Simple Dependent Types: Concord
Final Year Project Report
Paul Jolly

Supervised by Dr Sophia Drossopoulou and Christopher Anderson
Second markers: Dr Nobuko Yoshida and Susan Eisenbach

June 16, 2004
Abstract

The organisation of code at an object level has long been considered a prudent approach to software development. However, in many situations the single-class perspective is too fine grained a perspective on a system and collaborations or families of classes are a better unit of organisation.

CONCORD is a calculus designed to handle families, collaborations between multiple classes, and subsequent specialisations of this family concept. The calculus is a simple model for a restricted form of dependent types in object oriented languages. We introduce groups to represent this notion of a family, whereby classes belong to groups and dependency is introduced via intra-group references using the MyGrp keyword.

The work is based on previous work by Anderson et al. and Ernst; it is hoped that the simple calculus proposed in the form of CONCORD will provide a valuable insight into the development of a formal system for a dynamic form of class family management.
Acknowledgements

First and foremost, my sincere gratitude to Dr. Sophia Drossopoulou for her time (including the odd weekend), encouragement, patience and good humour; unquestionably, Sophia has taught me anything that I know about good technical writing and research.

My special thanks also to Christopher Anderson who, as joint supervisor to this project and co-author of an original work on which the topic is based, has provided invaluable experience and advice throughout the project lifetime. In particular, Chris has been unfailing with advice on matters of project and time management, something for which I am especially grateful.

Many thanks to Susan Eisenbach for her sound, considerate and well-reasoned advice on all matters project related and her forbearance on issues SSC related.

Finally, “Even More” thanks to the PhD Students of SLURP; for the recipe suggestions, the enigmatic impressions and a genuinely refreshing insight into the world of research that may yet encourage me to return to university life.
Contents

1 Introduction and Motivation 1
   1.1 Motivation .................................................. 1
   1.2 Contribution .................................................. 2
   1.3 This Report .................................................. 4

2 Background 5
   2.1 Object Oriented Development ............................... 5
      2.1.1 Basic Concepts .......................................... 6
   2.2 Mixins ......................................................... 11
   2.3 Mixin Layers .................................................. 12
   2.4 Family Polymorphism ........................................ 14
   2.5 \V J and Virtual Classes ................................. 16
   2.6 JAVA .......................................................... 22
      2.6.1 Member Inner Classes .................................... 22
      2.6.2 GJ and J2SE 1.5 ......................................... 24

3 The Formal System 25
   3.1 Syntax ......................................................... 25
## Contents

3.1.1 Example ...................................................... 27
3.1.2 Motivation ................................................. 28
3.2 Unique definitions, acyclic groups and acyclic types ................. 29
3.3 Inheritance .................................................... 29
   3.3.1 Example .................................................. 32
   3.3.2 Motivation .............................................. 33
3.4 Well-formedness of programs, groups and classes .................... 34
   3.4.1 Example .................................................. 36
   3.4.2 Motivation .............................................. 36
3.5 Execution ..................................................... 39
3.6 Subtypes and the Type System .................................. 41
   3.6.1 Example .................................................. 45
   3.6.2 Motivation .............................................. 46
3.7 Soundness and Decidability ..................................... 46

4 Proofs 49

4.1 Lemmata and Theorems ......................................... 49
   Lemma 1 ....................................................... 49
   Lemma 2 ....................................................... 49
   Lemma 3 ....................................................... 50
   Lemma 4 ....................................................... 54
   Lemma 5 ....................................................... 54
   Lemma 6 ....................................................... 59
   Lemma 7 ....................................................... 59
   Lemma 8 ....................................................... 61
Chapter 1

Introduction and Motivation

The organisation of code at an object level has long been considered a prudent approach to software development. However, in many situations the single-class perspective is too fine grained a perspective on a system and collaborations or families of classes are a better unit of organisation [16, 3].

Dependent types are just one approach to this end. Based on recent work in Anderson et al. [3], this project seeks to develop a small calculus capable of supporting dependent types, develop an implementation based on this calculus and extend the $\mathcal{L}_2$ language to support these features.

1.1 Motivation

Most commercial object oriented languages do not directly support code re-use combined with the expression of dependencies between classes. Consider the familiar graph example: regular graphs comprise edges connecting nodes while coloured graphs comprise edges connecting coloured nodes. This can be expressed through classes Node, Edge and ColouredNode, with some form of code reuse between Node and ColouredNode. A method Edge connect(Node x) in class Node creates an edge between receiver and argument. In order to maintain consistent graphs (a design decision), we restrict regular graphs to contain only regular nodes and coloured graphs to contain only coloured nodes. Thus, a receiver of type Node should be forbidden from calling connect(…) with an argument of type ColouredNode. A conventional encoding (perhaps in JAVA) would achieve code reuse through subclassing, making ColouredNode a subclass of Node (this can be seen in Listing 1.1). However, in such languages subclasses create subtypes. Thus, ColouredNode would be a subclass (and subtype) of Node: with a receiver of type Node, a call of the form connect(ColouredNode) is type correct, a situation that violates our original requirement.

Solutions to the above problem have already been suggested using family polymorphism [16], in the programming language SCALA [37] and in [12]. In this report we present CONCORD, a simple approach to the problem inspired by ideas from [12], less
1. Introduction and Motivation

```java
class Node {
    Edge connect(Node n) {... }
}

class Edge {
    Node end1, end2;
}

class ColouredNode extends Node {
    Colour c;
}
```

Listing 1.1 — A Java encoding of the graph example

powerful than Scala [37]. To our knowledge, CONCORD is the first work that combines a simple solution, an imperative model, a decidable system and a sketch–proof of soundness.

1.2 Contribution

This report presents a detailed formal system for a calculus designed to handle families, collaborations between multiple classes, and subsequent specialisations of this family concept. The calculus, CONCORD, is a simple model for a restricted form of dependent types in object oriented languages. CONCORD introduces groups to represent this notion of a family, whereby classes belong to groups and dependency is introduced via intra-group references using the `MyGrp` keyword. It is based on the following key ideas:

- Groups contain classes, thus allowing the expression of related classes.
- We distinguish absolute and relative types: Absolute types consist of a group and a class name, thereby expressing inter-group dependencies. Relative types consist of a reference to the current group, `MyGrp`, and a class name, thereby expressing intra-group dependencies.\(^2\)
- Groups may extend other groups; classes defined in a group \(g'\) (the supergroup) are further bound in any subgroup, \(g\), that extends the supergroup.
- A class \(g.c\) may extend another class \(gr.c'\) (\(gr\) is a group reference and is either a group name or `MyGrp`), in which case it is a subclass of \(gr.c'\).

\(^2\)Thus, relative types support a restricted form of dependent types, whereby a type may depend on a group — rather than on a value as in “full” dependent types.
Subclasses and further binding induce inheritance: If \( g.c \) is a subclass of \( gr.c' \), then it inherits all members (fields and methods) from \( gr.c' \), replacing occurrences of \( MyGrp \) in the member definitions by \( gr \) in the process. If \( g.c \) further binds \( g'.c \) (i.e. \( g \) extends \( g' \)), then it also inherits all members from \( g'.c \), but instead leaves occurrences of \( MyGrp \) in member definitions unmodified. Furthermore, subclasses create subtypes (i.e. if \( g.c \) is a subclass of \( gr.c' \) then \( g.c \) is a subtype of \( gr.c' \)), while further binding does not\(^3\). Classes may both further bind and subclass other classes thus introducing a limited form of multiple inheritance.

A CONCORD encoding of the graph example is presented in Listing 1.2; a graphical representation can be seen in Figure 1.1\(^4\). The group Graph contains definitions for two classes, Edge and Node; the latter defines a method MyGrp.Edge.connect(MyGrp.Node x). The group ColouredGraph extends Graph; hence, classes ColouredGraph.Node and ColouredGraph.Edge further bind Graph.Node and Graph.Edge respectively. In addition, ColouredGraph.Node extends inherited definitions by defining the field \( c \), of type Colour.Colour. Therefore, ColouredGraph.Node inherits the method MyGrp.Edge.connect(MyGrp.Node). Considering variables \( n1, n2 \) of type Graph.Node and \( cn1, cn2 \)

\(^3\)Thus, subclasses correspond to inheritance in Java; further binding is as per suggestions in [49]

\(^4\)Refer to Section A.2 for an explanation of our diagrammatic notation
of type ColouredGraph.Node, the expressions \texttt{n1.connect(n2)} and \texttt{cn1.connect(cn2)}, which return objects of types \texttt{Graph.Edge} and \texttt{ColouredGraph.Edge} respectively, are type correct, whereas \texttt{n1.connect(cn1)} and \texttt{cn1.connect(n1)} are type incorrect.

## 1.3 This Report

Chapter 2 discusses object oriented programming in the context of collaborations between classes, providing an introduction to the area by an exploration of previous work. A review of the basic concepts in object oriented programming is also provided, for the benefit of the reader unfamiliar with the paradigm. Chapter 3 represents the body of this report where we present, in detail, the formal system that underpins the CONCORD calculus. The importance of this Chapter cannot be underestimated; we are therefore careful to explain and motivate our design decisions for each of the constituent parts of the formal system. Chapter 4 contains the lemmata and theorems required in the proof of soundness of the CONCORD type system. Chapter 5 introduces CONCORD encodings of well known examples from the literature to further motivate our design decisions for the calculus. The comparisons in Chapter 6 are designed to distinguish particular aspects of CONCORD’s design from previous work, highlighting our contributions to the field. In Chapter 7 we conclude by summarising the contributions of this project but also by providing a personal reflection on the progress of our work. In light of the publication of a paper [30] on the subject, a situation unimaginable at the beginning of the project, this personal reflection is interleaved with discussion of initial proposals that consequently proved impossible to address because of time constraints. Appendix A presents an explanation of the diagrammatic format used throughout this report and auxiliary definitions to the formal system.

It should be noted that the bibliography to this thesis contains items not referenced from within the report body. This project report does not aim to present a step-by-step account of how the CONCORD calculus developed; we have therefore included mention of every document referenced during the project lifetime in recognition of the respective contributions to the CONCORD calculus.
Chapter 2

Background

This chapter gives an introduction to the concepts of object oriented programming and the motivation behind the extension of the object model with collaborations of classes. After providing an explanation of common terminology, we present examples of the various techniques proposed as solutions to the interaction between objects so common in complex systems, focussing on a discussion of VJ and virtual classes. We conclude with exploration of the support within Java for collaboration based programming.

2.1 Object Oriented Development

Object oriented programming maps real world concepts to classes in a programming language\(^1\). These classes represent a formalisation of understood common characteristics in the concept. Objects (instances of these classes) represent instances of the real-world concepts. Simulation of real world situations is modelled by the interaction between these objects. This class mapping is very natural and the interaction via simulations articulates our (mis)understanding of the real-world concept.

As in the real world, objects interact with their surroundings but do so in specified ways: this is modelled by the interface that a class presents. Encapsulation and information hiding are important in this respect as they protect the implementation detail just as a car hides the details of its engine workings under the bonnet.

Various texts have recognised that complex systems rarely consider interaction at merely the object level, but instead refer to slices of behaviour in a system [3]. This has led to various implementations trying to encapsulate the concept of role based design. Figure 2.1 presents a very abstract indication of the progression of ideas subsequent to object oriented programming.

The surveys included in this Chapter have, in part, been motivated by Figure 2.1.

Mixins [6], whilst not strictly a means of grouping objects, define abstract subclasses of behaviour that can subsequently by applied to an unlimited number of superclasses. Therefore, all classes utilising a common mixin loosely belong to a family determined

\(^1\)The word class is perhaps a misnomer in this mapping, suggesting that a group of objects or concepts are being considered. We use the object oriented meaning of the word.
by the mixin-supplied behaviour. Mixins suffer, however, from highly complex parameterisation in the presence of multiple classes [53] and led Smaragdakis and Batory to develop mixin layers.

*Mixin layers* [53] abstract away from the fine grained approach of mixins and scale to multi-class granularity. They are a technique for implementing layered object oriented designs (e.g., collaboration-based designs). This can be considered a grouping of classes. But Ostermann in [49] demonstrates that subtyping amongst these collaborations is too restrictive.

*Family polymorphism* [16] and \( \mathcal{VJ} \) [3] take this family based approach a step further by allowing true polymorphism amongst family objects. Ernst [16] describes an implementation of this approach via \( \text{gbeta} \). \( \mathcal{VJ} \) is an attempt to formalise the work by Ernst. Both approaches focus on grouping via instances of classes (objects), using dependent types to control mixing of family objects.

*Member inner classes* and *generics*, the latter as proposed by GJ, are two mechanisms that support collaboration based programming in *Java*. We explore the support this state-of-the-art language offers to programmers.

The remainder of this Chapter presents surveys of the techniques mentioned above, however we begin with a recap of basic object oriented programming concepts.

### 2.1.1 Basic Concepts

In order that the reader may become familiarised with the notation we shall use in describing methods for encoding collaborations, we now present a short section covering object-oriented terms and language\(^2\). This section can safely be skipped by the advanced reader already familiar with the paradigm.

*Abstraction* refers to the identification of important aspects in a system, ignoring its details. Abstraction can take two forms: behavioural and structural. *Encapsulation* refers to information hiding within abstractions. It hides implementation details, pro-

---

\(^2\)This and subsequent sections should be read in conjunction with Section A.2 where we provide an explanation of notation and conventions used.
A class represents a formal notation describing the abstracted characteristics of a set of objects. In class terminology, structural characteristics are termed fields or attributes and behavioural characteristics methods. An object (singular) is an instance of a class.

Generalisation is a relationship between a more general kind of class and a more specific one. The reverse relationship is called specialisation.

A subclass extends a class: the class extended by a subclass is referred to as the superclass. Subclasses inherit methods and attributes from their superclass. In specifying how subclasses differ from the superclass, a subclass may override method definitions it would otherwise have inherited from the superclass. This is termed incremental specification.

A class $A$ is a subtype of a class $B$ if an expression of type $A$ can be used in any context that expects an expression of type $B$ [10]: we write $A <: B$. This is expressed via the Liskov Substitution Principle[39]:

“If for each object $o_1$ of type $S$ there is an object $o_2$ of type $T$ such that for all programs $P$ defined in terms of $T$, the behaviour of $P$ is unchanged when $o_1$ is substituted for $o_2$ then $S$ is a subtype of $T$.”

Furthermore, the principle of subsumption states that if $A <: B$ then an expression of type $A$ also has type $B$.

We summarise the notation described thus far in Figure 2.2. The concept is a lamp. In this simple case, lamps have a colour and can be switched on/off by a user. Furthermore, there is a special sort of lamp whose brightness can be adjusted by said user. AdjLamp is a subclass of Lamp: we write AdjLamp <: Lamp.

The Liskov substitution principle requires that wherever some code expects an object of type Lamp, we can safely present an object of type AdjLamp. The following code listing...
shows a code snippet containing a method definition. In our simple example, we can safely supply an object of type \texttt{AdjLamp} as a parameter to the method:

```java
void putLampInCorner(Lamp l)
{
    ...,
}
...
```

Not only is \texttt{AdjLamp} a subclass of \texttt{Lamp}, it is also a subtype. In our simple example we can see by inspection that this is the case. \texttt{AdjLamp} inherits every definition from \texttt{Lamp}; the only difference between the two classes is the provision of \texttt{Dim()} in \texttt{AdjLamp}. In Bruce et al. [10], the author presents an example (using binary methods) where a subclass relationship does not generate a subtype relationship. This occurs when the types of methods need to change in subclasses to require more specialised behaviour from their arguments [10].

It is true to say that all subtype relationships are also subclass relationships but not the converse. Our diagrammatic distinction is presented in Section A.2.

A class defined as \textit{abstract} cannot be instantiated. Classes containing \textit{abstract methods}, methods which are simply declared with no definition, are abstract by implication. Abstract classes may be useful when only partial implementations can be provided, thereby delegating the responsibility to implement full behaviour to a concrete subclass. Typically, abstract classes are used to represent abstract concepts, e.g. shapes.

We now consider definitions of \textit{covariant}, \textit{contravariant} and \textit{invariant} (with respect to types) by way of example. Consider the JAVA like code snippet in Listing 2.1. In refining \(X^3\), class \(Y\) refines each of the methods defined in class \(X\). We consider each of the methods in turn:

- In method \texttt{m1}, the type of the parameter varies with the inheritance hierarchy of the enclosing class, that is \(Y <: X\) and \(B <: A\). This is termed \textit{covariant} redefinition\(^4\) of the argument type.

- In method \texttt{m2}, the type of the parameter varies inversely with the inheritance hierarchy of the enclosing class, that is \(Y <: X\) but \(B :> C\). This is termed \textit{contravariant} redefinition of the argument type.

- In method \texttt{m3}, the type of the parameter is unchanged. This is termed \textit{invariant} redefinition of the argument type.

It is worth noting that in JAVA, the redefinition of \texttt{m1} and \texttt{m2} would in fact be an \textit{overloading} of these methods and not a redefinition. However, an example of

\(^3\)The receiver (an instance of \(X\) or \(Y\)) is not relevant in discussion of these concepts. The methods are declared within a class simply to maintain consistency throughout this chapter. The issue of binary methods is more fully explored in Bruce et al. [10]

\(^4\)As described in [5, 10]
Listing 2.1 — Covariant, contravariant and invariant redefinition. Method bodies have been
omitted and only the types of parameters provided.

type covariance can be seen in Java arrays. As String <: Object, then in Java
String[] <: Object[].

More generally, the refinement of any parameterised definition (e.g. a method’s pa-
rameter or an array’s type) can be considered covariant, contravariant or invariant. For
such a definition \( f \), parameterised by \( p \), a refinement \( f’ \) with parameter \( p’ \) is:

- covariant if \( p :> p’ \)
- contravariant if \( p <: p’ \)
- invariant if \( p \equiv p’ \)

Discussion so far has considered references to objects using implicit pointers. A
pointer is a reference to an area of memory. Compare C++ where a distinction is made
between a pointer variable and a ‘solid variable’. In C++, given a type, \( T \), a pointer
to \( T \), written \( T* \), can hold the address of an object of type \( T \) (see Stroustrup [55]). We
adopt the approach used in JAVA where pointers are implicit and ‘solid variables’ are
not possible.

Polymorphic variables may refer to objects whose classes are statically unknown
[24]. At runtime, such variables respond according to the class of the object to which
they refer. Pointers (implicit or otherwise) are examples of such variables. For example,
Figure 2.3 — Example of polymorphic references

Figure 2.3 shows such a polymorphic reference, \( x \), that refers\(^5\) to objects \( \xi^C \). Assume that \( C_1 \) implements the method \texttt{print} and that subsequent \( C_{i>1} \) refine this method. *Late binding*, with regard to the variable \( x \), ensures that the \texttt{print} method invoked in the call \( x.\texttt{print()} \) is the refinement appropriate to the object to which \( x \) currently points. In the diagram, this would be the refinement as supplied by the class \( C_3 \).

