

# Simple Dependent Types: Concord

— work in progress —

Paul Jolly, Sophia Drossopoulou,  
Christopher Anderson, Klaus Ostermann

## Abstract

We suggest a simple model for a restricted form of dependent types in object oriented languages, whereby classes belong to groups and dependency is introduced via intra-group references using the `MyGrp` keyword. We show how our approach can code well-known examples from the literature, present the formal model and outline soundness of the type system.

## 1 Introduction and Motivation

Most commercial object oriented languages do not directly support code re-use combined with the expression of dependencies between classes.

Consider, for example, the familiar graph example: regular graphs comprise edges connecting nodes and coloured graphs comprise edges connecting coloured nodes. This can be expressed through classes `Node`, `Edge` and `ColouredNode`, with some form of code reuse between classes `Node` and `ColouredNode`. A method `Edge connect(Node x)` in class `Node` would create an edge between receiver and argument. Our intention is that a receiver of type `Node` is forbidden from calling `connect` with an argument of type `ColouredNode`. However, conventional languages achieve code reuse through subclassing and subclass implies subtype. 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 condition.

Solutions to the above problem have already been suggested through family polymorphism [6], in the programming language SCALA [9], and in [4]. In this paper we present CONCORD, a simple approach to the problem, inspired by ideas from [4], less powerful than SCALA. As far as we know, ours is the first work that combines a *simple* solution, an imperative model, a decidable system, and a sketch-proof of soundness.

In CONCORD, classes belong to *groups*. Groups may be extended by subgroups; the subgroup is said to extend the supergroup. In a subgroup, a class *further binds* its namesake in the supergroup. Further binding implies inheritance but not subtyping. Classes may also extend other classes, giving rise to subclasses. Subclasses imply subtyping. Types comprise a group reference and a class name. The group reference `MyGrp` refers to the group enclosing the current class; it expresses intra-group dependencies, and thus a restricted form of dependent types. For example:

<code>prog</code>	<code>::= group*</code>
<code>group</code>	<code>::= group g &lt;&lt; g { class* }</code>
<code>class</code>	<code>::= class c &lt;: type { field* meth* }</code>
<code>field</code>	<code>::= type f</code>
<code>meth</code>	<code>::= type m (type x) { exp }</code>
<code>exp</code>	<code>::= null   new type   this   exp.f   exp.m(exp)   exp.f = exp</code>
<code>type</code>	<code>::= gr.c</code>
<code>gr</code>	<code>::= g   MyGrp</code>
<code>ta</code>	<code>::= g.c</code>

Fig. 1 — Modified syntax for static groups

```

group Graph {
  class Node {
    MyGrp.Edge connect(MyGrp.Node x){ ... }
  }
  class Edge { ... }
}
group ColouredGraph << Graph {
  class Node {
    colour c
  }
}

```

The group `Graph` contains classes `Edge` and `Node`; the latter defines a method `MyGrp.Edge connect(MyGrp.Node x)`. Group `ColouredGraph` extends `Graph` which means that it contains implicit definitions of classes `Edge` and `Node`, further bindings those definitions in `Graph` [11]. Because `ColouredGraph.Node` further binds `Graph.Node`, it inherits members from but is *not* a subtype of the latter. Therefore, if `n`, `n1` have type `Graph.Node`, and `cn`, `cn1` have type `ColouredGraph.Node`, then the terms `n.connect(n1)` and `cn.connect(cn1)` are type correct, whereas `n.connect(cn)` and `cn.connect(n)` are type incorrect.

The rest of the paper is organised as follows: In Section 2 we give the syntax of CONCORD and a running example, in Section 3 we define inheritance, in Section 4 we define the operational semantics, in Section 5 we give the type system, in Section 6 we outline the proof of soundness, and in Section 7 we conclude. In the appendix we give the more straightforward definitions and further examples.

## 2 Syntax

Fig. 1 contains the syntax of CONCORD, where `g`, `c`, `f` and `m` stand for group, class, field and method identifiers respectively.

