdash S' <: T$ implies $\vdash t''[0 := v] : T$.

Case (T-Call). The hypotheses are:

$t \equiv p.m(\overline{x} = \overline{t}) \quad \vdash p : C \quad (\texttt{def } m(\overline{x} : \overline{S}) : S) \in C \quad \vdash \overline{t} : \overline{S}[p] \quad \vdash S[p] <: T$

1. By Lemma A.7, $\vdash p : C$ implies p is a value v.

2. We reason by cases on the last rule used to derive $t \rightarrow u$.

   Case (R-Receive). The hypotheses are: $p \rightarrow p' \quad u \equiv p'.m(\overline{x} = \overline{t})$

   (a) By Lemma A.5, p value implies p irreducible, so this case is impossible.

   Case (R-Arg). The hypotheses are:

   $\overline{t} \equiv \overline{v}, t', \overline{t}' \quad t' \rightarrow u' \quad u \equiv p.m(\overline{x} = \overline{v}, u', \overline{t}')$

   (a) Let $\overline{u} = \overline{v}, u', \overline{t}'$ and $n = |\overline{v}|$. We show that $\forall i.\ \vdash u_i : S_i[p]$ by considering two cases.

   Case $(i \neq n)$.

   - $i \neq n$ implies $u_i \equiv t_i$, implies $\vdash u_i : S_i[p]$.

Case $(i \equiv n)$.

- $t_n \equiv t'$ and $u_n \equiv u'$.
- By (IH), $\vdash t' : S_n[p]$ implies $\vdash u' : S_n[p]$.

(b) By rule (T-CALL), $\vdash p.m(\overline{x} = \overline{v}, u', \overline{t}') : T$.

Case (R-CALL). The hypotheses are:

$$v \equiv \texttt{new}\ C'(\overline{f} = \overline{w}) \quad \overline{t} \equiv \overline{v} \quad C' \lhd A \quad (\texttt{def}\ m = t') \in A \quad u \equiv t'[v][\overline{x} \backslash \overline{v}]$$

(a) By well-formedness of $(\texttt{def}\ m = t')$, $A;\ \overline{x} : \overline{S};\ \epsilon \vdash t' : S$.

(b) By Lemma A.22, $C' \lhd A$ implies $\vdash C' <: A$.

(c) By inversion, $\vdash \texttt{new}\ C'(\overline{f} = \overline{w}) : C$ implies:

$$\overline{f} \text{ disjoint} \qquad \vdash \overline{w} : \overline{U} \quad \overline{U} \text{ closed}$$
$$\forall i, U.\ (\texttt{val}\ f_i : U) \text{ implies } \texttt{this} : C' \vdash_{\texttt{struct}} U_i <: U$$
$$\text{isComplete}(C', \overline{f}) \qquad \vdash C' <: C$$

(d) By (T-NEW), $\vdash C' <: A$ implies $\vdash v : A$.

(e) By substitution Lemma A.39 (Property 1), $\vdash v : A$ and $A;\ \overline{x} : \overline{S};\ \epsilon \vdash t' : S$ implies $\texttt{root};\ \overline{x} : \overline{S}[v];\ \epsilon \vdash t'[v] : S[v]$.

(f) Well-formedness of $(\texttt{def}\ m(\overline{x} : \overline{S}) : S) \in C$ implies $\texttt{this} : C \vdash \overline{S}$ WF.

(g) By Lemma A.3, $\texttt{this} : C \vdash \overline{S}$ WF implies $\text{fp}(\overline{S}) \subset \emptyset$.

(h) v closed implies $\text{fp}(\overline{S}[v]) \equiv \text{fp}(\overline{S})$.

(i) By rule (WF-CTX), $\text{fp}(\overline{S}[v]) \equiv \emptyset$ implies wellFormed($\texttt{root};\ \overline{x} : \overline{S}[v];\ \epsilon$).

(j) By substitution Lemma A.39 (Property 2), $\vdash \overline{v} : \overline{S}[v]$ and $\texttt{root};\ \overline{x} : \overline{S}[v];\ \epsilon \vdash t'[v] : S[v]$ and wellFormed($\texttt{root};\ \overline{x} : \overline{S}[v];\ \epsilon$) implies $\vdash t'[v][\overline{x} \backslash \overline{v}] : S[v][\overline{x} \backslash \overline{v}]$.

(k) By well-formedness of $(\texttt{def}\ m(\overline{x} : \overline{S}) : S) \in C$, $\texttt{this} : C \vdash S$ WF.

(l) By Lemma A.3, $\texttt{this} : C \vdash S$ WF implies $\text{fp}(S) \equiv \emptyset$.

(m) $\overline{x} \notin \text{fp}(S)$ implies $S[v][\overline{x} \backslash \overline{v}] \equiv S[v]$.