The term *dependent types* describes how types can depend on the value of expressions. With this functionality, we can structurally express required relationships that can be checked statically. Consider the \texttt{JAVA-like}\(^6\) method signature in Listing 2.2. Here, the method \texttt{m} takes three arguments. By defining the expression \texttt{Graph g} as final (constant) we guarantee that, during evaluation of arguments and the method body, the types of \( e \), \( n \) and the method will not alter. Observe the types of the second and third parameters and the method’s type are dependent on the evaluation of the first argument. Type evaluation is therefore achieved via an instance of an object (in this example \( g \), an instance of \texttt{Graph} or one of its subtypes). This places restrictions on dependent types, giving rise to the family containment described by Ernst in [16] (see Section 2.4 for further explanation). Arrays are another example of dependent types: \texttt{int[3]} and \texttt{int[5]} are two different types. The type of an array declaration depends on the dimension, in this case 3 and 5.

We now introduce the various techniques proposed as solutions to the definition of interactions between collaborations of classes.

\(^5\)We use \( \xi^C \) to range over objects of type \( C \). Where a hierarchy of subclasses is defined, for example \( \texttt{C_1 <: C_2 <: C_3 ...} \), we use the polymorphic \( \xi^{C_i} \) to range over objects of type \( C_1, C_2, C_3 ... \).

\(^6\)JAVA-like but bearing no relation to permitted Java syntax
2.2 Mixins

A mixin\(^7\) is an abstract subclasses. That is, a mixin specifies the nature of an incremental extension without specifying what it may extend. We wish this extension to be valid for a number of different classes.

Take as an example a CAD system which defines DiagramDisplay objects that display diagrams. In certain situations we may require a display with scroll bars, in other situations we might like to overlay a grid on top of the display. The scroll bar and grid classes are mixins. They can be applied to any class presenting the DiagramDisplay interface (the DiagramDisplay class is one such example), creating a scroll-bar window and overlayed-grid window respectively.

The concept of mixins is most clearly imagined via parameterised classes in C++\(^8\):

```cpp
template <class Super>
class Mixin : public Super
{
    /* mixin definition */
};
```

The superclass of the mixin is specified by a parameter. Any use of the above template will be expanded by the compiler and the parent class set accordingly: mixins are defined and resolved statically (admittedly out of the programmer’s sight).

Returning to our example:

```cpp
class DiagramDisplay
{
    // implementation to display a diagram in a window
};
```

\(^7\)The term mixin has many definitions (cf. linearization in CLOS, common virtual base classes in C++ etc.) — we adopt the definition as presented by Bracha and Cook [6].

\(^8\)C++ parameterised inheritance represents one method for implementation. Others include linearization of multiple inheritance in CLOS. We present examples in C++ in the hope that the majority of readers have encountered the language.
2. Background

We define two mixins, ScrollBar and Grid, using parameterised inheritance. To utilise these mixins, we might then create a diagram display overlayed with a grid that uses scroll bars to view the entire document. This is achieved by the following type definition:

```cpp
template <class Super>
class ScrollBar : public Super
{
    // implementation to display scroll bars on window
};

template <class Super>
class Grid : public Super
{
    // implementation to overlay grid on window
};
```

The mixin is therefore only a programming construct designed to increase the reusability of code. However, if a mixin template is defined to extend more than one class, we immediately have a family of related class linked by the mixin-provided concept.

In [6], Bracha and Cook discuss mixin based inheritance using CLOS, specifically with reference to the `call-next-method` construct the language provides. They discuss composition of mixins as a formulation of inheritance achieved by considering a mixin as a primitive construct.

2.3 Mixin Layers

In [59] and other similar papers, collaboration based programming gained a face. Smaragdakis and Batory clearly articulate the approach in [53] by describing how collaboration based designs decompose an object-oriented application into a set of classes and collaborations. Classes within an application specify the roles objects of the respective class play within the application. It is likely an object of a particular class will participate in multiple collaborations in an application; this is visualised in Figure 2.4. A collaboration is thus identified as a set of objects and an associated protocol determining how the objects are permitted to interact.

In [59], VanHilst and Notkin adopt an approach that maps roles into class mixins. They view roles in two ways: in terms of participants/types involved and the tasks/concerns of the design. Composition of roles is achieved through parameterisation of the mixin role classes. Smaragdakis and Batory comment on how the VanHilst and Notkin approach suffers from over complicated parameterisation of templates in the presence of multiple classes and the inability to contain intra-collaboration design changes.

---

9Diagram due to VanHilst and Notkin and Smaragdakis and Batory. For a concrete example utilising this style of diagram see [59].
And so they go on to describe *mixin layers*, an approach that abstracts away from the fine-grained approach of role mixins. We present the idea of a mixin layer again using C++:

```cpp
template <class NextLayer>
class OuterMixin : public NextLayer
{
    /* mixin definitions */
    class InnerMixin1 : NextLayer::Mixin1
    {
        /* implementation */
    };
    class InnerMixin2 : NextLayer::Mixin2
    {
        /* implementation */
    };
    ...
};
```

The encapsulated mixins, `InnerMixin1`, `InnerMixin2` etc., are called *inner mixins*. The containing mixin, `OuterMixin`, is called the *outer mixin*. The outer mixin (`OuterMixin`) is termed a *mixin layer* when parameterised by a class (`NextLayer`) that satisfies all the inner mixins. We coin the term *super layer* to refer to `NextLayer` in the context of `InnerMixin1`.

Mixin layers can be composed with other mixin layers. Take the example provided by the diagram in Figure 2.4:
typedef c3 < c2 < c1 > > t1;

In this example, we can see the provision of roles is not necessarily achieved directly by the super layer, but instead may be defined further up the inheritance chain. More formally, mixin layers can inherit entire classes from the above layer; inner mixins can inherit members from corresponding inner mixins in their super layer.

Mixin layers recognised that the unit of functionality is not restricted to single objects or even parts of these objects. Rather a collaboration of objects, possibly only utilising parts of contained objects, is the base unit. Furthermore, that refinement of these collaborations is desirable.

In [49], Ostermann highlights two flaws of mixins layers. Firstly that the concept of subtyping amongst collaborations is too restrictive. Back to our example and the following type definitions:

typedef c2 < c1 > t2;
// t1 as before

We might, quite reasonably, expect t1 to be a subtype of t2: with mixin layers this turns out not to be the case. Polymorphism amongst collaborations is therefore not possible.

The second problem relates to composition consistency and is most clearly explained in [49] via the graphing example.

2.4 Family Polymorphism

In [16], Ernst takes polymorphism to the multi object level (a weakness of mixin layers mentioned in Section 2.3). Ernst first makes a significant observation about traditional single object polymorphism (with reference to Figure 2.3): the call-site x.print() is reused for the ξCi, where print is defined in class Ci, independent of the exact class of the object referred to by x and the implementation of print and that this set of class/method pairs is open-ended.

Consider the toy example where mixing of family members would not be desirable, seen in Figure 2.5. Presumably we want to prevent normal drivers from driving racing cars for insurance purposes. In a similar vein, we prevent racing drivers from using normal cars to keep our roads safe. Between the two inheritance hierarchies we only allow pairings of objects from the equivalent level of specialisation. Michalitis [44] refers to this notion as “interlocked” types10.

The work in [16] is motivated by the following example. Consider the case, visualised in Figure 2.6, where a method fail, defined in the hierarchy of C, takes a parameter from the equivalent position within the hierarchy of D. The format of a call via a ξCi reference x might therefore be x.fail(y), where y is a ξDi reference. C++ and JAVA can only associate two static types with this expression: the statically know type of x,

10In fact, Michalitis is more specific in the definition requiring that multiple subclasses at the same level of the inheritance hierarchy also be correctly paired. See the Subject-Observer example in section 6.3 of [44] for further explanation.
Ernst defines the term *family polymorphism* — a programming language feature that allows us to statically declare and manage relations between several classes polymorphically. Ernst agrees that we must consider families of classes but that when more than one variant of such a class exists we must be careful not to mix members from variants. With gbeta he seeks to combine the concept of reuse described above whilst maintaining

\[ C_x, \text{ and the statically known argument type of } \text{fail}, \ D_{\text{fail}}. \] Therefore at runtime, \( x \) may refer to any \( \xi^{C'} \) such that \( C \geq C' \) and \( y \) may refer to any \( \xi^{D_{\text{fail}}'} \) such that \( D_{\text{fail}} \geq D'_{\text{fail}} \).

In these two languages\(^{11}\) there is no static technique to ensure correct pairing of \( x \) and \( y \) objects.

\(^{11}\)Ernst continues to describe the scope of this problem in the context of other languages.
the safety that prevents us from mixing members from unrelated families. Both safety and reuse are made possible by the concept of dependent types.

We are asked to consider classes as attributes of objects: this leads to the the definition of an object as a repository of classes or class family. We then build up inheritance by defining variants of the enclosing objects (Graph in Listing 2.2): one such refinement can be seen in Listing 2.3. The declaration of the relation between a family of classes is now explicit (cf. sections 3.1 and 3.2 in [16]).

Association between patterns is therefore made via instances and not classes. For example, the object myGraph (of type Graph) is a repository/class family, whose members (Node and Edge) are accessed via myGraph.Node and myGraph.Edge. Furthermore, for two such instances of Graph, g1 and g2, unless the two can statically be proved to be the same object, g1.Node and g2.Node must be assumed to be completely unrelated.

Via polymorphism of the family object e (Listing 2.2), the method m(...) will accept any object that is a subtype of Graph (one such example is ColouredGraph). Furthermore, through the use of dependent types we guarantee never to mix members from different families. Hence the phrase, family polymorphism.

2.5 $\mathcal{V}J$ and Virtual Classes

$\mathcal{V}J$ is the calculus described by Anderson et al. in [3] supporting virtual classes and dependent types. It is an attempt to formalise previous work by Ernst.

To understand virtual classes, let us first introduce virtual procedures / methods. Subclasses allow us to incrementally specify specialisations of a superclass by adding attributes and or methods. Furthermore, languages like C++ allow us to refine procedures defined in the super class: such methods are termed virtual.

And so the definition of a virtual class follows. A virtual class may be extended in a subclass of the defining class enabling us to defer the definition of reference types[41]. To this end, $\mathcal{V}J$ defines two operators: ::< and ::::<. As in BETA, ::< expresses a subclass relation. ::::< is the further bind operator that allows this extension of a contained virtual class.

Let us consider the example presented in [3]: the code is reproduced in Listing 2.4.

---

12 For an amusing example highlighting the consequences of mixing family members, see the beginning of [16].

13 In C++, we cannot redefine non-virtual methods.

14 Again, this is JAVA-like syntax with subclassing expressed via the BETA :: operator.
We define the virtual class `ExprFamily` to contain virtual classes `Expr`, `Num` and `Plus`, the intention being to define a family container (`ExprFamily`) for expressions. The Listing continues to supply an incremental specification of `ExprFamily` in the form of `EvalExpr` (so as to allow for evaluation of our expressions). `EvalExpr` is a subclass of `ExprFamily` where we further bind the contained virtual classes `Expr`, `Num` and `Plus`.

At first glance, it would seem very natural to visualise the relationships between `ExprFamily` and `EvalExpr` as is suggested in Figure 2.7(a). It is true to say the lines indicate inheritance relationships between virtual classes but incorrect to therefore infer type relationships between virtual classes (as in strongly typed languages like C++ and Java).
2. Background

We therefore extend Figure 2.7(a) in Figure 2.7(b) to demonstrate more precisely the relationships (through inheritance and actual type subsumption) between the two containing classes.

Thus we now are far more careful in distinguishing instances of pure inheritance from inheritance that generates subtype relationships\textsuperscript{15}. As there is no type relationship between the \texttt{Expr} of family \texttt{ExprFamily} and the \texttt{Expr} of family \texttt{EvalExpr} we are guaranteed that any code mixing the two would not type check. Informally therefore we may understand the \textit{further bind} relation as inheritance that conveys no type relationship between the pair in the relation.

To conclude discussion of virtual classes, we turn to an example of final binding, seen in Listing 2.5\textsuperscript{16}, an extension of our expression example. \texttt{ObjectCell} is a new

\textsuperscript{15}Java and C++ are two examples of languages where the presence of a subclass relationship implies a subtype relationship between superclass and subclass

\textsuperscript{16}Again due to Anderson et al.
virtual class that contains a dependent type member variable `item` (line 4). `ItemType` is itself a virtual class defined as a subclass of the global `Object` class (line 3).

`ExprCell` is then defined as a subclass (and subtype) of `ObjectCell`. `ExprCell` contains the field `e` (line 8). Furthermore, `e` is an abstract field as it has not yet been assigned a value; `ExprCell` cannot therefore be instantiated, only extended by a subclass. `ItemType` is also final bound (line 9) to the dependent type `e.Expr` within `ExprCell`. The final binding of `ItemType` to a type dependent on an abstract field is permitted as `ExprCell` can never be instantiated: any subclass that can be instantiated is not permitted to contain abstract fields and so will have assigned a value to `e`, constrained by the upper bound of the static type.

The final binding of `ItemType` disallows any further binding of the virtual class in subsequent concrete subclasses, only assignment in accordance with the afore mentioned upper bound. We can consider the left hand side of a final binding to be an alias for the right hand side.

The final binding of a value (line 16) warrants further discussion. This statement

---

17 Dependent types are as per the concept defined in Section 2.4.
18 The availability of a Java-like `Object` superclass is assumed.
19 Anderson et al. in [3] define fields as immutable properties of objects `cf. final` fields in Java.
allows us to statically infer that MyExprCell.e is an alias for ef1. Combined with the final binding of ItemType the assignments on lines 21 and 22 can be proved statically type-safe.

Returning to our discussion about subtyping. We can now present a complete type relationship diagram for our expression example. Figure 2.8 shows that all type relationships are intra-family, even where virtual classes are further bound in a subclass of the containing family.

Any call to super from a virtual class is a reference to the inherited superclass. For example, the expression super.print() on lines 8 and 15 in Listing 2.4 is a reference to the print() method of Expr in the containing family ExprFamily (line 3). Similarly the call from EvalExpr.Expr on line 24 refers to that same method.

The further bind operation has the expected effect on super. The following code snippet is a redefinition of print() within EvalExpr.Plus not found in Listing 2.4:

```java
Plus ::= {
  ...
  void print() { printf("We are in EvalExpr.Plus");
  super.print(); }
  ...
}
```

Within EvalExpr.Plus, super refers to the inherited ExprFamily.Plus; the expression super.super is a reference to ExprFamily.Expr.

Our running expression example partially encoded in $\mathcal{VJ}$ is presented in Listing 2.6 (ignore the encoding of buildPlus for now). All classes in $\mathcal{VJ}$ are virtual; top-level classes (ExprFamily and buildPlus) are members of the root family.

20Conceptually speaking, super.super is not permitted Java syntax.
The keyword *outer* is used by an instance of a virtual class to access the instance of a containing object: the parameter allows navigation beyond the first level of containment. Notice that all references to virtual classes (within a family) are made via instances of a family object, for example, lines 8 and 9. As \( \mathcal{VJ} \) has no implicit scoping, the prefix `this.outer(ExprFamily)` is absolutely essential.

`outer()` has been overloaded to reference arbitrary superclasses and arbitrary levels of nested containment. This is somewhat counterintuitive and the left-hand-sides of the expressions on lines 18 and 20 will no doubt be confusing until this is realised. `this.outer(ExprFamily)` and `this.outer(buildPlus).e` are in fact two ways of referencing the same object.

Line 14 demonstrates that root is the immediate container for `ExprFamily` and `buildPlus` and further explains the need for static paths in the absence of implicit scoping.

`buildPlus` is an abstract virtual class (fields `e`, `e1` and `e2` are abstract). Indirectly, it is an encoding of the `buildPlus` method:\(^{22}\):
2. Background

e.Plus buildPlus(final ExprFamily e, e.Expr e1, e.Expr e2) {
    return new e.Plus(e1, e2);
}

\(V_J\) has no syntactic notion of methods: Anderson et al. encode buildPlus via a context-free encoding. The method is defined via the abstract virtual class on line 13. The concept of a return value is encoded by a field of the same type (line 23) whose expression is equivalent to the method body. Method arguments are encoded as abstract fields (lines 14–16).

Method call is simulated via instantiation of a concrete subclass of the abstract class. Argument passing is achieved by final binding the abstract fields in the aforementioned subclass. The return value is obtained by evaluating the result field.

2.6 Java

JAVA is recognised as a state-of-the-art object oriented programming language. It is therefore worthwhile considering the current and proposed mechanisms that (in)directly support collaboration based programming.

The first is inner classes. The second is generics in JAVA, a feature that, at the time of writing, has been completely adopted by Sun in a beta version of JDK 1.5\(^{23}\). A major influence on the adoption of generics in JAVA was the development of GJ; we consider the GJ modifications in this report.

2.6.1 Member Inner Classes

Inner classes in JAVA were not introduced to answer the needs of collaboration based programming, but nonetheless warrant some investigation given JAVA’s position in the market. JAVA’s event model demands the use of small listener and adapter classes for which inner classes are well suited\(^{54}\) but this does not preclude their use elsewhere.

Consider the JAVA code in Listing 2.7\(^{24}\) (class diagram can be seen in Figure 2.9). We say that class \(N\) is inner to class \(M\); it is lexically contained within the definition of \(M\). Similarly, class \(M\) is the enclosing class of \(N\). The same is true of \(O\)’s relationship to \(M\). An instance of \(N\), \(\xi^N\), has a link to an instance of \(M\), \(\xi^M\); we call the latter the host object. \(\xi^N\) can access \(\xi^M\) through the expression \(M\).this. \(N\) requires a host of class \(M\).

As our example shows, inner classes may extend and be extended by other classes with the caveat that any class extending an inner class ‘inherits’ host object requirements. In our example, \(O\) therefore requires two host objects of class \(M\):

1. by virtue of class \(O\)’s lexical position within \(M\)

2. because \(O\) extends \(N\) (another inner class) and therefore inherits \(N\)’s host requirements.

\(^{23}\)http://java.sun.com/j2se/1.5.0/index.jsp
\(^{24}\)Example due to [54]
```java
class M {
    int i;
    class N {
        int f() {
            return M.this.i;
        }
    }
    class O extends N {
        int g() {
            return M.this.i;
        }
    }
}
```

**Listing 2.7** — Example of inner classes

**Figure 2.9** — Class diagram for Listing 2.7

**Figure 2.10** — Inner classes can have multiple and distinct host objects. The links shown in the diagram are host requirements. Annotations to these links are the names of classes requiring the host.

There is no requirement that these two host objects be the same object as Figure 2.10 suggests.