A CONCORD program is a sequence of group definitions, which, in turn, consist of class definitions. Classes are similar to those in JAVA or C#, with the difference that CONCORD types consist of a group reference (`gr`) and a class identifier (`c`). The group reference may be a fixed group e.g. `g1` (thus introducing an *absolute* type, called `ta` in Fig. 1), or `MyGrp`, which refers to the context's group (thus introducing a *relative* type, which changes in subgroups).

$\frac{\begin{array}{l} \vdash P \diamond_u \\ P = \dots \text{group } g \ll g' \{ \dots \} \dots \end{array}}{\begin{array}{l} \vdash g \ll g \\ \vdash g \ll g' \end{array}}$	$\frac{\begin{array}{l} \vdash g \ll g' \\ \vdash g' \ll g'' \end{array}}{\vdash g \ll g''}$
---	--

**Fig. 2** — Definition of subgroups

Every group extends another group, and every class extends another class, with `GlobalGroup` and `GlobalGroup.c` at the top of the hierarchy. Groups cannot be nested; this will be considered in further work.

The CONCORD program given below will serve as our running example<sup>1</sup>.

```

group g1 { class D { MyGrp.D f3 } }
group g2 {
  class A {
    g2.A f1
    MyGrp.A f2
    MyGrp.A m1(MyGrp.A x) { new MyGrp.A }
    g2.A m2(g2.A x) { x }
  }
  class B <: MyGrp.A { ... }
  class C <: g1.A { ... }
}
group g3 << g2 {
  class C <: g1.D {
    g2.A m3(g3.B x) {
      this.f2 = new g2.A // type correct
      this.f2 = new MyGrp.A // type error!
    }
  }
}

```

### 3 Inheritance

A CONCORD class can inherit members from another class in two ways: Firstly, if `group g << g' { ... }`, then `g` is a *subgroup* of `g'` (see Fig. 2 for a definition of subgroups) and any class `g.c` *further binds* the class `g'.c` if `g'.c` is defined. Secondly, if `group g << ... { ... class c <: gr.c' { ... } }`, then `g.c` is a *subclass* of `gr.c'`. Further binding implies inheritance of members but not subtyping, while subclassing implies inheritance of members and subtyping.

Inheritance of members through subclasses and further binding is defined in Fig. 3<sup>2</sup>. The function  $\mathcal{F}(g.c, f)$  returns the type of field `f` in class `g.c`, while  $\mathcal{M}(g.c, m)$  returns the signature and body of method `m` in class `g.c`.

Inheritance through further binding “copies” all members into the subgroup, e.g. if `c` is defined within `g`, and `g` is a direct subgroup of `g'`, then  $\mathcal{F}(g.c, f) = \dots \oplus \dots \oplus \mathcal{F}(g'.c, f)$ .

<sup>1</sup>As in JAVA, we drop the supergroup and superclass when these are `GlobalGroup` or `GlobalGroup.c`.

<sup>2</sup>See Appendix B for the definition of the class lookup function  $\mathcal{C}(g.c)$  and  $\oplus$

$$\begin{array}{c}
\frac{\mathcal{C}(g.c) = \text{class } c <: \dots \{ \dots t \ m(t_1 \ x) \{ e \} \dots \}}{\mathcal{MW}(g.c, m) = t \ m(t_1 \ x) \{ e \}} \\
\\
\frac{\mathcal{MW}(\text{GlobalGroup}.c, m) = \mathcal{Ulf}}{\mathcal{G}(g) = \text{group } g \ll g' \{ \dots \}} \\
\frac{\mathcal{C}(g.c) = \text{class } c <: \text{gr}.c' \{ \dots \}}{\mathcal{M}(g.c, m) = \mathcal{MW}(g.c, m) \oplus \mathcal{M}(\text{gr}[g].c', m)[\text{gr}] \oplus \mathcal{M}(g'.c, m)} \\
\\
\frac{\mathcal{C}(g.c) = \text{class } c <: \dots \{ \dots t \ f \dots \}}{\mathcal{FW}(g.c, f) = t} \quad \frac{\mathcal{FW}(\text{GlobalGroup}.c, f) = \mathcal{Ulf}}{\mathcal{G}(g) = \text{group } g \ll g' \{ \dots \}} \\
\frac{\mathcal{C}(g.c) = \text{class } c <: \text{gr}.c' \{ \dots \}}{\mathcal{F}(g.c, f) = \mathcal{FW}(g.c, f) \oplus \mathcal{F}(\text{gr}[g].c', f)[\text{gr}] \oplus \mathcal{F}(g'.c, f)} \\
\\
\frac{\mathcal{F}(g.c, f) = \mathcal{Ulf}}{\mathcal{F}(g.c) = \{ f \mid \mathcal{F}(g.c, f) \neq \mathcal{Ulf} \}}
\end{array}$$