(n) By subsumption Lemma A.1, $\vdash t'[v][\overline{x} \backslash \overline{v}] : S[v]$ and $\vdash S[v] <: T$ implies $\vdash t'[v][\overline{x} \backslash \overline{v}] : T$.

Case (T-NEW). The hypotheses are:

$$t \equiv \texttt{new}\ C(\overline{f} = \overline{t}) \qquad \overline{f} \text{ disjoint} \qquad \vdash \overline{t} : \overline{U} \quad \overline{U} \text{ closed}$$
$$\forall i, S.\ (\texttt{val}\ f_i : S) \text{ implies } \texttt{this} : C \vdash_{\texttt{struct}} U_i : S$$
$$\text{isComplete}(C, \overline{f}) \qquad \vdash C <: T$$

1. We reason by cases on the last rule used to derive $t \to u$ (there is only one).

Case (R-FIELD). The hypotheses are:

$$\overline{t} \equiv \overline{v}, t', \overline{t}' \quad t' \to u' \quad u \equiv \texttt{new}\ C(\overline{f} = \overline{v}, u', \overline{t}')$$

(a) Let $\overline{u} = \overline{v}, u', \overline{t}'$ and $n = |\overline{v}|$. We show that $\forall\, i.\ \vdash u_i : U_i$ by considering two cases.

Case ($i \neq n$).

- $i \neq n$ implies $u_i \equiv t_i$, implies $\vdash u_i : U_i$.

Case ($i \equiv n$).

- $t_n \equiv t'$ and $u_n \equiv u'$.
- By (IH), $\vdash t' : U_n$ implies $\vdash u' : U_n$.

(b) By rule (T-New), $\vdash \texttt{new}\, C(\overline{f} = \overline{v}, u', \overline{t}') : T$.

Qed.

A.9 Soundness

Lemma A.41 (Multi-step subject reduction) $\vdash\ t : T$ *and* t *reduces in zero or more steps to* u *(noted* $t \rightarrow^* u$*) implies* $\vdash u : T$.

Proof: By induction on the length of the sequence of reductions in $t \rightarrow^* u$, using Subject-reduction (Theorem A.40).

Qed.

Lemma A.42 (Soundness) *If* Π *is a well-formed program,* $\Pi.main \rightarrow^* t$ *and* t *is not reducible, then* t *is a value.*

Proof:

1. Π well-formed implies there exists T such that $\vdash \Pi.\text{main} : T$ (Property 3 of well-formed programs in Definition 3.9).
2. By Lemma A.41, $\vdash \Pi.\text{main} : T$ and $\Pi.\text{main} \rightarrow^* t$ implies $\vdash t : T$.
3. Suppose now by contradiction that t is not a value. By the property of Progress (Theorem A.29), $\vdash t : T$ implies that there exists u such that $t \rightarrow u$, which contradicts the hypothesis that t is not reducible. It means that t is necessarily a value.

Qed.

Index

admissible rule, 83
auto-combination, 29

class
  virtual, 34
combination
  class, 50
  object, 28
  template, 28

inner class, 19

model of computation, 25

nominal type system, 33

object, 25
owner, 54

path, 74

self variable, 25
self recursion, 25
self reference, 25
subtyping
  general, 83
  structured, 90
super-call, 29
  dynamic, 29
  static, 29
super-selection, 29

template, 25
  atomic, 25
  combination, 25
type, 81
  expansion, 93
  lowering, 94
typing
  general, 83
  structured, 90

virtual type, 19

Bibliography

[1] Martin Abadi and Luca Cardelli. *A Theory of Objects.* Monographs in Computer Science. Springer Verlag, 1996.

[2] Philippe Altherr. *A Typed Intermediate Language and Algorithms for Compiling Scala by Successive Rewritings.* PhD thesis, EPFL, March 2006. No. 3509.

[3] Philippe Altherr and Vincent Cremet. Inner Classes and Virtual Types. EPFL Technical Report IC/2005/013, March 2005.

[4] Bruno Barras, Samuel Boutin, Cristina Cornes, Judicael Courant, Jean-Christophe Filliatre, Eduardo Gimenez, Hugo Herbelin, Gerard Huet, Cesar Munoz, Chetan Murthy, Catherine Parent, Christine Paulin-Mohring, Amokrane Saibi, and Benjamin Werner. The Coq proof assistant reference manual: Version 6.1. Technical Report RT-0203, INRIA, 1997.

[5] Kim B. Bruce, Luca Cardelli, and Benjamin C. Pierce. Comparing Object Encodings. *Information and Computation*, 155(1/2):108–133, November 1999. Special issue of papers from *Theoretical Aspects of Computer Software (TACS 1997)*. An earlier version appeared as an invited lecture in the Third International Workshop on Foundations of Object Oriented Languages (FOOL 3), July 1996.