Member inner classes convey existence dependence between the enclosing class and the inner class. In this way, they can be considered a mechanism for collaboration based programming. For example, an enclosing class `Graph` with inner classes `Node` and `Edge` represents our understanding that the existence of nodes and edges only makes sense
within the context of a graph.

The lexical positioning of inner classes has similarities to virtual classes contained in family classes. Furthermore, the current implementation of $VJ$ suggests the possibility of a virtual class having multiple outer (in the $VJ$ sense) objects. Reconsider the example presented in Listing 2.6. MyPlus can reference two outer classes (as suggested in Figure 2.11): buildPlus The lexical positioning of MyPlus makes this possible.

ExprFamily MyPlus inherits from Plus as defined within ExprFamily. The outer relationship between Plus and ExprFamily (due to lexical placement) is therefore inherited in much the same way as a host requirement in JAVA member inner classes.

As we suggest in Chapter 3 of [34], this similarity may help in the establishing of a clear model for super and outer in our virtual class calculus.

2.6.2 GJ and J2SE 1.5

GJ is the leading proposal, adopted by Sun,25 for generics support in the JAVA programming language. The end of 2003 saw Sun release a beta version of “Tiger” (J2SE 1.5), the next major revision to the Java platform and language, that included support for generic types.

The introduction of generics will not directly support collaboration based programming as gbeta and $VJ$ have sought to do. But instead they will provide JAVA programmers with power on a par with C++ templates. As such, GJ programmers will be able to implement mixin and mixin-layers as per C++ implementations.

In Chapter 6 of [44], Michalitis studies the suitability of generic types in expressing interlocked types using the Subject-Observer pattern. The interested reader should refer to this paper for a comparison of GJ and C++ templates as well as a thorough evaluation of the capabilities of generic types.

25http://www.sun.com/
Chapter 3

The Formal System

In this chapter, we introduce the formal system behind the calculus CONCORD\(^1\), developed in response to the specification laid down in [34]. In our presentation of the constituent parts of the formal system, we follow a defined format: we first present the formal system with accompanying explanation of symbology and terminology used; we then offer relevant examples in order to instill intuition and understanding of the formal method; finally, we motivate our design decisions in light of existing work and calculus goals. The components of our formal system include: syntax, our model of inheritance, execution, our type system and soundness. A running example is introduced in Section 3.1 to serve as a point of reference for the duration of this Chapter. Lemmata and Theorems referenced in this Chapter are to be found in Chapter 4.

3.1 Syntax

Figure 3.1 introduces the formal CONCORD syntax.

CONCORD introduces the notion of a group, an intuitive means of defining related concepts. Groups contain classes (classes in the standard Object-Oriented sense e.g. JAVA). As one might expect, classes contain field and method definitions. By defining two classes within the same group, we express our understanding that the two concepts (represented by classes) are in some way related; e.g. regular nodes and regular edges (the graph example of Chapter 1) belong together, just as coloured nodes and edges belong together. At a more abstract level, the group construct represents the context in which the two class concepts are related: e.g. graphs contain nodes and edges which, by themselves, perhaps have little meaning. We use \(g, g', \ldots\) as group identifiers and \(c, c', \ldots\) as class identifiers. \(m, m', \ldots\) and \(f, f', \ldots\) represent method and field identifiers respectively.

In order to bind grouped classes in some way, we introduce the notion of a relative type. Relative types consist of a reference to the current group, \(\text{MyGrp}\), and a class

\(^1\)For an amusing discussion on naming decisions, refer to Chapter 7
name (e.g. a \texttt{MyGrp.Node} reference was used as a field type inside \texttt{Graph.Edge}), thereby introducing \textit{intra-group} dependencies\textsuperscript{2}. Compare \textit{absolute} types: Absolute types consist of a group and a class name (cf. references to \texttt{JAVA} inner classes), thereby introducing \textit{inter-group} dependencies. Types therefore take the form \texttt{gr.c} where \texttt{gr}, \texttt{gr'}, \ldots range over group names (\texttt{g}, \texttt{g'}, \ldots) and the keyword \texttt{MyGrp}.

Groups may \textit{extend} other groups, giving rise to a \textit{supergroup} (the extended group) and \textit{subgroup} (the extending group); the ‘sub’ prefix mimics the terminology used for classes in object-oriented parlance. Subgroups are formally defined in Figure 3.2. Classes defined in the supergroup are “inherited” by the subgroup; we term this process \textit{further binding}\textsuperscript{3}. Further binding introduces implementation inheritance, a subject covered more closely in Section 3.3.

In a similar fashion to existing object-oriented languages (\texttt{JAVA}, \texttt{C++}), a class may \textit{extend} another class giving rise to a \textit{subclass} and \textit{superclass} (in line with existing understanding). Subclasses introduce inheritance but also create subtypes (cf. the further bind relationship). Superclasses may be both absolute and relative; each has different rules for inheritance of members. This involved inheritance procedure is addressed in Section 3.3.

Classes may both further bind and subclass other classes; hence \texttt{CONCORD} supports a limited form of multiple inheritance.

\textsuperscript{2}Thus, relative types support a \textit{restricted form} of dependent types, whereby a type may depend on a group — rather than on a value as in “full” dependent types.

\textsuperscript{3}Further binding is as per suggestions in e.g. \cite{49}
Every group is obliged to extend another group and every class obliged to extend another class; we therefore provide `GlobalGroup` and `GlobalGroup.c` at the top of the hierarchy. `GlobalGroup` is a special group that contains empty definitions for all classes c, c', .... Therefore, subclasses inherit nothing from `GlobalGroup.c`.

### 3.1.1 Example

We present our running example in Listing 3.1 (visualised in Figure 3.3); the code and diagram will be used as points of reference for the remainder of this Chapter. The program defines three groups: g1, g2 and g3 (we first concern ourselves with group g2). Within group g2, we define three classes: A, B and C.

Class A defines two fields, f1 and f2. He we have our first example of absolute and relative types: field f1 has the absolute type g2.A whereas f2 has a relative type MyGrp.A. The relative type has meaning because of its lexical containment within group g2. In the context of group g2, both g2.A and MyGrp.A refer to the same class, namely

---

4Reminder: groups and classes must always extend other groups and other classes (respectively). Therefore, in much the same as JAVA, we omit supergroup and superclass references in group and class definitions when those references are `GlobalGroup` or `GlobalGroup.c` respectively.
g2. A. In addition, class A defines two methods, m1 and m2. Method m1 demonstrates the use of relative types as return types and parameter types. As we shall see shortly, the relative types used in the definition of field f2 and method m1 refer to different classes when inherited into another group context via subclassing or further binding.

Class B is defined to be a relative subclass of MyGrp.A, the relative type again receiving meaning by its lexical containment within group g2. Contrast class C which is defined as an absolute subclass of class g2.A (requiring no context for meaning).

Listing 3.1 defines an additional two groups, g1 and g3. The definition of group g1 presents no surprises. Group g3, however, is defined to be a subgroup of g2: by Figure 3.2, we may conclude ⊢ g3 << g2. The classes A, B and C are further bound in the subgroup g3. Notably, in this example we choose to extend the inherited definition of C with addition of method m3 (the body of method m3 will be of interest in later sections).

We note, but only in passing, that the further binding of classes A, B and C gives the relative types in the respective definitions new context in group g3. This is covered more thoroughly in Section 3.3.

3.1.2 Motivation

We briefly mention points of note in relation to the Concord syntax:

- We do not require additional syntax to indicate further binding. Further binding can be inferred by the presence of a subgroup relationship. For example, as group g3 extends group g2, we can infer that the definitions of classes A, B and C are further bound in the subgroup g3.

- We oblige all classes and groups to extend other classes and groups respectively. Arguably, class or group extension should be optional (and via provision of the special GlobalGroup we achieve this effect). However, enforcing this requirement

\footnote{Further bound classes are omitted from subgroups in diagrams except in the case where they extend or redefine the inherited definition.}
Unique definitions, acyclic groups and acyclic types

via the syntax makes the remainder of the formal system (and subsequent proofs) cleaner and clearer.

3.2 Unique definitions, acyclic groups and acyclic types

The concepts of unique definitions, acyclic groups and acyclic types are simple ones. We therefore omit example and motivation subsections for this part of the formal system.

Our requirement for unique definitions is presented in Figure 3.4. We summarise the formal system via the following bullets:

- Group names in a program are unique
- Member class names are unique within a group
- Groups cannot be nested\(^6\), hence there can be no naming clash
- Field and method names are unique within member classes, even where subclassing and further binding is involved (we shall address this issue more thoroughly in Section 3.4)

Our requirement for acyclic types and groups is also presented in Figure 3.4. We simply require that there be no cycles of extension at the group or class level.

3.3 Inheritance

As we said earlier, both further bound classes and subclasses introduce inheritance but in slightly different ways. The difference between the two inheritance models may be summarised: further binding preserves intra-group dependencies (i.e. preserves relative types), whereas subclassing sometimes (we shall see an exception later) replaces intra-group dependencies by inter-group dependencies (i.e. replaces relative types by absolute types).

All classes from a certain group are further bound in a subgroup. For example, classes g2.A and g2.B are further bound in group g3 (as g3.A and g3.B respectively). Furthermore, not only does g3.C further bind g2.C, it is also a subclass of g1.D. Thus, in CONCORD a class c, defined via \texttt{group g \ll g'} \{ \ldots \texttt{class c <: \texttt{gr.c'}} \{ \ldots \} \ldots \}, further binds g'.c (if such a class exists), and is a subclass of gr.c'. Therefore, g.c inherits from both g'.c and gr.c'. The exact process is described via the definitions of functions \(\mathcal{F}\) and \(\mathcal{M}\) in Figure 3.6; the operator \(\oplus\) is defined in Appendix A.1.

Before assessing the intricacies of field and method inheritance, we first turn our attention to the group and class lookup functions defined in Figure 3.5.

\(^6\)The lack of group nesting follows from the syntax
3. The Formal System

The group lookup function $G$ (defined in rule $(G_1)$) is fairly self explanatory — if a group $g$ is defined within a program its definition is returned, else the result is undefined (abbreviated $\textit{Undef}$). By rule $(C_1)$, $CW(g, c)^7$ returns the definition of class $c$ as defined within group $g$ if such a definition exists, else is undefined. Rule $(C_2)$ expresses our understanding that $\textit{GlobalGroup}$ contains undefined definitions of all classes $c$. Rules $(C_3)$ and $(C_4)$ express our understanding of further binding in subgroups: given $C(g, c)$, if a definition of class $c$ exists within group $g$, this is returned, else our search continues in supergroups $g'$ of $g$.

We now return to issues of field and method inheritance.

In a similar fashion to $CW$, functions $\mathcal{M}W(g, c, m)$ and $\mathcal{F}W(g, c, f)$ return field and method definitions (respectively) if such definitions exist within class $g, c$, else are un-

---

7The $W$ in $CW$ stands for “worker”
Inheritance

\[
P = P' \text{ group } g << g' \{ gbody \} P''
\]

\[
G(g) = \text{ group } g << g' \{ gbody \} (G_1)
\]

\[
G(g) = \text{ group } g << g' \{ \ldots \text{class } c : t \{ cbody \} \ldots \} (C_1)
\]

\[
CW(g.c) = \text{ class } c : t \{ cbody \}
\]

\[
CW(GlobalGroup.c) = \text{ Udf} (C_2) \quad \frac{CW(g.c) \neq \text{ Udf}}{C(g.c) = CW(g.c)} (C_3)
\]

\[
CW(g.c) = \text{ Udf}
\]

\[
G(g) = \text{ group } g << g' \{ \ldots \} (C_4)
\]

\[
C(g.c) = C(g'.c)
\]

**Figure 3.5** — Group and class lookup functions.

defined (seen in rules \((M_1)\) and \((F_1)\)).

Our understanding of \textit{GlobalGroup} is further refined via rules \((M_2)\) and \((F_2)\): no methods or fields can be defined in the empty classes of \textit{GlobalGroup}.

Rules \((M_3)\) and \((F_3)\) are sufficiently similar in behaviour that they be treated together\(^8\). The lookup \(F(g.c,f)\) first considers the definitions contained in class \(g.c\) (if such a class exists). If no such definition exists, we then consider definitions inherited from the superclass (if such a class exists) or the class further bound from the supergroup (again, if such a class exists). We now consider inheritance through further binding and subclassing.

Through further binding, \(g.c\) inherits all fields and methods from \(g'.c\) unmodified, i.e. \(F(g.c,f) = \ldots + \ldots + F(g'.c,f)\).

Inheritance though subclassing is more intricate. Firstly, recall we may define a class to have a relative superclass. e.g.: \textit{group } g << g' \{ \ldots \text{class } c : \text{MyGrp}.c' \{ \ldots \} \ldots \}. Looking up a field in \textit{MyGrp}.c' is meaningless: we have no context in which to resolve the \textit{MyGrp} reference. Hence, the superclass is first determined by substituting\(^9\) any intra-group reference in \(gr.c'\) with the current group, \(g\) via \(gr[g]\). Our lookup will then be well defined, but may of course return undefined if no such definition exists in our superclass\(^10\). Then, any intra-group references in the returned member definition are replaced by the reference to the supergroup, \(gr: F(g.c,f) = \ldots + F(gr[g].c',f)[gr] + \ldots\)

\(^8\)Even though we only consider field lookup, it is clear that method lookup would proceed by a similar argument

\(^9\)A definition of substitution may be found in Section 3.6

\(^{10}\)Notice the definitions of \(M\) and \(F\) are recursive. Hence when questions are asked of a superclass (or class further bound from the supergroup) the lookup is again recursive.
3. The Formal System

\[ CW(g.c) = \text{class } c :: \ldots \{ \ldots t \text{ m}(t_1 \ x) \{ e \} \ldots \} \]

\[ MW(g.c,m) = t \text{ m}(t_1 \ x) \{ e \} \quad (M_1) \]

\[ MW(\text{GlobalGroup}.c,m) = U df \quad (M_2) \]

\[ G(g) = \text{group } g \ll g' \{ \ldots \} \]
\[ C(g.c) = \text{class } c :: \text{gr}.c' \{ \ldots \} \]

\[ M(g.c,m) = MW(g.c,m) \oplus M(\text{gr}[g'.c',m][\text{gr}] \oplus M(g'.c,m) \quad (M_3) \]

\[ CW(g.c) = \text{class } c :: \ldots \{ \ldots t \text{ f} \ldots \} \]
\[ FW(g.c,f) = t \quad (F_1) \]
\[ FW(\text{GlobalGroup}.c,f) = U df \quad (F_2) \]

\[ G(g) = \text{group } g \ll g' \{ \ldots \} \]
\[ C(g.c) = \text{class } c :: \text{gr}.c' \{ \ldots \} \]
\[ F(g.c,f) = FW(g.c,f) \oplus FW([\text{gr}[g'.c',f][\text{gr}] \oplus FW(g'.c,f) \quad (F_3) \]

\[ F_s(g.c) = \{ f \mid FW(g.c,f) \neq U df \} \quad (F_4) \]

Figure 3.6 — Method and field lookup functions

... This final substitution deserves further comment. We notice that if \( g.c \) has a relative superclass, the substitution of relative types in the returned definition has no effect. Hence, we may refine our earlier précis of inheritance as follows:

Further bound classes and subclasses both introduce inheritance. In further bound classes and relative subclasses, fields and methods are inherited as defined. In absolute subclasses, the definitions of inherited fields and methods defined in terms of relative types are subject to substitution such that references to \text{MyGrp} are replaced by the supergroup name.

We conclude our discussion of the formal model for inheritance by considering rule \((F_4)\). The function \( F_s \) simply returns the set of fields defined in class \( g.c \).

3.3.1 Example

Let us return to our example. We have already identified that classes \( A, B \) and \( C \) are further bound in group \( g3 \). We have also identified class \( B \) as a relative subclass of class \( A \) in the context of group \( g2 \). Furthermore, as group \( g3 \) is a subgroup of group \( g2 \), class \( B \) is a relative subclass of class \( A \) in the context of group \( g3 \) (this is formalised in our definition of subtypes in Section 3.6). With reference to our earlier précis of inheritance, we demonstrate field and method inheritance via the following table:
3.3.2 Motivation

Undoubtedly, the foremost question relating to inheritance is that of MyGrp substitution in the case of absolute subclasses. Why do we need to replace instances of MyGrp when inheriting from an absolute superclass? The answer is perhaps best seen by reference to our example. We reproduce the pertinent snippet of code for ease of reference:

```java
class D {  
    MyGrp.D f3  
}

...  

class C <: g1.D {  
    ...  
}
```

Class g1.D is an absolute superclass of g3.C; in a system without substitution of inherited members, what is the type of field f3 in class g3.C? If inherited without substitution, the type is MyGrp.D. Remembering we have asked this question of field f3 in the context of group g3, we check whether this relative type has meaning, i.e. does class g3.D exist? Unfortunately, in our example the answer is no, hence we have a problem. But, in other cases we may be lucky. Therefore, in the general case of absolute inheritance, we can say nothing about whether relative types in the definitions of fields and methods in the superclass will have well defined meaning in the context of the subclass’s containing group, i.e. we cannot infer any (subgroup) relationship between the superclass group
3. The Formal System

and the subclass group. Hence, we substitute to be safe. In summary, the interface of relative classes presented in the contexts of the subclass group and the superclass group are not guaranteed to coincide.

There is one notable situation where the two interfaces do coincide, namely when the absolute subclass is defined within the same group as (or a subgroup of) the absolute superclass. If, in this restricted case, we did not substitute relative types, what would be the difference between an absolute subclass and a relative subclass? The answer is seen immediately when we consider extension (via further binding) of the superclass in a subgroup of its defining group. In the case of a relative subclass, we expect a further bound subclass to inherit additional members defined in subsequent further bound extensions of the superclass; whereas, in the case of an absolute subclass, we expect the inheritance to be anchored to the named class.

Similarly, we might ask why no substitution is required in the case of relative subclasses. A relative subclass, by definition, belongs to the same group as its relative superclass. Hence, the interface of relative classes in the context of the defining group presented to both subclass and superclass will be exactly the same; furthermore, this condition will persist in every subgroup of the defining group (even if the group definition is extended and either subclass or superclass are extended).

Given the definition of $\oplus$ (Appendix A.1), one might reasonably ask whether the order of functions on the right hand side of $M$ and $F$ definitions is significant. i.e. would we get a different answer if the order was altered? In Section 3.4, we introduce well-formedness conditions that ensure inherited methods and fields will be nominally unique at every point of inheritance, whether inherited through furthering binding or subclassing. Hence, beyond ensuring that $MW$ is first consulted in the case of method lookup (in order that we receive the most specific redefinition of a method), order is irrelevant.

3.4 Well-formedness of programs, groups and classes

As is typical of formal systems, we demand well formedness of programs, itself a recursive requirement which, in the case of CONCORD, demands well formed groups and correspondingly classes. Our requirements are presented in Figure 3.7.