**Fig. 3** — Method and field lookup functions

Inheritance through subclassing is more intricate: Any intra-group reference in the superclass name is replaced by the current group, and any intra-group reference within the member is replaced by the group reference of the superclass name. That is, if  $c$  is defined within  $g$  as a subclass of  $\text{gr}.c'$  then  $\mathcal{F}(g.c, f) = \dots \oplus \mathcal{F}(\text{gr}[g].c', f)[\text{gr}] \oplus \dots$ . Thus:

**subclassing — fields**

$$\begin{aligned}
\mathcal{F}(g2.A, f1) &= \mathcal{F}(g2.B, f1) = \mathcal{F}(g2.C, f1) = g2.A \\
\mathcal{F}(g2.A, f2) &= \mathcal{F}(g2.B, f2) = \text{MyGrp}.A \\
\mathcal{F}(g2.C, f2) &= g2.A \\
\mathcal{F}(g3.C, f3) &= g1.D
\end{aligned}$$

**further binding — fields**

$$\begin{aligned}
\mathcal{F}(g3.A, f1) &= \mathcal{F}(g3.B, f1) = \mathcal{F}(g3.C, f1) = g2.A \\
\mathcal{F}(g3.A, f2) &= \mathcal{F}(g3.B, f2) = \text{MyGrp}.A \\
\mathcal{F}(g3.C, f2) &= g2.A
\end{aligned}$$

**subclassing — methods**

$$\begin{aligned}
\mathcal{M}(g2.A, m1) &= \mathcal{M}(g2.B, m1) = \text{MyGrp}.A \ m1 \ (\text{MyGrp}.A \ x) \ \{ \text{new MyGrp}.A \} \\
\mathcal{M}(g2.C, m1) &= g2.A \ m1 \ (g2.A \ x) \ \{ \text{new } g2.A \}
\end{aligned}$$

Fields and methods are inherited by member classes in subgroups:

**Lemma 1**    *Given*  $\vdash g' \ll g$ :

1.  $\mathcal{F}(g.c, f) = t \implies \mathcal{F}(g'.c, f) = t$
2.  $\mathcal{M}(g.c, m) = t_0 \ m(t_1 \ x) \ \{ \dots \} \implies \mathcal{M}(g'.c, m) = t_0 \ m(t_1 \ x) \ \{ \dots \}$

		stack	heap
$store$	$=$	$\overbrace{(\{\mathbf{this}\} \mapsto addr) \cup (\{x\} \mapsto addr)}$	$\cup \overbrace{(addr \mapsto object)}$
$val$	$=$	$\{\mathbf{null}\} \cup addr$	
$dev$	$=$	$\{\mathbf{nullPtrExc}\}$	
$object$	$=$	$\{\llbracket \mathbf{g} . \mathbf{c} \parallel \mathbf{f}_i : \mathbf{v}_i^{i \in 1 \dots n} \rrbracket \mid \mathbf{f}_i, \mathbf{g}, \mathbf{c} \text{ identifiers, } \mathbf{v}_i \in val\}$	
$addr$	$=$	$\{\iota_i \mid i \in \mathbb{Z}^*\}$	
$o(\mathbf{f})$	$=$	$\begin{cases} \mathbf{v}_l & \text{if } \mathbf{f} = \mathbf{f}_l \mid l \in 1, \dots, r \\ \mathcal{U}f & \text{otherwise} \end{cases}$	
$o[\mathbf{f} \mapsto \mathbf{v}]$	$=$	$\llbracket \mathbf{g} . \mathbf{c} \parallel \mathbf{f}_1 : \mathbf{v}_1 \dots \mathbf{f}_l : \mathbf{v} \dots \mathbf{f}_r : \mathbf{v}_r \rrbracket$	if $\exists l \in 1, \dots, r \mid \mathbf{f} = \mathbf{f}_l$
$\sigma[z \mapsto \mathbf{v}](z)$	$=$	$\mathbf{v}$	
$\sigma[z \mapsto \mathbf{v}](z')$	$=$	$\sigma(z')$	if $z' \neq z$