[6] Kim B. Bruce, Martin Odersky, and Philip Wadler. A Statically Safe Alternative to Virtual Types. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 1445 of *Lecture Notes in Computer Science*, pages 523–549, Brussels, Belgium, July 1998. Springer-Verlag.

[7] Kim B. Bruce and Joseph C. Vanderwaart. Semantics-driven Language Design: Statically type-safe virtual types in object-oriented languages. In *Fifteenth Workshop on the Mathematical Foundations of Programming Semantics (MFPS)*, April 1999.

[8] Peter Canning, William Cook, Walter Hill, Walter Olthoff, and John Mitchell. F-bounded Polymorphism for Object-Oriented Programming. In *ACM Symposium on Functional Programming Languages and Computer Architecture (FPCA), London, England*, pages 273–280, September 1989.

[9] Castagna and Chen. Dependent Types with Subtyping and Late-bound Overloading. *INFCTRL: Information and Computation (formerly Information and Control)*, 168, 2001.

[10] Vincent Cremet, François Garillot, Sergueï Lenglet, and Martin Odersky. A Core Calculus for Scala Type Checking. In *Proceedings of the 31st International Symposium on Mathematical Foundations of Computer Science (MFCS)*, Springer LNCS, September 2006. Invited talk.

[11] Vincent Cremet and Grégory Mermoud. Generating Typing Proofs for Scaletta, June 2005. Semester project at EPFL.

[12] Nicolas G. de Bruijn. Lambda-calculus notation with nameless dummies: a tool for automatic formula manipulation with application to the Church-Rosser theorem. *Indag. Math.*, 34(5):381–392, 1972.

[13] 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.

[14] Erik Ernst. Propagating Class and Method Combination. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, LNCS 1628, pages 67–91, Lisboa, Portugal, June 1999. Springer-Verlag.

[15] Erik Ernst, Klaus Ostermann, and William R. Cook. A Virtual Class Calculus. In *Proceedings of the Symposium on Principles of Programming Languages (POPL)*, January 2006.

[16] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Java Series, Sun Microsystems, 1996. ISBN 0-201-63451-1.

[17] Atsushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In *ACM SIGPLAN Conference on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, October 1999. Full version in ACM Transactions on Programming Languages and Systems (TOPLAS), 23(3), May 2001.

[18] Atsushi Igarashi and Benjamin C. Pierce. Foundations for Virtual Types. In *European Conference on Object-Oriented Programming (ECOOP), Lisbon, Portugal*, June 1999. Also in informal proceedings of the *Workshop on Foundations of Object-Oriented Languages (FOOL)*, January 1999. Full version in *Information and Computation*, 175(1): 34–49, May 2002.

[19] 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.

[20] 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 (ECOOP)*, Darmstadt, Germany, July 2003.

[21] Kresten Krab Thorup. Genericity in Java with Virtual Types. In *European Conference on Object-Oriented Programming (ECOOP), Jyväskylä, Finland*, volume 1241 of *Lecture Notes in Computer Science*, pages 444–471. Springer-Verlag, June 1997.

[22] Mads Torgersen. Virtual Types are Statically Safe. In *International Workshop on Foundations of Object-Oriented Languages (FOOL)*, informal proceedings, January 1998.

[23] Mads Torgersen, Erik Ernst, and Christian Plesner Hansen. Wild FJ. In *FOOL'12: Proceedings of The Twelth International Workshop on Foundations of Object-Oriented Languages*, 2005.

[24] David Ungar and Randall B. Smith. Self: The power of simplicity. *Lisp and Symbolic Computation*, 4(3):187–205, 1991.

[25] Andrew K. Wright and Matthias Felleisen. A Syntactic Approach to Type Soundness. *Information and Computation*, 115, 1994.

Curriculum Vitæ

Personal information

Name	Vincent Cremet
Citizenship	France
Date of birth	Februar 11th, 1975
Place of birth	Belfort, France

Education

2001–2006	Ph.D., Laboratoire des Méthodes de Programmation (LAMP), EPFL, Switzerland
1999–2000	DEA d'Informatique "Programmation : Sémantique, Preuves et Langages", Université Paris 7
1996–1999	Ecole Nationale Supérieure des Mines de Nancy, option Informatique
1995–1996	Licence de Mathématiques, Université Claude Bernard, Lyon 1
1993–1995	DEUG A, Université Claude Bernard, Lyon 1
juin 1993	Baccalauréat série C (scientifique), Lycée Lamartine, Mâcon

Professional experience and Projects

2001–2006	Teaching Assistant, Laboratoire des Méthodes de Programmation (LAMP), EPFL
2000	Diploma project (DEA), Design and implementation of a model checker for a real-time language based on linear logic, supervised by J-P. Jouannaud et M. Okada, LRI (Université d'Orsay–Paris Sud)
1999	Diploma project (Ecole des Mines), (5 months), C++ coding rules and generation of design patterns from UML, Dassault Electronique, Paris