The first judgement, \[ g . c \vdash m \diamond \], demands that method definitions inside classes be well formed. For method definitions inside the class \( g . c \), every type must be well formed in the context of group \( g \) (remember we may have relative types). The type of the method body, in the context of the environment \( t_1 \ x . g . c \ this \), must be a subtype, in the context of group \( g \), of the return type specified in the method signature: these two requirements will become clear in the light of Section 3.6. Judgement $\vdash g . c \diamond_{general}$ encompasses the first judgement and additionally requires that fields defined within a class \( g . c \) have well formed types in the context of group \( g \).

The first two lines of both $\vdash g . c \diamond_f$ and $\vdash g . c \diamond_{inh}$ share a common requirement: as is standard in many object oriented languages (access modifiers granted), redefinition of a field in a subclass or further bound class is forbidden. For further bound classes,
Well-formedness of programs, groups and classes

\[\mathcal{MW}(g, c, m) = t \cdot m(t_1 \cdot x) \{ e \}\]
\[g \vdash t \circ t\]
\[g \vdash t_1 \circ t\]
\[t_1 \cdot x, g \vdash c \text{ this } \vdash e : t_{\text{ret}}\]
\[g \vdash t_{\text{ret}} < : t\]
\[g \vdash m \diamond\]
\[\mathcal{FW}(g, c, f) = t_0 \implies g \vdash t_0 \circ t\]
\[\mathcal{MW}(g, c, m) = t \cdot m(t_1 \cdot x) \{ e \} \implies g \vdash m \diamond\]

\[\vdash g \cdot c \diamond_{\text{general}}\]

\[G(g) = \text{group } g \ll g' \{ \ldots \text{ and } C(g, c) = \text{class } c <: gr \cdot c' \{ \ldots \} \implies \]
\[\mathcal{FW}(g, c, f) \neq \mathcal{ULf} \implies \mathcal{F}(g', c, f) = \mathcal{ULf}\]
\[\mathcal{MW}(g, c, m) = t \cdot m(t_1 \cdot x) \{ \ldots \} \text{ and } \mathcal{M}(g', c, m) \neq \mathcal{ULf} \implies \]
\[t = t'[gr] \text{ and } t_1 = t'_1[gr]\]

\[\vdash g \cdot c \diamond_{\text{fb}}\]

\[C(g, c) = \text{class } c <: gr \cdot c' \{ \ldots \} \text{ and } g' \cdot c = gr[g] \cdot c \implies \]
\[\mathcal{FW}(g, c, f) \neq \mathcal{ULf} \implies \mathcal{F}(g', c', f) = \mathcal{ULf}\]
\[\mathcal{MW}(g, c, m) = t \cdot m(t_1 \cdot x) \{ \ldots \} \text{ and } \mathcal{M}(g', c, m) = t' \cdot m(t_1' \cdot x) \{ \ldots \} \implies \]
\[t = t'[gr] \text{ and } t_1 = t'_1[gr]\]

\[\vdash g \cdot c \diamond_{\text{inh}}\]

\[G(g) = \text{group } g \ll g' \{ \ldots \}
C(g, c) = \text{class } c <: gr \cdot c' \{ \ldots \}
\mathcal{F}(g', c, f) \neq \mathcal{ULf} \implies \mathcal{F}(gr[g] \cdot c', f) = \mathcal{ULf}\]
\[\mathcal{M}(g', c, m) \neq \mathcal{ULf} \implies \mathcal{M}(gr[g] \cdot c', m) = \mathcal{ULf}\]

\[\vdash g \cdot c \diamond_{\text{uniq}}\]

\[\vdash P \diamond_{ag} \vdash P \diamond_{ac} \vdash g \cdot c \diamond_{\text{general}} \vdash g \cdot c \diamond_{\text{fb}} \vdash g \cdot c \diamond_{\text{inh}} \vdash g \cdot c \diamond_{\text{uniq}}\]

\[\vdash g \cdot c\]

\[G(g) = \text{group } g \ll g' \{ \ldots \} \text{ and } C(g', c) = \text{class } c <: MyGrp \cdot c' \{ \ldots \} \implies \]
\[C(g, c) = \text{class } c <: MyGrp \cdot c' \{ \ldots \} \implies \]
\[\vdash g \cdot c \diamond_{\text{restrict}}\]

\[\forall c: C(g, c) = \text{class } c <: gr \cdot c' \{ \ldots \} \implies \vdash g \cdot c\]

\[\vdash g \cdot c \diamond_{\text{restrict}}\]

\[\vdash g\]

\[\forall g: G(g) \neq \mathcal{ULf} \implies \vdash g\]

\[\vdash P \diamond\]

Figure 3.7 — Well formed classes, groups and programs
we further require that any redefinition of a method \( m \) uses a signature identical to that found in supergroup class definition. A similar requirement exists for subclasses, except that in the case of absolute subclasses, the method signature is identical modulo substitution of intra-group references (relative types): this is in line with our inheritance model (Section 3.3).

Judgement \( \vdash g . c \Diamond_{\text{uniq}} \) ensures that the set of field and method names inherited via further binding is disjoint from the set of field and method names inherited via subclassing (hence our earlier discussion about the order of functions in the definitions of \( M \) and \( F \) being irrelevant (rules \((M_3)\) and \((F_3))\)).

Judgement \( \vdash g . c \Diamond_{\text{restrict}} \) was included late in the development of the CONCORD calculus. It relates to classes defined as relative subclasses in a given group \( g \), for example group \( g << \ldots \{ \ldots \text{class} \ c :: \text{MyGrp.c'} \{ \ldots \} \ldots \} \). If, in any subgroup \( g' \) of \( g \), we chose to further bind class \( c \), our definition must take the form \( \text{class} \ c :: \text{MyGrp.c'} \{ \ldots \} \), i.e. any further binding of a relative subclass cannot specify extension of another class \( g . c'' \), relative or absolute. A motivating example is presented in Subsection 3.4.2.

Well formed classes, \( \vdash g . c \Diamond \) satisfy every judgement discussed thus far. Well formed groups, \( \vdash g \Diamond \), contain only well formed classes. Well formed programs, \( \vdash P \Diamond \), contain only well formed groups.

### 3.4.1 Example
Consider the situation in Listing 3.2.

**Case 1** Class \( \text{A} \) is further bound in group \( g10 \) because \( \vdash g10 << g9 \). Therefore to redefine \( \text{A} \) by extending its original definition in the supergroup \( g9 \) is nonsensical and would lead to a violation of the \( \vdash g . c \Diamond_{\text{uniq}} \) judgement.

**Case 2** Acceptable, but perhaps not what the author intended. Perhaps he/she intended class \( \text{B} \) to extend \( \text{MyGrp.A} \)?

**Case 3** In addition to further binding class \( \text{A} \), we introduce multiple inheritance by defining class \( g10 . \text{A} \) to extend \( g12 . \text{D} \). Field and methods names inherited via further binding and subclassing are unique.

**Case 4** This is the simple case of further binding class \( \text{A} \) is the subgroup \( g10 \) where redefinition does not introduce multiple inheritance.

**Case 5** Again, a class effectively inheriting definitions from itself. This is nonsensical and, by reasoning similar to that for case 1, violates the \( \vdash g . c \Diamond_{\text{uniq}} \) judgement.

### 3.4.2 Motivation
The restrictions imposed by the first four judgements of Figure 3.7 should come as no surprise to JAVA programmers. The restrictions on field and method name uniqueness, in the case of multiple inheritance, exist in order that our proofs and formal system may be more concise (and therefore clearer).
Well-formedness of programs, groups and classes

```java
Listing 3.2 — Exploring the definition of well-formedness

group g9 {
    class A { ... }
}
group g10 ::= g9 {
    // case 1
    class A ::= g9.A { ... }
    // case 2
    class B ::= g9.A { ... }
    // case 3
    class A ::= g12.D { ... }
    // case 4
    class A { ... }
    // case 4
    class A ::< MyGrp.A { ... }
}
```

```java
Listing 3.3 — Subtleties in method redefinition

group ga {
    class A {
        MyGrp.A m(MyGrp.A) { ... }
    }
    class B ::= MyGrp.A {
        // initial constraints require
        ga.A m(ga.A) { ... }
        // intuition (and current version) requires
        MyGrp.A m(MyGrp.A) { ... }
    }
    class C ::= ga.A {
        ga.A m(ga.A) { ... }
    }
}
group gb {
    class D ::= ga.A {
        ga.A m(ga.A) { ... }
    }
}
```
However, the judgement \( \vdash_{\text{inh}} \varphi \) is subtle in one respect. Our initial version of the judgement required that \( t = t'([g']) \) and \( t_1 = t'_1([g']) \). Prima facie, there is no difference between the requirements imposed by two sets of constraints. Consider, however, the code example presented in Listing 3.3.

We first emphasise that, in the case of absolute subclasses (lines 12 and 17), we unquestionably must redefine method \( m \) via the method signature of the superclass \((\text{MyGrp.A m(\text{MyGrp.A})})\) modulo substitution of relative types by group names\(^\text{11}\) (hence giving \( \text{ga.A m(\text{ga.A})} \) on lines 13 and 18): thankfully, both versions of the constraints treat absolute subclasses in the same way.

Where relative subclasses are concerned, the different constraints place different requirements on the signature in the redefinition. The initial constraints demand that redefinition of method \( m \) in a relative subclass replace relative types in the method signature in exactly the same manner as we saw for absolute subclasses (hence line 7). But this requirement, whilst safe, is counterintuitive given that class \( B \) is a relative subclass of class \( A \) in the context of group \( \text{ga} \). Our intuition demands redefinition via the signature shown in line 10\(^\text{12}\).

Hence, to cater for the requirements of both absolute and relative subclasses, method redefinition occurs under the signature of the method as defined in the superclass \((\text{MyGrp.A m(\text{MyGrp.A})})\) modulo substitution via the superclass reference found in the definition of the subclass. For relative subclasses, the superclass reference is \( \text{MyGrp} \); hence, by the definition of substitution in Section 3.6, no substitution occurs in the method signature.

The judgement \( \vdash_{\text{restrict}} \varphi \) requires special consideration. Why do we need to restrict the redefinition of relative subclasses in subgroups? Consider the example presented in Listing 3.4 (with reference to Figure 3.6). In group \( g_4 \), we define class \( A \) to be a relative subclass of \( B \) (line 2). Therefore, given \( \vdash g_5 << g_4 \) and \( \vdash g_6 << g_5 \), we expect the lookup of field \( f \) in class \( g_6.A \), \( F(g_6.A,f) \), to return the value \( \text{int} \). However, analysing the definition of \( F \), this turns out not to be the case.

The problem arises because our field and method lookup functions do not consider relative subclassing. Relative subclassing not only has meaning in the defining group (in this case \( g_4 \)), it also has meaning in every subgroup \((g_5, g_6, ...)\)\(^\text{13}\). Contrast absolute subclassing: the absolute superclass is anchored and does not refer to a different class in the context of a subgroup. The corner case shown in Listing 3.4 demonstrates how redefining the further bound class \( A \) to extend \( g_7.D \) (line 8), thereby introducing multiple inheritance, masks the relative definition of line 2: members defined via extension of class \( B \) in subgroups of \( g_4 \) are therefore hidden from class \( A \) (in context of these subgroups). Hence, contrary to our intuition, we receive an undefined result: \( F(g_6.A,f) = \text{undefined} \).

Notice, however, that we are careful to only consider relative subclasses in this restriction. The absolute superclass \( g_7.D \) always refers to the class \( g_7.D \), regardless of context. Hence, extension of \( g_7.D \) via further binding in subgroups of group \( g_7 \) will

\(^\text{11}\)See Subsection 3.3.2 for a thorough explanation of this argument

\(^\text{12}\)This covariance in a method’s return and parameter type ensures we demand the most specific type of an object in further bound definitions in subgroups

\(^\text{13}\)This is formalised in Section 3.6
(and should) never be considered when assessing which members are inherited by class A in the context of group g5 and its subgroups.

3.5 Execution

We define execution in terms of large step semantics whereby an expression and store are mapped onto a value and store. The store, \( \sigma \), represents the stack and heap. It maps \this\ and the method parameter \( x \) onto addresses and addresses onto objects. Objects \( [g.c \parallel f_i : v_i] \) contain their runtime type (or absolute type, \( t_a \)) and the values of their fields. This can be seen in Figure 3.8.

In order to describe execution, we need a way to obtain the group to which the current receiver belongs. We define the function:

\[
\mathcal{M}_{\text{MyGrp}}(\sigma) = g \text{ where } \sigma(\this) = [g.c \parallel \ldots]
\]

In Figure 3.9 we define the operational semantics. All rules are straightforward and similar to those in [15] (hence we omit example and motivation subsections), with the exception of the rule for object creation. The runtime type of the object being created may depend on the runtime type of the current receiver. For example, with a receiver of runtime type \( g3.\text{A} \), execution of \( \text{new MyGrp.C} \) will create an object of dynamic type \( g3.\text{C} \).
### 3. The Formal System

<table>
<thead>
<tr>
<th>stack</th>
<th>heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>store = {this \mapsto \text{addr}} \cup {x \mapsto \text{addr}} \cup {\text{addr} \mapsto \text{object}}</td>
<td></td>
</tr>
<tr>
<td>val = {\text{null}} \cup \text{addr}</td>
<td></td>
</tr>
<tr>
<td>dev = {\text{nullPtrExc}}</td>
<td></td>
</tr>
<tr>
<td>object = {\text{g.c} \parallel f_i : v_i^{1\ldots n} \mid f_i, g, c \text{ identifiers}, v_i \in \text{val}}</td>
<td>{\text{addr} \mapsto \text{object}}</td>
</tr>
<tr>
<td>addr = {i_i \mid i \in \mathbb{Z}^\star}</td>
<td></td>
</tr>
</tbody>
</table>

\[ o(f) = \begin{cases} v_i & \text{if } f = f_i | l \in 1, \ldots, r \\ \text{Undef} & \text{otherwise} \end{cases} \]

\[ o[f \mapsto v] = [g.c \parallel f : v_1 \ldots f_l : v_l \ldots f_r : v_r] \] if \( \exists l \in 1, \ldots, r \mid f = f_i \)

\[ \sigma[z \mapsto v](z) = v \]

\[ \sigma[z \mapsto v](z') = \sigma(z') \] if \( z' \neq z \)

---

**Figure 3.8** — Stores of and operations on objects \( o \); store \( \sigma \) and identifier or address \( z \).

---

<table>
<thead>
<tr>
<th>val</th>
<th>var</th>
</tr>
</thead>
<tbody>
<tr>
<td>( v, \sigma \leadsto v, \sigma )</td>
<td>( x, \sigma \leadsto \sigma(x), \sigma )</td>
</tr>
<tr>
<td>( e, \sigma \leadsto i, \sigma'' )</td>
<td>( \text{this}, \sigma \leadsto \sigma(\text{this}), \sigma )</td>
</tr>
<tr>
<td>( e', \sigma'' \leadsto v, \sigma''' )</td>
<td></td>
</tr>
<tr>
<td>( \sigma''''(i)(f) \neq \text{Undef} )</td>
<td></td>
</tr>
</tbody>
</table>
| \( e.f = e', \sigma \leadsto v, \sigma' \) | \( \text{(fldAss)} \)
| \( \sigma' = \sigma''''(i)[f \mapsto v] \) | \( \text{e.f}, \sigma \leadsto \sigma(i)(f), \sigma'' \) |

\[ g = g.r[M\text{MyGrp}(\sigma)] \]

\[ F_s(g.c) = \{f_1, \ldots, f_r\} \]

\[ \text{new gr.c, } \sigma \leadsto i, \sigma[i \mapsto [g.c \parallel f_1 : \text{null}, \ldots, f_r : \text{null}]] \]

\[ e_0, \sigma \leadsto i, \sigma_0 \]

\[ e_1, \sigma_0 \leadsto v_1, \sigma_1 \]

\[ \sigma_1(i) = [t a \parallel \ldots] \]

\[ M(ta,m) = t \cdot m(t_1, x) \{ e \} \]

\[ \sigma'' = \sigma_1[\text{this} \mapsto i][x \mapsto v_1] \]

\[ e, \sigma'' \leadsto v, \sigma' \]

\[ e_0.m(e_1), \sigma \leadsto v, \sigma'[\text{this} \mapsto \sigma(\text{this}), x \mapsto \sigma(x)] \]

**Figure 3.9** — Operational semantics. We omit rules for throwing and propagation of exceptions; they are standard.
Subtypes and the Type System

We roughly follow the Java approach, whereby subclasses (both relative and absolute), which introduce inheritance, introduce subtypes. However, with relative subclasses the situation is complicated by the presence of relative types in the declaration of the class. In the definition group g << ... { ... class c <: MyGrp.c' { ... } ... }, the reference MyGrp.c' only has meaning by virtue of the context supplied by the containing group. Hence, the meaning of the subtype relationship is context dependent. Figure 3.10 defines the subtype relationship g ⊢ t <: t', whereby a type t is a subtype of another type t' in the context of a specific group g. Further binding does not introduce subtypes (even though it introduces inheritance); we motivate the reasons behind this decision in Subsection 3.6.2.

Rule (ST1) presents no surprises. Rule (ST2) conveys a powerful motive. The judgement g ⊢ t <: t' may refer to either relative or absolute types. In the case that both t and t' are absolute, we notice the substitutions t[g] and t'[g] have no effect, such that t[g] = t and t'[g] = t'. Furthermore, we can identify that the context, g, has no bearing on the subtype relationship (see rule (ST5)). Hence, we may conclude that the subtype relationship between these two absolute types holds in any context. However, when t and or t' are relative types, the context does have a bearing on the subtype relationship: in fact, it is crucial to its existence (because in another context either of the relative types may not be well defined). Where relative types are concerned, the substitutions t[g] and t'[g] have an effect. The effect is to “convert” our relative subtype relationship into an absolute subtype relationship (requiring no context); hence, we are permitted to choose arbitrary context g'. This conversion process allows subtype relationships to embrace meaning in any context (modulo necessary substitutions as just described). Rule (ST3) is interesting when either t and t' are relative or t alone is relative14. In either case, the rule expresses our intuitional understanding that relative

14Lemma 2 expresses a subtlety in subtype judgements, namely that judgements of the form g' ⊢ g.c <: MyGrp.c' do not exist.
3. The Formal System

subtype relationships adopt new meaning (and context) in subgroups. Rule (ST4) is a standard transitivity extension. Rule (ST5) expresses our understanding the absolute subtype relationships require no context for meaning.

We can prove that any subtype relationship satisfied in the context of a group is also satisfied in the context of its subgroups. More formally, for any \( g, g' \) with \( g \vdash g' < < g \):

\[
\Gamma \vdash t' <: t \implies g \vdash t' <: t.
\]

This property is expressed in Lemma 4.

We now move on to discussion of the type system.