**Fig. 4** — Stores of and operations on objects  $o$ ; store  $\sigma$  and identifier or address  $z$ .

## 4 Execution

We define execution in terms of large step semantics, where an expression and store is 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 contain their runtime type (or absolute type,  $\mathbf{ta}$ ) and the values of their fields. This can be seen in Fig. 4.

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

$$\mathcal{MyGrp}(\sigma) = \mathbf{g} \text{ where } \sigma(\sigma(\mathbf{this})) = \llbracket \mathbf{g} . \mathbf{c} \parallel \dots \rrbracket$$

In Fig. 5 we give the operational semantics. All rules are straightforward, and similar, e.g. to those in [5], except for 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, if the current receiver has runtime type  $\mathbf{g3} . \mathbf{A}$ , then execution of `new MyGrp.C` will create an object of dynamic type  $\mathbf{g3} . \mathbf{C}$ .

## 5 Types

Expressions are typed in the context of an environment,  $\Gamma$ , which gives types to the receiver, **this**, and the method's parameter,  $x$ .  $\Gamma(id)$  returns the type of  $id$  in  $\Gamma$ . For  $\Gamma = \mathbf{t} \ x, \mathbf{g} . \mathbf{c} \ \mathbf{this}$ , we define  $\Gamma(id)$  to be  $\mathbf{t}$  if  $id = x$ ,  $\mathbf{g}$  if  $id = \mathbf{MyGrp}$ ,  $\mathbf{MyGrp} . \mathbf{c}$  if  $id = \mathbf{this}$  and  $\mathcal{U}f$  otherwise.  $\Gamma(\mathbf{this})$  always has the form  $\mathbf{MyGrp} . \mathbf{c}$ .

The functions  $\mathcal{MyGrp}(\mathbf{t})$  and  $\mathcal{MyGrp}(\Gamma)$  extract the group of the type  $\mathbf{t}$ , or of the receiver of  $\Gamma$ . The operations  $\mathbf{e}[\mathbf{g}]$  and  $\mathbf{t}[\mathbf{t}']$  replace any occurrence of **MyGrp** in an expression and type respectively.

$$\begin{aligned}
\mathcal{MyGrp}(\Gamma) &= \Gamma(\mathbf{MyGrp}) \\
\mathcal{MyGrp}(\mathbf{t}) &= \mathbf{gr} \text{ where } \mathbf{t} = \mathbf{gr} . \mathbf{c} \\
\mathbf{e}[\mathbf{g}] &= \mathbf{e}[\mathbf{g} / \mathbf{MyGrp}] \\
\mathbf{t}[\Gamma] &= \mathbf{t}[\mathcal{MyGrp}(\Gamma) / \mathbf{MyGrp}] \\
\mathbf{t}[\sigma] &= \mathbf{t}[\mathcal{MyGrp}(\sigma) / \mathbf{MyGrp}] \\
\mathbf{t}[\mathbf{t}'] &= \mathbf{t}[\mathcal{MyGrp}(\mathbf{t}') / \mathbf{MyGrp}] \\
(\mathbf{t} \ \mathbf{m} \ (\mathbf{t}' \ \mathbf{x}) \ \{\mathbf{e}\})[\mathbf{g}] &= \mathbf{t}[\mathbf{g}] \ \mathbf{m} \ (\mathbf{t}'[\mathbf{g}] \ \mathbf{x}) \ \{\mathbf{e}[\mathbf{g}]\}
\end{aligned}$$