Expressions are typed in the context of an environment, \( \Gamma \). The environment assigns types to the receiver, \( \text{this} \), and the method parameter, \( x \). \( \Gamma(id) \) returns the type of \( id \) in \( \Gamma \) and is defined as follows:

\[
\text{For } \Gamma = t \ x, g, c \text{ this: } \quad \Gamma(id) = \begin{cases} 
    t & \text{if } id = x \\
    g & \text{if } id = \text{MyGrp} \\
    \text{MyGrp}.c & \text{if } id = \text{this} \\
    \text{undefined} & \text{otherwise}
\end{cases}
\]

Notice that \( \Gamma(\text{this}) \) always has the form \( \text{MyGrp}.c \).

Well formed environments and types are defined in Figure 3.11. The definition of well formed types should come as no surprise, however the rules defining \( \Gamma \vdash t \diamond t \) and \( g \vdash t \diamond t \) warrant mention. The two judgements are defined for convenience in the broader definition of our formal system. Of course, we could optionally drop the judgements in favour of a more succinct formal system, relying on judgements of the form \( \vdash t|g| \diamond t \). However, the shape of the existing form of the judgement, with the left hand side providing context for the types in the right hand side, more often than not expresses our intuitional understanding of the definition, for example in well formedness rule defining \( g.c \vdash m \diamond \). The definition of well formed environments, \( \vdash \Gamma \diamond \), follows.
immediately, requiring that the types of the receiver, \texttt{this}, and method parameter, \texttt{x}, are well formed. Judgements involving \texttt{GlobalGroup} further refine our understanding of these global concepts developed in Section 3.1.

We now define two functions required for the definition of our type system.

We define the function \( \texttt{MyGrp}(s) \) to extract the group from the subject \( s \), where \( s \) ranges over types \( t \), stores \( \sigma \) and environments \( \Gamma \):

\[
\begin{align*}
\texttt{MyGrp}(\sigma) &= g \text{ where } \sigma(\texttt{this}) = [ g. c || \ldots ] \\
\texttt{MyGrp}(\Gamma) &= \Gamma(\texttt{MyGrp}) \\
\texttt{MyGrp}(t) &= g.r \text{ where } t = g.r.c
\end{align*}
\]

The substitution function, \( s[r] \), replaces occurrences of the \texttt{MyGrp} reference in the subject \( s \) by the group \( g \) of \( r \). It is defined in the following way:

\[
\begin{align*}
e[g] &= e[g/\texttt{MyGrp}] \\
t[\Gamma] &= t[\texttt{MyGrp}(\Gamma)/\texttt{MyGrp}] \\
t[\sigma] &= t[\texttt{MyGrp}(\sigma)/\texttt{MyGrp}] \\
t[t'] &= t[\texttt{MyGrp}(t')/\texttt{MyGrp}] \\
(t \cdot (t' x)\lbrace e \rbrace)[g] &= t[g] \cdot (t'[g] x)\lbrace e[g] \rbrace
\end{align*}
\]

We now return to the definition of our type system.

Figure 3.12 defines the type rules \( \Gamma \vdash e : t \), whereby an expression \( e \) has type \( t \) in the context of an environment \( \Gamma \). The rules \((\text{NewNull})_T\) and \((\text{Subsump})_T\) are standard.

\((\text{Meth})_T\) is more interesting for two reasons. Firstly, we notice that type \( t \) may be relative. For our field lookup to be meaningful, we must resolve this reference using the
context provided by the environment, $\Gamma$, to an absolute type. Hence, we lookup field $f$ in class $t[\Gamma]$ (through $t' = F(t[\Gamma], f)$), the absolute type found by replacing occurrences of $\text{MyGrp}$ in $t$ by the group containing the current method definition. Secondly, and perhaps most confusingly, the expression $e.f$ is typed $t'[t]$. We turn our attention as to why this should be.

Recall that the target of the field access, $e$, may have a relative type in the context of the environment $\Gamma$. Similarly, the target field may be defined via a relative or absolute type. If the target field, $f$, is defined with an absolute type, $t'$, then intuition demands this be the type of the field access: we are assured of this conclusion by again noting that substitution in an absolute type has no effect. If, however, the target field, $f$, is defined with a relative type we must consider the type of $e$, namely $t = gr.c$. In order to make the type of the field access “relevant” to the target of the field access, we substitute occurrences of $\text{MyGrp}$ in the field type $t'$ by the group reference in the type of $e$, namely $gr$. Hence, for a field defined via a relative type, field access has a relative type if the target of the field access has a relative type and an absolute type otherwise.

The first conclusion of $(\text{VarThis})_T$ is unsurprising. However, with reference to the definition of $\Gamma()$, we notice that the type of $\text{this}$ is always of the form $\text{MyGrp}.c$. This requirement is essential for the correct typing of access to relative fields (covered in discussion of $(\text{Fld})_T$).

The formats of the remaining two rules, $(\text{Meth})_T$ and $(\text{FldAss})_T$, are reasoned by arguments similar to those presented for $(\text{Fld})_T$; hence these two rules require no further mention.

We can prove that any absolute type, $g'.c'$, inherits every member from its supertype $g.c$; the members are inherited unmodified if the two groups $g$ and $g'$ are equal, otherwise occurrences of $\text{MyGrp}$ are replaced by $g$. This property is expressed in Lemma 6. A more general presentation of this property is expressed in Lemma 5.

We can also prove that expressions preserve their types when typed in a subgroup. Therefore, inheritance of methods through further binding preserves the method body type. This property is expressed in Lemma 8.

Furthermore, we can prove that, given $\Gamma \vdash e : t$, if we replace the type of the receiver in the environment $\Gamma$ with a subtype (to give $\Gamma'$), and substitute occurrences of $\text{MyGrp}$ in an expression $e$ by $\text{MyGrp}(\Gamma)$ (to give $e[\Gamma]$), then, in the context of the environment $\Gamma'$, the expression $e[\Gamma']$ has type $t[\Gamma']$, the type obtained by replacing all occurrences of $\text{MyGrp}$ with $\text{MyGrp}(\Gamma)$ in $t$. Therefore, inheritance of methods by subclasses preserves method body types, modulo the necessary substitutions of $\text{MyGrp}$. This property is expressed in Lemma 9.

Thus, we can prove that in a well formed program (well formed programs, $\vdash P \Diamond$, are described in Section 3.4), the body of any method in an absolute type $ta$, whether inherited or defined in $ta$ itself, has a type in accordance with its type in $ta$. This property is expressed in Theorem 1.
3.6.1 Example

The examples within this Section are numerous. Hence, we first address examples of subtypes and then examples of typings.

3.6.1.1 Subtypes

Our subtype examples are neatly summarised in the following table:

<table>
<thead>
<tr>
<th>Group</th>
<th>Class</th>
<th>Type</th>
<th>Group</th>
<th>Class</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>g2</td>
<td>MyGrp.B</td>
<td>&lt;: MyGrp.A</td>
<td>g2</td>
<td>MyGrp.C</td>
<td>&lt;: g2.A</td>
</tr>
<tr>
<td>g3</td>
<td>MyGrp.B</td>
<td>&lt;: MyGrp.A</td>
<td>g3</td>
<td>MyGrp.C</td>
<td>&lt;: g1.D</td>
</tr>
<tr>
<td>g1</td>
<td>MyGrp.A</td>
<td>&lt;: MyGrp.A</td>
<td>g1</td>
<td>MyGrp.A</td>
<td>&lt;: g2.A</td>
</tr>
<tr>
<td></td>
<td>g2.B</td>
<td>&lt;: g2.A</td>
<td></td>
<td>g3.B</td>
<td>&lt;: g3.A</td>
</tr>
<tr>
<td></td>
<td>g1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Notice, the group context is necessary when t or t’ reference MyGrp. For example, g2.B is a subtype of MyGrp.A in the context of g2, whereas g2.B is not a subtype of MyGrp.A in the context of g1 (in fact, MyGrp.A is not even a type in the context of g1). Additionally, we notice that, as a relative subclass of class A in the context of group g2, class B is a also a subtype of class A in the context of group g3, a subgroup of group g2. Hence, we summarise by informally specifying that all subtype relationships valid in the context of a group g, are “inherited” by the subgroups g’ of group g.

Clearly, g2.C is an absolute subtype of g2.A; this judgement requires no context. However, g3.B is not a subtype of g2.B; the formed further binds the latter.

3.6.1.2 Typings

Given the involved explanation of field (and method) access, it is only proper that we first provide examples of this kind. We provide two quintessential examples.

Firstly, consider the expression (new g3.A).f2. By rule (NewNull)T, e = new g3.A has type g3.A. The type of e is therefore absolute, hence, our lookup of field f2 may proceed without recourse to substitution. F(g3.A,f2) returns t’ = MyGrp.A (the definition inherited via the further binding of g2.A in g3.A), a relative type. The field access is now made relevant to the type (g3.A) of the target of field access (new g3.A) by substituting MyGrp references in t’ by the group reference in the type of e, namely g3. Hence, (new g3.A).f2 is typed g3.A.

Secondly, consider the expression this.f2. Further consider an environment Γ, such that MyGrp(Γ) = g3 and Γ(this) = MyGrp.A. By rule (VarThis)T, e = this has type MyGrp.A in the context of Γ. The type of e is therefore relative (in the context of our environment), hence, we first resolve MyGrp.A to an absolute type via substitution of context, i.e. ta = MyGrp.A[Γ], for our lookup to have meaning: by our earlier definition of substitution and Γ, we can see that ta = g3.A. Our field lookup then proceeds as before: F(g3.A,f2) returns t’ = MyGrp.A. The field access is again made relevant to the type (MyGrp.A in the context of Γ) of the target of field access (this) by substituting MyGrp references in t’ by the group reference in the type of e, namely MyGrp. We notice that
the substitution has no effect such that \( t'[\text{MyGrp.A}] = t' = \text{MyGrp.A} \). Hence, \text{this}.f2 (in the context of \( \Gamma \)) is typed \text{MyGrp.A}.

A summary of our remaining examples is presented in the following table:

\[
\begin{array}{ccc}
\Gamma_2 &=& g2.A \text{ this}.g2.A x \\
\Gamma_2 &\vdash& x.f2 : g2.A \\
\Gamma_2 &\vdash& \text{this}.f2 : \text{MyGrp.A} \\
\Gamma_4 &=& g3.C \text{ this}.g3.B x \\
\Gamma_4 &\vdash& \text{this}.f2 : g2.A \\
\Gamma_4 &\vdash& x.f2 : g3.A \\
\Gamma_4 &=& g3.B \text{ this...} \\
\Gamma_4 &\vdash& \text{this}.f2 : \text{MyGrp.A}
\end{array}
\]

### 3.6.2 Motivation

The alert reader will no doubt question why a subtype relationship does not exist between further bound classes. The reasoning is twofold: firstly, the presence of a subtype relationship would no longer make the following expression (taken from Chapter 1) a type error: \text{n1.connect(cn1)}. Hence, our system would no longer solve the graph problem; the previous expression has allowed us to connect a plain node to a coloured node. Secondly, for reasons of safety. Methods defined with relative return or parameter types are covariantly redefined in the context of a subgroup. For example, in the context of group \( g2 \), method \( m1 \) (defined in class \text{g2.A}) has type \text{g2.A m1(g2.A)}. However, when further bound in group \( g3 \), the type in this new context is \text{g3.A m1(g3.A)}. In section 7 of [9], Bruce deals with this issue in the context of \text{MyType} and binary methods. The argument transfers to \text{CONCORD} in the case of method \( m2 \): the type of the method parameter varies covariantly with the size of class definition (remembering we can extend class \text{A} when further bound in group \text{g3}). The requirement for subtypes (presented in the paper) is that method parameters should vary contravariantly.

It must also be noted that every class, in the context of its defining group and subgroups of the defining group, is a subtype of \text{GlobalGroup}.c. However, according to the subtype rules presented in Figure 3.10, these relationships have absolutely no bearing on subtype relationships between non-\text{GlobalGroup} classes, i.e. no new subtypes relationships are generated by this fact.

### 3.7 Soundness and Decidability

In Figure 3.13, we define agreement between addresses and types, \( \sigma \vdash \iota : \text{ta} \), whereby an address agrees with the runtime type (or any supertype) of the corresponding object, and \text{null} agrees with all types. An address \( \iota \) corresponds to a well formed object, \( \sigma \vdash \iota \), if all the fields declared in its runtime type, \text{g.c}, have values which agree with their types in \text{g.c} (modulo substitution of \text{MyGrp} by \text{g}). A store is well-formed, \( \Gamma \vdash \sigma \Diamond \), if all addresses correspond to well formed objects and \text{this} and \( x \) contain addresses which agree with their types in \( \Gamma \).

We are now in a position to prove soundness of our type system: this important property is expressed in Theorem 2, reproduced here for ease of reference.
The requirement that $e[\Gamma] = e[\sigma]$ guarantees that either there are no occurrences of MyGrp in $e$, or that $\sigma(MyGrp) = \Gamma(MyGrp)$. This requirement is necessary when proving the step for new MyGrp.c and it is guaranteed by the substitutions taking place when inheriting through subclassing. Also, the type of $x$ does not change.

It is easy to argue that typing in Concord is decidable: The lookup functions $F$ and $M$ depend on a class’s supergroup and superclass, and these relationships are acyclic. The subtype relationship is the transitive closure of the extensions defined in the program, with some substitution of groups. The type system has the sub-formula property and the subexpressions are strictly smaller.
Chapter 4

Proofs

In this chapter, we present the lemmata and theorems developed in the process of proving soundness. Regrettably, time constraints prohibited the completion of every proof and so this Chapter provides work covered in the time available. Motivation for each proof is to be found in discussion of the formal system in Chapter 3.

4.1 Lemmata and Theorems

Lemma 1

\[ \vdash ta <: ta' \implies \forall g : g \vdash ta <: ta' \]

Proof of Lemma 1:

1: \[ \vdash ta <: ta' \] (given)
2: \[ \exists g' : g' \vdash ta <: ta' \] (1, (ST₃))
3: \[ g \vdash ta[g'] <: ta'[g'] \] (2, (ST₂), arbitrary g)
4: \[ ta[g'] = ta \] (Def of substitution and 3)
5: \[ ta'[g'] = ta' \] (Def of substitution and 3)
6: \[ g \vdash ta <: ta' \] (3, 4, 5)

Given the use of arbitrary \( g \), it can be concluded that \( \forall g : g \vdash ta <: ta' \), which was to be shown.

Lemma 2

\[ g \vdash t <: t', \{ t' = MyGrp.c' \} \implies \exists c : t = MyGrp.c \]

Proof of Lemma 2:
4. Proofs

By induction on \( g \vdash t <: t' \)

**Case 1:** (\( ST_1 \))

Then \( \exists c : g \vdash MyGrp.c <: t' \) as required.

Case complete.

**Case 2:** (\( ST_2 \))

This rule cannot be applied. Substitution guarantees instances of \( MyGrp \) are removed in every type passed as an argument to the function. Hence, bottom line of (\( ST_2 \)) will be of the form \( g' \vdash ta <: ta' \).

Case complete.

**Case 3:** (\( ST_3 \))

1: \( g \vdash t <: t' \) (given)
2: \( \exists g' : g \ll g' \) (1, current case)
3: \( \exists g' : g' \vdash t <: t' \) (1, current case)
4: \( \exists c : t = MyGrp.c \) (Inductive hypothesis on 2, 3)

As required. Case complete.

**Case 4:** (\( ST_4 \))

5: \( t' = MyGrp.c' \) (given)
6: \( g \vdash t <: t' \) (given)
7: \( \exists t'' : g \vdash t <: t'' \) (6, (\( ST_4 \)))
8: \( \exists t'' : g \vdash t'' <: t' \) (6, (\( ST_4 \)))
9: \( \exists c' : t'' = MyGrp.c' \) (Inductive hypothesis on 8, 5)
10: \( g \vdash t <: MyGrp.c' \) (9)
11: \( \exists c : t = MyGrp.c \) (Inductive hypothesis on 10, 9)

As required.

By inspection, (\( ST_5 \)) cannot be applied, hence we have shown what was required.

**Lemma 3**  
Given \( g \ll g' \):

1. \( \mathcal{F}(g'.c,f) = t \implies \mathcal{F}(g.c,f) = t \)
2. \( \mathcal{M}(g'.c,m) = t_0 m(t_1 x) \{ \ldots \} \implies \mathcal{M}(g.c,m) = t_0 m(t_1 x) \{ \ldots \} \)
Lemmata and Theorems

Proof of Lemma 3 (part 1 — fields):

By induction on the length $n$ of $\vdash g \ll g'$.

**Case 1: $n = 0$ (Base case)**

$n = 0 \implies g = g'$. Immediate. Case complete.

**Case 2: $n > 0$ (Inductive case)**

i.e. $\vdash g \ll g'$ and $g \neq g'$. Furthermore:

1: $\exists g'' : g \neq g''$ and $\vdash g \ll g''$ and $\vdash g'' \ll g'$

Then:

2: $\exists g''' : \vdash g \ll g'''$ and $\vdash g''' \ll g'$

where length $\vdash g \ll g'''$ is 1. i.e.:

3: $G(g) = \text{group } g \ll g'' \{ \ldots \}$

Notice:

4: $\vdash g''' \ll g'$ has length $n - 1$

We continue. We are given $F(g'.c,f) = t$ hence:

5: $F(g'''.c,f) = t$ (Inductive hypothesis on 4)

6: $\exists g'.c' : C(g'''.c) = \text{class } c : g'.c' \{ \ldots \}$

(5, (F3))

We must analyse two cases with regard to $g.c$.

**Case 2.A: CW(g.c) = lhf**

7: $C(g.c) = C(g'''.c)$

(5, (C4), 3, current case)

8: $C(g.c) = \text{class } c : g'.c' \{ \ldots \}$

(6, 7)

9: $F(g'.c',f) = lhf$

(\vdash g.c \Diamond_{uniq}, 3, 5, 8)

Assume for contradiction that $FW(g.c,f) \neq lhf$.

By (F1), $CW(g.c) = \text{class } c : \ldots \{ \ldots \} \neq lhf$. This contradicts current case.

Therefore:
4. Proofs

10: \( F(W(g.c.f) = Ulf \) (by contradiction)
11: \( F(g.c.f) = Ulf \oplus Ulf \oplus F(g''.c.f) \) (3, 8, 9, 10, (F_3))
12: \( F(g.c.f) = F(g''.c.f) = t \) (\( \oplus \) def, 11, 5)

As required. Case complete.

CASE 2.B: \( CW(g.c) \neq Ulf \)

Then \( \exists t' = gr'.c'' \) such that:

13: \( G(g) = \text{group } g < < g'' \{ \ldots \text{class } c < : t' \{ \ldots \} \ldots \} \) ((C_4), current case)
14: \( C(g.c) = \text{class } c < : gr'.c'' \{ \ldots \} \) (3, 13)
15: \( C(g.c) = \text{class } c < : gr'.c'' \{ \ldots \} \) (14, current case, (C_3))
16: \( F(gr'[g.c''.f) = Ulf \) (5, 3, 15, \( \vdash g.c \otimes_{uniq} \))

Assume for contradiction, \( F(W(g.c.f) \neq Ulf \).

Then by \( \vdash g.c \otimes_{\mu}, 3 \text{ and } 15, F(g''.c.f) = Ulf \). This contradicts 5.

Thereby:

17: \( F(W(g.c.f) = Ulf \) (by contradiction)
18: \( F(g.c.f) = Ulf \oplus Ulf \oplus F(g''.c.f) \) (3, 15, 16, 17, (F_3))
19: \( F(g.c.f) = F(g''.c.f) = t \) (\( \oplus \) def, 18, 5)

As required. Case complete.

End of sub-cases. Therefore, we have shown what was required. \( \square \)

Proof of Lemma 3 (part 2 — methods):

By induction on the length \( n \) of \( \vdash g < < g' \).

CASE 1: \( n = 0 \) (Base case)

\( n = 0 \implies g = g' \). Immediate. Case complete.

CASE 2: \( n > 0 \) (Inductive case)

I.e. \( \vdash g < < g' \) and \( g \neq g' \). Furthermore:

1: \( \exists g'' : g \neq g'' \) and \( \vdash g < < g'' \) and \( \vdash g'' < < g' \) (current case)

Then:
Lemmata and Theorems

2: \exists g'' : \vdash g \ll g'' \text{ and } \vdash g'' \ll g' \quad (1)

where \( \vdash g \ll g'' \) is 1. i.e.:

3: \( \mathcal{G}(g) = \text{group } g \ll g'' \{ \ldots \} \) \quad (2)

Notice:

4: \( \vdash g'' \ll g' \) has length \( n - 1 \) \quad (observation)

We continue. We are given \( \mathcal{M}(g'.c,m) = t_0 \ m(t_1 \ x) \{ \ldots \} \) hence:

5: \( \mathcal{M}(g''.c,m) = t_0 \ m(t_1 \ x) \{ \ldots \} \) \quad (Inductive hypothesis on 4)

6: \( \exists \text{gr}.c' : \mathcal{C}(g''.c) = \text{class } c <: \text{gr}.c' \{ \ldots \} \) \quad (5, (M3))

We must analyse two cases with regard to \( g \cdot c \).

CASE 2.A: \( \mathcal{C}W(g.c) = \text{Ulf} \)

7: \( \mathcal{C}(g.c) = \mathcal{C}(g'' . c) \) \quad ((C4), 3, current case)

8: \( \mathcal{C}(g.c) = \text{class } c <: \text{gr}.c' \{ \ldots \} \) \quad (6, 7)

9: \( \mathcal{M}(\text{gr}[g].c',m) = \text{Ulf} \) \quad (\( \vdash g . c \bowtie_{uniq} 3, 5, 8 \))

Assume for contradiction that \( \mathcal{M}W(g.c,m) \neq \text{Ulf} \).

By (M1), \( \mathcal{C}W(g.c) = \text{class } c <: \{ \ldots \} \neq \text{Ulf} \). This contradicts current case. Therefore:

10: \( \mathcal{M}W(g.c,m) = \text{Ulf} \) \quad (by contradiction)

11: \( \mathcal{M}(g.c,m) = \text{Ulf} \oplus \text{Ulf} \oplus \mathcal{M}(g'' . c.m) \) \quad (3, 8, 9, 10, (M3))

12: \( \mathcal{M}(g.c,m) = \mathcal{M}(g''.c,m) = t_0 \ m(t_1 \ x) \{ \ldots \} \) \quad (\( \oplus \text{def}, 11, 5 \))

As required. Case complete.

CASE 2.B: \( \mathcal{C}W(g.c) \neq \text{Ulf} \)

Then \( \exists t' = \text{gr}.c'', g''' \) such that:

13: \( \mathcal{G}(g) = \text{group } g \ll g''' \{ \ldots \text{class } c <: t' \{ \ldots \} \ldots \} \) \quad ((C1), current case)

14: \( \mathcal{G}(g) = \text{group } g \ll g'''' \{ \ldots \text{class } c <: \text{gr'} . c'' \{ \ldots \} \ldots \} \) \quad (3, 13)

15: \( \mathcal{C}(g.c) = \text{class } c <: \text{gr'} . c'' \{ \ldots \} \) \quad (14, current case, (C3))

16: \( \mathcal{M}(\text{gr'}[g].c'' ,m) = \text{Ulf} \) \quad (3, 5, 15, \( \vdash g . c \bowtie_{uniq} \))

We must analyse two cases with regard to \( m \) in \( g \cdot c \).
4. Proofs

Case 2.b.i: \( MW(g, c, m) \neq \text{Ulf} \)

17: \( MW(g, c, m) = t_0 m(t_1 x) \{ \ldots \} \)
18: \( M(g, c, m) = MW(g, c, m) \oplus \ldots \)
19: \( M(g, c, m) = MW(g, c, m) = t_0 m(t_1 x) \{ \ldots \} \)

As required. Case complete.

Case 2.b.ii: \( MW(g, c, m) = \text{Ulf} \)

20: \( M(g, c, m) = \text{Ulf} \oplus \text{Ulf} \oplus M(g''', c, m) \)
21: \( M(g, c, m) = M(g''', c, m) = t_0 m(t_1 x) \{ \ldots \} \)

As required. Case complete.

All cases analysed, therefore we have proven what was required.

Lemma 4  Take any \( g, g' \) with \( \vdash g \preceq g' \):

\[
g' \vdash t < t' \implies g \vdash t < t'
\]

Proof of Lemma 4:

1: \( \vdash g \preceq g' \) \hspace{1cm} (given)
2: \( g' \vdash t < t' \) \hspace{1cm} (given)
3: \( g \vdash t < t' \) \hspace{1cm} (1, 2, (ST_3))

As required.

Lemma 5

\[
\begin{align*}
g \vdash t_1 < t_2, \\
& \quad t_1[g] = g_1 \cdot c_1, \\
& \quad t_2[g] = g_2 \cdot c_2, \\
& \quad F(g_2, c_2, f) = t'
\end{align*}
\]

\( \implies F(g_1, c_1, f) = t' \) and \( g_1 = g_2 \) or

\( \implies F(g_1, c_1, f) = t'[g_2] \)

Proof of Lemma 5:

By induction on \( g \vdash t_1 < t_2 \).

54
1: $g \vdash t_1 <: t_2$ (given)

2: $t_1[g] = g_1.c_1$ (given)

3: $t_2[g] = g_2.c_2$ (given)

4: $F(g_2.c_2, f) = t'$ (given)

**Case 1:** (ST1)

5: $t_1 = MyGrp.c_1$ (inspection of rule, 2)

6: $C(g.c_1) = class c_1 <: t_2 \{ \ldots \}$ (current case, 5)

7: $\exists gr : t_2 = gr.c_2$ (convenience def)

8: $g_1 = g$ (2, 5)

9: $C(g_1.c_1) = class c_1 <: t_2 \{ \ldots \}$ (6, 8)

10: $\exists g' : G(g_1) = group g_1 << g' \{ \ldots \}$ (9, (C1), (C3), (C4))

Assume for contradiction that $FW(g_1.c_1, f) \neq Ulf$.

By 10, 9 and $g.c \Diamond_{inh}, F(t_2[g_1], f) = Ulf$. Therefore by 8 and 3, $F(g_2.c_2, f) = Ulf$. Contradiction with 4. Therefore:

11: $FW(g_1.c_1, f) = Ulf$ (by contradiction)

12: $F(g_1.c_1, f) = Ulf \oplus F(gr[g_1].c_2, f)[gr] \oplus \ldots$ (9, 10, 11, (F3))

13: $gr[g_1].c_2 = t_2[g_1] = t_2[g] = g_2.c_2$ (def of substitution, 7, 3, 8)

14: $F(g_1.c_1, f) = t'[gr]$ (12, 13, $\oplus$ def, 4, def of substitution)

We have two cases to consider with respect to gr.

**Case 1.a:** gr = MyGrp

15: $t'[gr] = t'$ (current case, def of substitution)

16: $t_2[g] = g_2.c_2 = g.c_2$ (3, current case, 7)

17: $g = g_2 = g_1$ (16, 8)

By 17, 15 and 14, we have $F(g_1.c_1, f) = t'$ and $g_1 = g_2$ as required.

Case complete.

**Case 1.b:** gr = $g_3$ (for some $g_3$)

18: $g_3[g].c_2 = g_3.c_2$ (current case, def of substitution)

19: $g_3 = g_2$ (18, 3, 7)

20: $F(g_1.c_1, f) = t'[g_2]$ (14, current case, 19)

As required. Case complete.
4. Proofs

Case 2: (ST_2)

We have $g \vdash t_1 <: t_2$ in $n + 1$ steps.

1. $\exists g', t_3, t_4 : g' \vdash t_3 <: t_4$ in $n$ steps (current case)
2. $t_1 = t_3[g']$ (21, current case)
3. $t_2 = t_4[g']$ (21, current case)
4. $g_1 \cdot c_1 = t_1[g] = t_3[g'][g] = t_3[g']$ (def of substitution, 2, 3, 22, 23)
5. $g_2 \cdot c_2 = t_2[g] = t_4[g'][g] = t_4[g']$ (def of substitution, 2, 3, 22, 23)
6. $F(t_4[g'], f) = t'$ (4, def of substitution, 2, 3, 22, 23)
7. $F(g_1 \cdot c_1, f) = t'$ and $g_1 = g_2$ (24, 25, Inductive hypothesis on 21, 24, 25, 26)

As required. Case complete.

Case 3: (ST_3)

By 1, $\exists g_3$ such that

1. $\vdash g < g_3$ (current case)
2. $g_3 \vdash t_1 <: t_2$ (current case)

We now perform a case analysis on $t_1$ and $t_2$.

Case 3.A: $t_1 = MyGrp \cdot c_1$ AND $t_2 = MyGrp \cdot c_2$

30. $t_1[g] = g_1 \cdot c_1$ (current case, def of substitution, 2)
31. $t_2[g] = g_2 \cdot c_2$ (current case, def of substitution, 3)
32. $g_3 \vdash g \cdot c_1 <: g \cdot c_2$ (1, (ST_2), 30, 31)
33. $t_3 = g \cdot c_1$ (convenience def)
34. $t_4 = g \cdot c_2$ (convenience def)
35. $t_3[g_3] = g \cdot c_1$ (33, 34, def of substitution)
36. $t_4[g_3] = g \cdot c_2$ (33, 34, def of substitution)
37. $F(g \cdot c_2, f) = F(g_2 \cdot c_2, f) = t'$ (31, 4)

By inductive hypothesis on 32, 37, 35, 36, 33, 34

38. $F(g \cdot c_1, f) = t'$ and $g = g_1 \lor F(g \cdot c_1, f) = t'[g]$ (30, 31)
39. $g_1 = g_2 = g$ (30, 31)
40. $F(g_1 \cdot c_1, f) = t'$ and $g_1 = g_2 \lor F(g_1 \cdot c_1, f) = t'[g_2]$ (38, 39)

As required. Case complete.

Case 3.B: $t_1 = MyGrp \cdot c_1$ AND $t_2 = g_a \cdot c_2$ (FOR SOME $g_a$)
41: \( t_1[g] = g_1 \cdot c_1 = g \cdot c_1 \) (2, current case, def of substitution)

42: \( t_2[g] = g_2 \cdot c_2 = g_a \cdot c_2 \) (2, current case, def of substitution)

43: \( g_3 \vdash g \cdot c_1 <: g_a \cdot c_2 \) (1, \( \text{ST}_2 \), 41, 42)

44: \( g_a = g_2 \) (41, 42)

45: \( g_1 = g \) (41, 42)

46: \( F(g_a \cdot c_2, f) = t' \) (44, 4)

47: \( t_3 = g \cdot c_1 \) (convenience def)

48: \( t_4 = g_a \cdot c_2 \) (convenience def)

49: \( t_3[g_3] = g \cdot c_1 \) (def of substitution, 46, 47)

50: \( t_4[g_3] = g_a \cdot c_2 \) (def of substitution, 46, 47)

By inductive hypothesis on 43, 46, 47, 48, 49, 50:

51: \( F(g \cdot c_1, f) = t' \) and \( g = g_a \) or \( F(g \cdot c_1, f) = t'[g_a] \)

Using 51, 44, 45:

52: \( F(g_1 \cdot c_1, f) = t' \) and \( g_1 = g_2 \) or \( F(g_1 \cdot c_1, f) = t'[g_2] \)

As required. Case complete.

CASE 3.c: \( t_1 = g_a \cdot c_1 \) AND \( t_2 = \text{MyGrp} \cdot c_2 \)

Vacuous case. By Lemma 2, no such case can exist.

CASE 3.d: \( t_1 = g_a \cdot c_1 \) AND \( t_2 = g_b \cdot c_2 \) (FOR SOME \( g_a, g_b \))

53: \( t_1[g] = g_a \cdot c_1 = g_1 \cdot c_1 \) (2, 3, current case, def of substitution)

54: \( t_2[g] = g_b \cdot c_2 = g_2 \cdot c_2 \) (2, 3, current case, def of substitution)

55: \( g_a = g_1 \) (53)

56: \( g_b = g_2 \) (54)

57: \( t_1[g_3] = t_1 \) (def of substitution, current case)

58: \( t_2[g_3] = t_2 \) (def of substitution, current case)

By induction on 4, 29, 57, 58, 53, 54:

59: \( F(g_1 \cdot c_1, f) = t' \) and \( g_1 = g_2 \) or \( F(g_1 \cdot c_1, f) = t'[g_2] \)

As required. Case complete.

CASE 4: \( \text{ST}_4 \)

Then \( \exists t_3 \) such that:
4. Proofs

60: \( g \vdash t_1 <: t_3 \)  
61: \( g \vdash t_3 <: t_1 \)  
62: \( t_3 = \text{gr} \cdot c_3 \)  
(convenience def, arbitrary gr, arbitrary c_3)

We consider two cases with respect to gr:

**CASE 4.A:** \( \text{gr} = \text{MyGrp} \)

Then:

63: \( t_3[g] = g \cdot c_3 \)  
(62, current case)

By induction on 61, 63, 3, 4:

64: \( F(g \cdot c_3, f) = t' \) and \( g = g_2 \) or \( F(g \cdot c_3, f) = t'[g_2] \)

We consider these two possibilities:

**CASE 4.A.1:** \( F(g \cdot c_3, f) = t' \) AND \( g = g_2 \)

By induction on \( F(g \cdot c_3, f) = t' \) (current case), 63, 60, 2:

65: \( F(g_1 \cdot c_1, f) = t' \) and \( g_1 = g \) or \( F(g_1 \cdot c_1, f) = t'[g] \)

By \( g = g_2 \) (current case), result is immediate. Case complete.

**CASE 4.A.2:** \( F(g \cdot c_3, f) = t'[g_2] \)

By induction on \( F(g \cdot c_3, f) = t'[g_2] \) (current case), 63, 2, 60:

66: \( F(g_1 \cdot c_1, f) = t'[g_2] \) and \( g = g_1 \) or \( F(g_1 \cdot c_1, f) = t'[g_2][g] \)

67: \( t'[g_2][g] = t'[g_2] \)  
(66, def of substitution)

68: \( F(g_1 \cdot c_1, f) = t'[g_2] \)  
(66, 67)

As required. Case complete.

**CASE 4.B:** \( \text{gr} = g_a \) (FOR SOME \( g_a \))

Then:

69: \( t_3[g] = g_a \cdot c_3 \)  
(68, current case, def of substitution)

By induction on 61, 69, 3, 4:
Lemmata and Theorems

70: \( F(g_a \cdot c_3, f) = t' \) and \( g_a = g_2 \) or \( F(g_a \cdot c_3, f) = t'[g_2] \)

We consider these two possibilities:

**Case 4.b.i**: \( F(g_a \cdot c_3, f) = t' \) and \( g_a = g_2 \)

By induction on \( F(g_a \cdot c_3, f) = t' \) (current case), 69, 60, 2:

71: \( F(g_1 \cdot c_1, f) = t' \) and \( g_1 = g_0 \) or \( F(g_1 \cdot c_1, f) = t'[g_a] \)

By \( g_a = g_2 \) (current case), result is immediate. Case complete.

72: \( F(g_1 \cdot c_1, f) = t'[g_2] \)

By induction on \( F(g_a \cdot c_3, f) = t'[g_2] \) (current case), 69, 2, 60:

73: \( F(g_1 \cdot c_1, f) = t'[g_2] \)

As required. Case complete.

All case examined, therefore we have shown what was required.

---

**Lemma 6**

Take any \( g, c, g', c' \) with \( \vdash g . c <: g' . c' \):

1. \( F(g'.c',f) = t \implies F(g.c,f) = t \) and \( g' = g \) or \( F(g.c,f) = t[g'] \)
2. \( M(g'.c',f) = t_0 m(t_1 x) \{ ... \} \implies M(g.c,m) = t_0 m(t_1 x) \{ ... \} \) and \( g' = g \) or \( M(g.c,m) = t_0[g'][t_1[g'] x] \{ ... \} \)

**Lemma 7**

\[
\Gamma \vdash \text{gr}.c' \diamond t,
\Gamma(\text{MyGrp}) = g',
\Gamma(\text{this}) = \text{MyGrp}.c \text{ for some } c,
\vdash g <\ll g',
\Gamma' = \Gamma[\text{this} \mapsto g . c],
\implies \Gamma' \vdash \text{gr}.c' \diamond t
\]

**Proof of Lemma 7:**

Proof by induction on the length \( n \) of \( \vdash g <\ll g' \).

**Case 1**: \( n = 0 \) (Base Case)

59
4. Proofs

i.e. $g = g'$. Therefore, $\Gamma = \Gamma'$ and result is immediate. Case complete.

**Case 2: $n > 0$ (Inductive Case)**

i.e. $\vdash g \ll g'$ and $g \neq g'$. Furthermore:

1. $\exists g'' : g \neq g''$ and $\vdash g \ll g''$ and $\vdash g'' \ll g'$

Then:

2. $\exists g'' : \vdash g \ll g''$ and $\vdash g'' \ll g'$(current case)

where length $\vdash g \ll g''$ is 1. Then:

3. $G(g) = \text{group } g \ll g'' \{ ... \}$

Notice:

4. $\vdash g'' \ll g$ has length $n - 1$

By induction on 4, $\exists \Gamma''$ such that:

5. $\Gamma'' \vdash \text{gr}.c' \Diamond t$

where

6. $\Gamma'' = \Gamma[\text{this} \mapsto g''.c]$

We consider two cases with respect to $\text{gr}$:

**Case 2.A: $\text{gr} = g_0$ (for some $g_0$)**

7. $\vdash g_0.c' \Diamond c$

8. $\Gamma' \vdash g_0.c' \Diamond t$

As required.