$\frac{}{v, \sigma \rightsquigarrow v, \sigma} \text{ (val)}$	$\frac{}{x, \sigma \rightsquigarrow \sigma(x), \sigma} \text{ (var)}$ $\text{this}, \sigma \rightsquigarrow \sigma(\text{this}), \sigma$
$\frac{\begin{array}{l} e, \sigma \rightsquigarrow \iota, \sigma'' \\ e', \sigma'' \rightsquigarrow v, \sigma''' \\ \sigma'''(\iota)(f) \neq \mathcal{U}lf \\ \sigma' = \sigma'''[\iota \mapsto \sigma'''(\iota)[f \mapsto v]] \end{array}}{e.f = e', \sigma \rightsquigarrow v, \sigma'} \text{ (fldAss)}$	$\frac{\begin{array}{l} e, \sigma \rightsquigarrow \iota, \sigma' \\ \sigma'(\iota)(f) \neq \mathcal{U}lf \end{array}}{e.f, \sigma \rightsquigarrow \sigma(\iota)(f), \sigma'} \text{ (fld)}$
$\frac{\begin{array}{l} g = \text{gr}[\text{MyGrp}(\sigma)] \\ \mathcal{F}\mathcal{S}(g.c) = \{f_1, \dots, f_r\} \\ \iota \text{ is new} \end{array}}{\text{new gr.c}, \sigma \rightsquigarrow \iota, \sigma[\iota \mapsto \llbracket g.c \parallel f_1 : \text{null}, \dots, f_r : \text{null} \rrbracket]} \text{ (new)}$	
$\frac{\begin{array}{l} e_0, \sigma \rightsquigarrow \iota, \sigma_0 \\ e_1, \sigma_0 \rightsquigarrow v_1, \sigma_1 \\ \sigma_1(\iota) = \llbracket \text{ta} \parallel \dots \rrbracket \\ \mathcal{M}(\text{ta}, m) = t \ m \ (t_1 \ x) \ \{e\} \\ \sigma'' = \sigma_1[\text{this} \mapsto \iota][x \mapsto v_1] \\ e, \sigma'' \rightsquigarrow v, \sigma' \end{array}}{e_0.m(e_1), \sigma \rightsquigarrow v, \sigma'[\text{this} \mapsto \sigma(\text{this}), x \mapsto \sigma(x)]} \text{ (methCall)}$	

**Fig. 5** — Operational semantics. We omit rules for throwing and propagation of exception, because these are standard

Fig. 6 defines the subtype relationship  $g \vdash t <: t'$ , whereby a type  $t$  is a subtype of another type  $t'$  in the context of a certain group  $g$ . The group context is necessary when  $t$  or  $t'$  reference `MyGrp`. For example:

$$\begin{array}{ll} g2 \vdash \text{MyGrp.B} <: \text{MyGrp.A} & g2 \vdash \text{MyGrp.C} <: g2.A \\ g3 \vdash \text{MyGrp.B} <: \text{MyGrp.A} & g3 \vdash \text{MyGrp.C} <: g1.D \\ \vdash g2.B <: g2.A & g3 \vdash \text{MyGrp.C} <: g2.A \\ \vdash g3.B <: g3.A & \\ \not\vdash g3.B <: g2.B & \end{array}$$

We can prove that a subgroup  $g'$  satisfies all subtype relationships from its supertype  $g$ , and that it may replace  $g'$  in any subtype relationship:

**Lemma 2** Take any  $g, g'$  with  $\vdash g' \ll g$ :

1.  $g \vdash t' <: t \implies g' \vdash t' <: t$
2.  $\vdash g.c <: ta \implies \vdash g'.c <: ta$

We can also prove that any absolute type  $g'.c'$  inherits all members from from a supertype  $g.c$ ; the members are inherited unmodified if the two groups  $g$  and  $g'$  are equal, otherwise any occurrence of `MyGrp` is replaced by  $g$ .