**Case 2.B: $\text{gr} = \text{MyGrp}$

9. $\Gamma' = \Gamma[\text{this} \mapsto g.c]$

10. $\Gamma'(\text{MyGrp}) = g$

We have two cases to consider with respect to $g.c'$:
Case 2.b.i: $CW(g, c') = Ulif$

Then:

11: $\exists g_1 : g_1 . c' \diamond_c$  
12: $\Gamma''(MyGrp) = g_1$  
13: $\Gamma''(MyGrp) = g''$  
14: $g_1 = g''$  
15: $\vdash g''. c' \diamond_c$  
16: $\mathcal{C}(g . c') = \mathcal{C}(g''. c')$  
17: $\mathcal{C}(g''. c') = \text{class } c' <: \ldots \{ \ldots \}$  
18: $\vdash g . c' \diamond_c$  
19: $\Gamma' : MyGrp . c' \diamond_t$

Given current case, we have required result. Case complete.

Case 2.b.ii: $CW(g, c') \neq Ulif$

20: $CW(g, c') = \text{class } c' <: \ldots \{ \ldots \}$  
21: $\mathcal{C}(g . c') = \text{class } c' <: \ldots \{ \ldots \}$  
22: $\vdash g . c' \diamond_c$  
23: $\Gamma' : MyGrp . c' \diamond_t$

Given current case, we have required result. Case complete.

All cases considered, we have shown what was required. \(\square\)

Lemma 8

\[ \vdash g <= g', \]  
\[ \Gamma(this) = MyGrp . c \text{ for some } c, \]  
\[ \Gamma(MyGrp) = g'. \]  
\[ \Gamma' = \Gamma[this \mapsto g . c], \]  
\[ \Gamma \vdash e : t \]  

Definition 1

\[ \Gamma' <: \Gamma \iff \left\{ \begin{array}{l} \Gamma = g . c \text{ this, } t x \\ \Gamma' = g'. c' \text{ this, } t' x \\ (\Gamma' \vdash MyGrp . c' <: gr . c \text{ or } \Gamma' \vdash t' <: t[g]) \\ g = gr[g'] \end{array} \right\} \]

Lemma 9

\[ \Gamma \vdash e : t \]  
\[ \Gamma' <: \Gamma \]  

[61]
4. Proofs

Lemma 10  Given $\vdash g \ll g'$:

$C(g.c) = \text{class } c <\text{: MyGrp}.c \{ \ldots \} \implies C(g'.c) = \text{class } c <\text{: MyGrp}.c \{ \ldots \}$

Theorem 1  In a well formed program:

$\mathcal{M}(ta,m) = t_1 \ m(t_2 \ x)\{e\} \implies ta \ this,t_2 \ x \vdash e : t_1$

Theorem 2 (Soundness)

$\vdash P \diamond$

$\Gamma \vdash \sigma \diamond$

$\Gamma \vdash e : t$

$e, \sigma \sim i, \sigma'$

$e[\Gamma] = e[\sigma]$

$\Rightarrow \Gamma \vdash \sigma' \diamond$

$\sigma' \vdash i : t[\Gamma]$

$\sigma' \vdash i : t[\Gamma]$

$\Rightarrow \Gamma \vdash \sigma' \diamond$

$\sigma' \vdash i : t[\Gamma]$
4.2 Properties that did not hold

We had thought to include the following Lemma:

**Lemma 11**  
Take any $g, g'$ with $\vdash g << g'$:

$\vdash g'.c <= ta \implies \vdash g.c <= ta$

However, the following code listing provides a counter example:

```plaintext
group g1 {
    class c <= MyGrp.d {}
    class d {}
}

group g2 << g1 {
    // c further bound
    // d further bound
}
```

i.e. Lemma 11 fails to take into account relative superclasses.

Clearly $g1 \vdash MyGrp.c <= MyGrp.d$. By Lemma 4, $g2 \vdash MyGrp.c <= MyGrp.d$. However, the property would only hold in the case where $c$ is defined as an absolute subclass of $d$; this restriction does not form a part of the Lemma.
Chapter 5

Examples in Concord

In this chapter, we offer encodings of well know examples in Concord. These examples were harvested from the literature in the process of researching Chapter 2. Such examples are commonly developed as motivating problems and as such are salient indicators of a language’s expressiveness and capabilities. Our presentation therefore demonstrates the expressiveness of Concord but does so in light of our goals [34]. We present three examples: multi-dimensional points, the cow example and the expression problem.

5.1 Encoding ThisClass — multi-dimensional points

As we explain in Section 6.4, ThisClass allows references to the type of self: we informally summarise the capabilities of this reference to include families containing only one member. For our encoding of ThisClass, we therefore present the widely known one- and two-dimensional points example. The example describes points that exist with varying dimensions (in this case one and two dimensions) and a method for testing equality between points of similar dimension.

A method equal, defined in classes Point1D and Point2D, should test either two Point1D objects for equality, or two Point2D objects for equality, but never a Point1D and a Point2D object (we see shortly why this would be unsafe).

In [11], this requirement is enforced via a parameter of type ThisClass, indicating that the argument should have the same class as the receiver. Imagine an encoding where class Point2D is defined to extend\(^1\) class Point1D. Therefore, the definition of method equal() in the class Point1D with type ThisClass equal(ThisClass), when inherited by the class Point2D, requires the more precise parameter of type Point2D and returns the more precise Point2D in the context of class Point2D.

The concept of ThisClass does not exist in Concord but can very naturally be encoded via a MyGrp reference and further binding. This can be seen in Listing 5.1.

\(^1\)Notice how we carefully choose the word ‘extend’ instead of ‘subclass’. The latter is widely considered, thanks in part to the popularity of Java, to imply subtypes. Where we have covariance in a method parameter in subclasses, we cannot have subtypes: this point is discussed in Subsection 3.6.2
5. Examples in Concord

### Listing 5.1 — The multi-dimensional points example encoded in Concord

```
group OneDim {
  class Point {
    int d1;
    bool equal(MyGrp.Point x) { this.d1 == x.d1 }
  }
}
group TwoDim << OneDim {
  class Point {
    int d2;
  }
}
```

The only seemingly awkward feature of our encoding is the unusual names given to the groups: this is a side effect of our earlier observation that points are the sole members of the family under consideration.

Assuming that od1 and od2 are variables of type OneDim.Point, and td1 and td2 are variables of type TwoDim.Point, the following expressions type check as follows:

- **Type correct:**
  - od1.equal(od2);
  - td1.equal(td2);
  - od1 = od2; td1 = td2;

- **Type incorrect:**
  - od1.equal(td1);
  - td1.equal(od1);
  - od1 = td1;

 Needless to say, in our extension of the further bound class Point we could also have overwritten the method equal in order to test for equality in the additional dimension:

```
group TwoDim << OneDim {
  class Point {
    int d2;
    bool equal(MyGrp.Point x) {
      this.d1 == x.d1 and this.d2 == x.d2
    }
  }
}
```

However, overwriting the method as follows would be not well-formed (such a definition would be permitted if we supported overloading as in e.g. JAVA):

```
group TwoDim << OneDim {
  class Point {
    int d2;
    bool equal(TwoDim.Point x) {
      this.d1 == x.d1 and this.d2 == x.d2
    }
  }
}
```
Notice that, in CONCORD, given the function:

```java
bool equalPoints(OneDim.Point x, OneDim.Point y) {
    x.equal(y)
}
```

the call `equalPoints(od1, od2)` is legal, but `equalPoints(td1, td2)` is illegal. As execution of the latter would not fail, it must be considered a shortcoming of CONCORD that the expression is illegal. Indeed, in [11] it is legal. One means of overcoming this shortcoming is to support type and method signatures parameterised by groups (we list this suggestion amongst further work in Chapter 7). For example, this would permit such a function definition:

```java
bool equalPoints(g << OneDim, g.Point x, g.Point y) {
    x.equal(y)
}
```

5.2 The Cow Example

This humorous example has appeared in many papers but is credited, at least in [26], to Shang in [51]. In this example, animals eat food, and cows eat grass. Consider the following modelling scenario:

- class Cow is a subclass of the class Animal
- classes Grass and Meat are subclasses of the class Food
- the class Animal has a field of type Food

Given these constraints, the JAVA type system cannot prevent cows from eating meat. Indeed, this shortcoming was used to motivate the introduction of virtual types in [26]. In Listing 5.2, we show an encoding of the Cow example in CONCORD, using groups of animals, herbivores and carnivores; we thus prevent mixing of eating habits between groups.

Now, assume the following variables declarations:

```java
FoodG.Grass grass
FoodG.Meat meat
HerbivoreG.Animal herbiA
HerbivoreG.Cow cow
HerbivoreG.Food herbiF
CarnivoreG.Animal carniA
CarnivoreG.Food carniF
```

Then the following expressions would type check as indicated:

```
type correct:  herbiA.eat = herbiF;  cow.eat = herbiF;
carniA.eat = carniF;
type incorrect: herbiA.eat = grass;  carniA.eat = meat;
carniA.eat = grass;
```
group FoodG {
    class Food { int calories; }
    class Grass <: FoodG.Food {
        // standard stuff to do with grass
    }
    class Meat <: FoodG.Food {
        // standard stuff to do with meat
    }
}
group AnimalG {
    class Animal {
        MyGrp.Food eat;
    }
    class Food {
        // AnimalG.Food is unrelated to FoodG.Food
        // This is an empty placeholder class
    }
}
group HerbivoreG << AnimalG {
    class Cow <: MyGrp.Animal {
        // stuff to do with cows
    }
    class Food <: FoodG.Grass {
        // additional stuff to do with grass
    }
}
group CarnivoreG << AnimalG {
    class Dog <: MyGrp.Animal {
        // stuff to do with dogs
    }
    class Food <: FoodG.Meat{
        // additional stuff to do with meat
    }
}

Listing 5.2 — A Concord encoding of cows and dogs
The expression problem, at least in its current guise, is credited to Wadler in a post [61] to the Java-genericty mailing list [60]. The goal: to define a data-type by cases such that one can add new cases to the data-type and new functions over the data-type without recourse to recompiling existing code whilst retaining static type safety [61]. In Concord, we can express this requirement but cannot encode the combination of said extensions.

Let us consider the famous application of the expression problem to a language that describes expressions (pun intended in the original formulation of the problem): this is seen in Listing 5.3.

We can extend the BaseGroup abstract data type of expressions to support sums, i.e. one more case:

```java

group PlusGroup {
    class Plus <: MyGrp.Expr{
        MyGrp.Expr left;
        MyGrp.Expr right;
        int eval() {
            this.left.eval() + this.right.eval();
        }
    }
}
```

Thus, the following would be type correct:

```java
PlusGroup.Num n1;
PlusGroup.Num n2;
PlusGroup.Plus p;

p.left=n1;
p.right=n2;
p.eval()
```

We can also extend the BaseGroup abstract data type of expressions to support printing, i.e. extra functionality:
group ShowGroup {
    class Expr {
        String show() {
            this.value().toString();
        }
    }
}

Thus, the following would be legal:

ShowGroup.Num n;

n.show()

Concord does not support a way of combining PlusGroup and ShowGroup. It is suggested that a notion of traits or virtual superclasses might possibly achieve this goal\textsuperscript{2}.

\textsuperscript{2}Suggested by Dr. Sophia Drossopoulou
Chapter 6

Comparisons

In comparing our calculus with prior work, we seek to distinguish Concord from other solutions addressing collaborations of classes, highlighting problems solved by each approach and therefore our contribution to the field. We present comparisons with GJ (Java 1.5), Java inner classes, distinguishing subtypes, the ideas of MyType and This-Class and an executive summary of the capabilities of Scala.

6.1 Generics — GJ (Java 1.5)

In [44], Michalitsis addresses the issue of “Interlocked Types”. Michalitsis identifies that most design patterns in object oriented programming consist of families of participating types. The paper aims to develop means by which families of classes may be declared and have code statically checked to ensure no mixing between families occurs.

One approach taken in [44] is that of generics, specifically GJ. Since the publishing of [44], the proposal for GJ (initially proposed by Bracha et al. in [7]) has been adopted by Sun in the J2SE “Tiger” release of the Java programming language. We now present an encoding of the graph example from Chapter 1 in Java 1.5.

The encoding is based heavily on multi-parameterisation of classes developed in [44], in particular the Subject-Observer pattern developed in 6.3.5. However, our encoding supplements the parameterisation via generics with factory methods to ensure maximum code reuse. A class diagram of our approach can be seen in Figure 6.1.

In an abuse of our established notation, we use dash-tailed arrows to represent implementation of the LockNodeEdge interface. Abstract classes are denoted by italicised names (e.g. ANode) and concrete classes are distinguished by tele-type font (e.g. Node).

The interface LockNodeEdge is the family factory: every type of node and edge is required to know how to create instances of its sibling. For example, Node’s know how to create Node’s and Edge’s, ColouredEdge’s know how to create ColouredNode’s and ColouredEdge’s etc.. Therefore, the concrete classes shown are only required to define method bodies for the factory methods allowing us to define the remaining functionality within the abstract versions of the concepts node and edge, something that makes for very clean code. Why do we require the factory methods? Consider the example of the
method \texttt{Edge connect(Node)} which creates an edge joining receiver and parameter. Out of necessity, somewhere inside the definition of this method we will be required to invoke \texttt{new Edge()}, possibly passing the two nodes as arguments. In our generic specification of this method (parameterised by \texttt{nodeType} and \texttt{edgeType}), we would naturally encode this as \texttt{new edgeType()}. However, since type parameters are not available at runtime, such an expression is illegal [7]. Hence, we supply a factory.

Although long, the code listing that follows is clear when considered in conjunction with the class diagram\footnote{An online version of the code is available from the project website: \url{http://myitcv.org.uk/projects/vj}}:

```java
interface LockNodeEdge
    <nodeType extends ANode<nodeType, edgeType>, edgeType extends AEdge<edgeType, nodeType>>
{
    edgeType newEdge();
    nodeType newNode();
}

abstract class ANode
    <nodeType extends ANode<nodeType, edgeType>, edgeType extends AEdge<edgeType, nodeType>>
    implements LockNodeEdge<nodeType, edgeType>
{
    edgeType connect(nodeType n) {
        edgeType e = this.newEdge();
```

\textbf{Figure 6.1} — Class diagram of the graph example encoded in \texttt{JAVA} generics
class Node
    extends ANode<Node, Edge>
{
    Node() {}
    public Node newNode() { return new Node(); }  // return new Node();
    public Edge newEdge() { return new Edge(); }  // return new Edge();
}

abstract class AColouredNode
    <nodeType extends AColouredNode<nodeType, edgeType>,
    edgeType extends AColouredEdge<edgeType, nodeType>>
    extends ANode<nodeType, edgeType>
{
    void changeColour() {}
    edgeType connect1(nodeType n) {
        edgeType e = this.newEdge();
        e.setEnd1(n);
        e.setEnd2(n);
        return e;
    }
}

class ColouredNode
    extends AColouredNode<ColouredNode, ColouredEdge>
{
    int colour;
    ColouredNode() {}
    public ColouredNode newNode() { return new ColouredNode(); }  // return new ColouredNode();
    public ColouredEdge newEdge() { return new ColouredEdge(); }  // return new ColouredEdge();
}

abstract class ASpottedNode
    <nodeType extends ASpottedNode<nodeType, edgeType>,
    edgeType extends ASpottedEdge<edgeType, nodeType>>
    extends AColouredNode<nodeType, edgeType>
{
}

class SpottedNode
    extends ASpottedNode<SpottedNode, SpottedEdge>
{
    int no_spots;
    SpottedNode() {}
public SpottedNode newNode() { return new SpottedNode(); }
public SpottedEdge newEdge() { return new SpottedEdge(); }

abstract class AEdge
    <edgeType extends AEdge<edgeType, nodeType>,
     nodeType extends ANode<nodeType, edgeType>>
   implements LockNodeEdge<nodeType, edgeType>
{
    void setEnd1(nodeType n) {}
    void setEnd2(nodeType n) {}
}

class Edge
    extends AEdge<Edge, Node>
{
    Edge() {}
    public Node newNode() { return new Node(); }
    public Edge newEdge() { return new Edge(); }
}

abstract class AColouredEdge
    <edgeType extends AColouredEdge<edgeType, nodeType>,
     nodeType extends AColouredNode<nodeType, edgeType>>
   extends AEdge<edgeType,nodeType>
{
}

class ColouredEdge
    extends AColouredEdge<ColouredEdge, ColouredNode>
{
    int colour;
    ColouredEdge() {}
    public ColouredNode newNode() { return new ColouredNode(); }
    public ColouredEdge newEdge() { return new ColouredEdge(); }
}

abstract class ASpottedEdge
    <edgeType extends ASpottedEdge<edgeType, nodeType>,
     nodeType extends ASpottedNode<nodeType, edgeType>>
   extends AColouredEdge<edgeType,nodeType>
{
}

class SpottedEdge
    extends ASpottedEdge<SpottedEdge, SpottedNode>
{
    int no_spots;
    SpottedEdge() {}
}
```java
public SpottedNode newNode() { return new SpottedNode(); }
public SpottedEdge newEdge() { return new SpottedEdge(); }
}

class Test
{
    public static void main(String[] args)
    {
        Node n1, n2;
        ColouredNode cn1, cn2;
        SpottedNode sn1, sn2;

        n1 = new Node();
        n2 = new Node();
        cn1 = new ColouredNode();
        cn2 = new ColouredNode();
        sn1 = new SpottedNode();
        sn2 = new SpottedNode();

        n1.connect(n2);
        cn1.connect1(cn2);

        sn1.changeColour();
        n1.connect(cn2); // static type error
        sn1.connect(cn1); // static type error
    }
}
```

When passed through the Java 1.5 compiler\(^2\), the above code produces the following output:

Compilation.java:139: connect(Node) in ANode<Node,Edge> cannot be applied to (ColouredNode)
        n1.connect(cn2); // static type error
Compilation.java:140: connect(SpottedNode) in ANode<SpottedNode,SpottedEdge> cannot be applied to (ColouredNode)
        sn1.connect(cn1); // static type error

2 errors

i.e. we are statically prevented from joining a Node to a ColouredNode.

We might reasonably question whether generics can be encoded in CONCORD. We encode the example of a list in Listing 6.1. The definition of group AList represents an abstract list. The class AList.TheList defines the actual machinery behind maintenance of the list and items in the list. The equivalent such definition in Java generics is

\(^2\)http://java.sun.com/j2se/1.5.0/
6. Comparisons

would see us defining \texttt{class List<T>}, where \( T \) is a type parameter used to represent the type of items to be contained in the list. Now, in generics, we would instantiate particular lists via declarations of the form \texttt{List<String>}. The equivalent form in CONCORD is to define a subgroup of the abstract list group such that the class \texttt{Item} is defined to extend the class of items we wish to store in the list. In Listing 6.1, we define a list of integers.

In conclusion, an encoding of the graph example has demonstrated the involved process of parameterisation required in JAVA generics. In fact, Michalitsis proves that the complexity of this approach is \( O(n^2) \). We can clearly see how poorly the approach scales when considered in the light of families of \( n \) members, where \( n \) is any number larger than two. Contrast the relative simplicity of the \texttt{MyGrp} keyword requiring no parameterisation of group or class definitions.

6.2 Java inner classes

We highlight the main features of JAVA’s inner classes in Subsection 2.6.1. Although the lexical positioning of inner classes in JAVA affords some degree of “belonging”, JAVA inner classes are unable to relatively refer to siblings contained in the same class. In developing specialisations of a concept, this relative reference is essential to ensuring that we never mix members from different families (the graph example of Chapter 1). CONCORD makes specific provision for this fact via the \texttt{MyGrp} keyword.

Another notable difference is that JAVA inner classes are contained inside real classes. This allows instances of the inner classes to hold references to objects of the enclosing class. In further work, we plan to consolidate the concept of groups and classes thereby making this possible in CONCORD: see Chapter 7 for more details.
Distinguishing subtypes

This comparison would never have existed but for the alert mind of an anonymous reviewer of our paper at *FTfJP 2004*. Consider the following Java code snippet:

```java
class Node {
    Edge connect(Node n){ ... }
}

class ColouredNode extends Node {
    Colour c;
}
```

In Java (as we have noted in Section 3.6), the introduction of subclasses introduces subtypes. Our reader questioned whether removing this implied introduction would solve the graph problem, i.e. if we selectively choose which subclass relationships generate subtype relationships, are we home and dry? In answer to this question, we provide a slightly extended version of the graph problem encoded in CONCORD:

```java
group Graph {
    class Node {
        MyGrp.Edge connect(MyGrp.Node x) { ... }
    }

class Edge {
    MyGrp.Node end1; MyGrp.Node end2
    }
}

group ColouredGraph << Graph {
    class Node {
        Colour.Colour c
    }

class Edge {
    Colour.Colour c
    }
}
```

Our coloured graphs now not only contain coloured nodes but also coloured edges. Ideally, therefore, a call to the method `connect()` via a coloured node receiver passing a coloured node parameter will return a coloured edge — perhaps the colour of the edge might be initialised to the colour achieved by mixing the colours of the two nodes\(^3\). However, even if we selectively remove the subtype relationship between objects of classes `Node` and `ColouredNode` in the Java encoding, the object returned from a call to the method `connect()` returns the less precise type `Node` (instead of `ColouredNode`). Whilst

---

\(^3\) We ignore issues of correct initialisation here: they are trivially addressed via parameterless constructor-like methods that are redefined in the subclass to handle the extended state of coloured nodes.
this problem could be circumvented with suitable type casts, we are required to cover this additional ground when using any instance of a specialised node in specialisations of the graph concept.

6.4 **MyType** and **ThisClass**

The notion of “the type of self” is not new. Examples include SATHER [38], EIFFEL [42, 43] and the work of Bruce and Vanderwaart in [12]. The notion exists in many guises including *MyType* and *ThisClass*; we consider the use of *ThisClass* in this Section.

We notice that relative types in CONCORD are more powerful than references to the type of self. The reference *ThisClass* is a direct reference to a class, specifically the class of which the current object is an instance. Our references are therefore restricted to instances of the same class, i.e. *ThisClass*; although an obvious statement to make, the reference *ThisClass* introduces a degree of dependency between fellow objects of a class. In the consideration of binary methods, such restrictions may be powerful enough. But, as we note in Chapter 1 and Chapter 2, we often find situations where dependencies exist between numerous classes — we elucidate this point via the graph example. Dependencies exists between classes of the same family, classes that notionally “belong together”. The reference *ThisClass* defines families with only one member, the class of the current object. The additional layer of abstraction afforded by groups (an encoding of this family concept) allows references to classes of the same family. Groups allow us to define “families” of arbitrary size; the *MyGrp* reference allows us to introduce intra-group (or intra-family) dependencies that persist in specialisations of the family concept (e.g. coloured graphs in our motivational example).

6.5 **Scala**

*Very regrettably, time did not allow allow for an in depth comparison with the work of SCALA [37, 62]. We instead present a short précis of the contributions of the SCALA language (with thanks to Dr Sophia Drossopoulou).*

SCALA [37] combines functional and object oriented programming and contains several features supporting code reuse. It is more powerful than CONCORD; types may contain both intra-group references and references to an object’s identity. The latter, not supported by CONCORD, allows distinct types *graph1.Edge* and *graph2.Edge*, where *graph1* and *graph2* are variables of a *Graph* type, thus forbidding mixing components from two different graph objects even if those objects have the same type. [37] offers an implementation of SCALA with extensive accompanying documentation: a formalisation via the *νOBJ* calculus, including a proof of type system soundness, is presented in [48]. The correspondence between *νOBJ* and SCALA is, however, not immediate. It is unclear whether subtyping is decidable, not necessarily due to SCALA’s treatment of dependent types.
Chapter 7

Conclusion

In this, our final Chapter, we review the work presented in this report, discuss difficulties encountered in the development of the CONCORD calculus and suggest the potential for future development. We conclude with some personal remarks on various aspects of the project.

7.1 This Project’s Contribution

In this project we have presented the calculus CONCORD, a simple model for a restricted form of dependent types in object oriented languages. We introduce the notion of a group which, informally, may be used to identify families of related classes. The concept of a relative type is introduced via the keyword MyGrp; it allows classes, defined within groups, to reference sibling classes within the same defining group. Relative types, equally termed intra-group (or intra-family) relationships, thus introduce dependency into our system. Absolute types reference a group and class name and thus express inter-group (or inter-family) relationships. Specialisation of a family concept is achieved by group extension such that intra-group relations, dependencies and type relationships are preserved. The formal system underpinning CONCORD is examined in great detail with discussion supplemented by intuitive examples and arguments exploring motivational reasoning.

We have demonstrated the lemmata and theorems required to demonstrate soundness of the CONCORD type system, presenting a subset of proofs to these lemmata.

We have presented well known examples encoded in CONCORD to demonstrate the expressive power of our calculus and its suitability in light of our goals. In comparing our calculus with prior work, we have sought to distinguish CONCORD from other solutions addressing collaborations of classes, highlighting problems solved by each approach.

7.2 Difficulties

The development of CONCORD posed the following challenges:
7. Conclusion

The meaning of relative types The exact meaning of relative types, in the broader picture of families of classes, is still not entirely clear. The notion that relative types are intra-family relationships is an appealing one, and one which, when combined with the “inheritance” of these relationships in subsequent specialisations of a family concept (through group extension and further binding), seems to map well onto our human understanding of such problems.

Differences between absolute and relative superclasses Again, the full implications of this property are quite unclear. In contrast to relative types (dependent relationships), relative subclassing does not offer an immediately intuitive concept that can be related to our human understanding. Needless to say, the capabilities of our approach are likely to become considerably clearer when groups and classes are consolidated within the calculus and support for nested groups is provided.

Relative types for this and subexpressions Not immediately obvious, but absolutely essential when considered in light of accessing fields and methods of relative types: the concept of “making relevant” (wording of Section 3.6 when discussing field lookup) is not an intuitive one.

7.3 Further Work

As we later develop in Section 7.4, Concord is a first attempt at formalising the concept of dependencies between collaborations of classes. However, it only allows for static management of such families and must therefore be considered only an excursion into the realm of collaboration based programming and not a complete solution.

Correspondingly, the potential for development of this work is considerable and varied: we briefly discuss possible avenues for further research below:

Nested groups If the concept of a group enclosing classes encodes the idea of a family of related concepts, it seems natural that we extend the language to support families of families of related concepts. We have elucidated the need to consider programming at a coarser level than the single-class perspective, so why stop there? Just as classes, via intra-group relationships, may depend on classes of the same family in a relative fashion (see Figure 7.1(a)), we should allow classes to depend on classes from other families in a relative fashion. This is rather clumsily demonstrated in Figure 7.1(b). The relative inter-group relationship is simply an extension of our intra-group relationship between classes: one might possibly imagine such references being achieved through types of the form $\text{MyGrp} . \text{MyGrp} . \text{A}$.

Comparison with related work Time constraints have prohibited necessary comparison with previous, similar work, e.g. Sather [38], Eiffel [42, 43] and Scala [37].

\footnote{Notice, that in both Figure 7.1(a) and Figure 7.1(b) we redraw relative references between classes even in subgroups where such references would be inherited by further binding. We include these extra arrows to aid our explanation.}
Further Work

Figure 7.1 — Contrasting the current representation of groups with a proposal for nested groups

(a) The current group situation
(b) A possible representation of nested groups

Unify concept of group and class This is inextricably linked to nested groups. By allowing nested class definitions, contained classes may then hold references to objects of the enclosing class, much like JAVA inner classes [54].

Develop dynamic containers Given the original intentions of this project (see Section 7.4), reinvestigation of previous work on dynamic containers [3, 2] in light of conclusions from our work with CONCORD will hopefully provide new insight into the power of the dynamic approach.

Method overloading This is an obvious extension to CONCORD but one that can of course be encoded using similarly named methods.

Delegation Extend CONCORD to consider ideas of delegation presented in [49].

Map Concord onto Java 1.5 In a similar vein to our encoding of the graph problem in GJ (Section 6.1), map CONCORD onto JAVA 1.5 (JAVA 1.5 has officially superseded work on GJ).

Methods parameterised by groups As per suggestion at the end of Section 5.1.

Implementation Two possible approaches to this work are interpreter based evaluation and pre-processor style translation. Taking the interpreter based approach, one might chose to adopt technologies from either JAVA or C/C++ (this should not be considered a fixed domain of languages). With C/C++, FLEX and YACC/BISON are good starting points. FLEX\(^2\) is a lexical analyser generator used to split source code into tokens as defined by a syntax. YACC\(^3\) and BISON\(^4\) are parser generators; they discover the structure of source code based on a grammar, generating an abstract syntax tree in the process. This abstract syntax

\(^2\)http://www.gnu.org/software/flex/
\(^3\)http://dinosaur.compilertools.net/yacc/
\(^4\)http://www.gnu.org/software/bison/bison.html
tree can either be interpreted in place, or translated to another language. The equivalent tools in Java are JFlex\textsuperscript{5} and CUP\textsuperscript{6}. Alternatively, a pre-processor style translation may be taken.

7.4 Execution of our Plan

Discussion in this Section should be read with reference to the outsourcing report in [34], the specification document for this project.

Broadly speaking, the initial goals of this project were:

- Develop a calculus supporting virtual classes and dependent types
- Implement this calculus
- Extend $\mathcal{L}_2$ to support virtual classes and dependent types

In the early stages, we very much focussed on virtual classes and dependent types in light of then recent papers by Anderson et al., specifically [3, 2, 49]. It was envisaged, at least in [34], that we might use the $\mathcal{VF}$ calculus as a basis (or in its entirety) for our virtual class calculus. However, the powerful implications of the calculus developed by Anderson et al. were not immediately obvious, hence it was decided that we first define a simple, static calculus in order to provide some insight into the potential for the wider dynamic calculus.

The development of the calculus progressed well during the spring term of 2004 (the bulk of the formal system was completed by late March) and so, following a suggestion by Dr. Sophia Drossopoulou, it was decided that a paper ([28]) on the subject should be written and submitted to FTfJP 2004\textsuperscript{7}, a workshop with the objective of bringing together people working on formal techniques and tool support for Java-like languages.

When the paper was accepted in late April (mid exams), the focus of the project was permanently altered. With acceptance came the need to rewrite the paper in accordance with comments made by the paper’s anonymous referees. The submission of the modified paper ([30]) was completed at the end of May. In preparation for the workshop itself, a presentation on the paper ([29]) was produced. Writing of the report you currently hold began, in earnest, early June. This schedule of events clearly left no room for completion of either an implementation or an extension to $\mathcal{L}_2$.

7.5 Project Management

This project has been managed via the project website at http://myitcv.org.uk/projects/vj/. The website is a public access point for all things project related, in-

\textsuperscript{5}http://jflex.de/
\textsuperscript{6}http://www.cs.princeton.edu/~appel/modern/java/CUP/
\textsuperscript{7}http://www.cs.kun.nl/~erikpoll/ftfjp/2004.html
including: the \texttt{BibTeX} bibliography, meeting minutes and pre-meeting notes, intermediate papers\textsuperscript{8}, published papers and required reports.

The benefits of maintaining a comprehensive and up-to-date resource are matched by equally significant drawbacks. Unquestionably, the availability of an online resource for all matters project related is of massive help to the communication between project student and supervisor. Indeed, it may also be of help to other students in need of inspiration! But, where such infrastructure does not exist, one is forced to implement efficient methods to make the process worthwhile. The line between a maintaining useful resource and needless micro-managing is a thin one: for example, meeting minutes and pre-meeting notes take time to prepare and where presented material is mathematical in nature, typesetting can be a time consuming process. However, I feel the correct balance was achieved with this project to the extent that I would recommend the approach to other students.

\section{Personal Comment and Conclusions}

The concerned reader might detect a slightly negative tone in the presentation of Section 7.4: this is unintentional but perhaps unavoidable when one considers the completion rate of initial objectives. Unquestionably, the decision to submit a paper to \textit{FTjJP 2004} was exactly the right one, however, this is not to say the decision was made lightly. Designing and coding an implementation is a necessary skill, especially for a student studying towards a Master’s in Engineering. It was with mixed emotions therefore that I turned my back on a project that included some hard-core coding.

But, even with hindsight, I would make exactly the same decision again. It is difficult to describe my feelings at the time our paper was accepted (no doubt due in large part to the then impending threat of exams), but they can perhaps be summed up by the topical\textsuperscript{9} cliché “over the moon”.

The skill of writing and preparing a technical paper is a fantastic asset, but a task that I had honestly underestimated. The necessary attention to technical detail reveals the acute need for care in presentation using well formed and well considered language (i.e. good English sentences). Paper deadlines are nerve-racking times; I would never wish them upon anyone!

On a more technical note, I have learnt the importance of formal systems and proofs in the development of new concepts: even during the latter stages of the calculus’ development, detailed proofs revealed corner cases previously unnoticed. But more importantly, the formal system and accompanying proofs helped develop valuable intuition behind the broader concept of families of classes.

\textsuperscript{8}Papers or reports not required for fulfilment of the MEng project scheme but considered significant enough to warrant individual publication.

\textsuperscript{9}http://www.euro2004.com/
7. Conclusion

7.7 Name silliness

Naming the calculus proved a surprisingly protracted but nonetheless lighthearted and fun process. In early development of [28], the name “Static Groups” was adopted, a rather 'vanilla' opening if you will, but a safe bet. Outlandish suggestions like “Passion”\textsuperscript{10} were mercifully sidelined in favour of the more conservative “Clan”. But with the passing of an era\textsuperscript{11}, we thought it only right and proper to get one over on the French\textsuperscript{12} and hence decided on CONCORD (note the missing ‘e’).

\textsuperscript{10}http://www.thepassionofthechrist.com/
\textsuperscript{11}http://www.britishairways.com/concorde/index.html
\textsuperscript{12}See http://news.bbc.co.uk/sport1/hi/football/euro_2004/3787491.stm and related sites for more tales of anguish
Bibliography


[19] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.


LAMP/EPFL. SCALA website. URL http://scala.epfl.ch/.

C.-C. Lim and A. Stolke. SATHER. URL http://www.icsi.berkeley.edu/~sather/.


Appendix A

Auxiliary Definitions and Diagrammatic Notation

A.1 Definition of $\oplus$

Given two functions $f, g : A \to B$ for any sets $A$ and $B$ we define $f \oplus g : A \to B$ as follows:

$$f \oplus g(a) = \begin{cases} f(a) & \text{if } f(a) \neq \mathbb{U} \\ g(a) & \text{otherwise} \end{cases}$$

A.2 An Explanation of Diagrammatic Notation

We present a legend of our diagrammatic notation in Figure A.1 to complement the description of the CONCORD syntax and presentation of the running example in Chapter 3. For ease of understanding and reference, we now provide minimal examples of each of the concepts shown in the legend.

**Group** Groups definitions constitute programs:

```plaintext
group g1 {
    // group definition
}
```

**Class** Classes are defined inside groups:

```plaintext
group g1 {
    class c1 {
        // class definition
    }
}
```
A. Auxiliary Definitions and Diagrammatic Notation

**Absolute subclass**  Here, class \( g_1.c_1 \) is defined as an absolute subclass of class \( g_2.c_2 \):

```java
group g1 {
    class c1 <: g2.c2 {
        // class definition
    }
}
group g2 {
    class c2 { ... }
}
```

We therefore use the symbol \(<:\) to represent the subclass (and subtype) relationship between classes.

**Relative subclass**  Here, class \( c_1 \) is defined as a relative subclass of class \( c_2 \) in the context of group \( g_1 \):

```java
group g1 {
    class c1 <: MyGrp.c2 {
        // class definition
    }
    class c2 { ... }
}
```

**Implementation inheritance**  This notation is not applicable to CONCORD but can be considered equivalent to further bind in meaning and operation. Its inclusion is required in discussion, amongst other things, of the \( \forall \mathcal{J} \) calculus (Chapter 2) in order that we might distinguish between subtypes and implementation inheritance.

**Further bind**  Here, class \( g_1.c_1 \) is further bound in group \( g_2 \) by class \( g_2.c_1 \) (because group \( g_2 \) is a subgroup of group \( g_1 \)):

```java
group g1 {
    class c1 {
        // class definition
    }
}
group g2 << g1 {
    class c1 {
        // extended class definition
    }
}
```

**Subgroup**  Here, group \( g_2 \) is defined as a subgroup of group \( g_1 \):

```java
group g1 {
    // group definition
}
group g2 << g1 {
    // extended class definition
}
```
// subgroup definition
}

We therefore use the symbol \textless to represent group extension and, correspondingly, the subgroup relation between groups.

\textbf{Absolute association}  Absolute association refers to fields defined by an absolute type. Here, an absolute association exists between field $f_1$ (of class $g_1.c_1$) and class $g_2.c_2$:

```plaintext

group g1 {
    class c1 {
        c2.g2 f1
    }
}
group g2 {
    class c2 { ... }
}
```

\textbf{Relative association}  Relative association refers to fields defined by a relative type. Here, a relative association exists between field $f_1$ (of class $c_1$) and class $c_2$ in the context of group $g_1$:

```plaintext

group g1 {
    class c1 {
        MyGrp.g2 f1
    }
    class c2 { ... }
}
```
A. Auxiliary Definitions and Diagrammatic Notation

Figure A.1 — Notation