**Lemma 3** Take and  $g, c, g', c'$  with  $\vdash g'.c' <: g.c$ :

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

$\frac{\mathcal{C}(g.c) = \text{class } c <: t \{ \dots \}}{g \vdash \text{MyGrp}.c <: t}$	$\frac{g \vdash t <: t'}{g' \vdash t[g] <: t'[g]}$
$\frac{g \vdash t <: t' \quad \vdash g' \ll g}{g' \vdash t <: t'}$	$\frac{g \vdash t <: t'' \quad g \vdash t'' <: t'}{g \vdash t <: t'}$
$\frac{g \vdash ta <: ta'}{\vdash ta <: ta'}$	
$\frac{\vdash \Gamma \diamond}{\Gamma \vdash x : \Gamma(x) \quad \Gamma \vdash \text{this} : \Gamma(\text{this})} \text{ (VarThis)}$	$\frac{\vdash \Gamma \diamond \quad \Gamma \vdash t \diamond_i}{\Gamma \vdash \text{null} : t \quad \Gamma \vdash \text{new } t : t} \text{ (NewNull)}$
$\frac{\Gamma \vdash e : t \quad \text{MyGrp}(\Gamma) \vdash t <: t'}{\Gamma \vdash e : t'} \text{ (Subsump)}$	$\frac{\Gamma \vdash e : t \quad \mathcal{F}(t[\Gamma], f) = t'}{\Gamma \vdash e.f : t'[t]} \text{ (Fld)}$
$\frac{\Gamma \vdash e_0 : t_0 \quad \mathcal{M}(t_0[\Gamma], m) = t \ m(t_1 \ x) \{ \dots \} \quad \Gamma \vdash e_1 : t_1[t_0]}{\Gamma \vdash e_0.m(e_1) : t[t_0]} \text{ (Meth)}$	
$\frac{\Gamma \vdash e : t \quad \mathcal{F}(t[\Gamma], f) = t' \quad \Gamma \vdash e' : t'[t]}{\Gamma \vdash e.f = e' : t'[t]} \text{ (FldAss)}$	

**Fig. 6** — Subtypes and the type system

$$t_0 \ m(t_1 \ x) \{ \dots \} = t \ \text{and} \ g' = g \ \text{or} \\ \mathcal{M}(g'.c', m) = t_0[g] \ m(t_1[g] \ x) \{ \dots \}$$

Fig. 6 also defines the type rules  $\Gamma \vdash e : t$ , whereby an expression  $e$  has type  $t$  in the context of an environment  $\Gamma$ . The rules (VarThis), (NewNull) and (Subsump) are standard. The rule (Fld) is more interesting, because (a) it looks up the field  $f$  in  $t[\Gamma]$ , (through  $t' = \mathcal{F}(t[\Gamma], f)$ ), that is, it finds the absolute type by replacing any occurrence of `MyGrp` by the group where the current method has been defined, and (b) it replaces in  $t'$  any occurrences of `MyGrp` by the group to which the receiver,  $e$ , belongs. For example:

$$\begin{array}{ll} \Gamma_2 = g2.A \ \text{this}, g2.A \ x & \Gamma_2 \vdash x.f2 : g2.A \\ & \Gamma_2 \vdash \text{this}.f2 : \text{MyGrp}.A \\ \Gamma_4 = g3.C \ \text{this}, g3.B \ x & \Gamma_4 \vdash \text{this}.f2 : g2.A \\ & \Gamma_4 \vdash x.f2 : g3.A \\ \Gamma_4 = g3.B \ \text{this} \dots & \Gamma_4 \vdash \text{this}.f2 : \text{MyGrp}.A \end{array}$$

We can prove that expressions preserve their types when typed in a subgroup.

This means that inheritance of methods through further binding preserves the type of the method body.

**Lemma 4**  $\vdash g' \ll g, \Gamma(\text{this}) = g.c$  for some  $c$ ,  $\Gamma' = \Gamma[\text{this} \mapsto g'.c]$ ,  
 $\Gamma \vdash e : t \implies \Gamma' \vdash e : t$

In lemma 5 we prove that if we replace the type of the receiver by a subtype in an environment  $\Gamma$ , and any occurrences of **MyGrp** by the group to which the receiver of  $\Gamma$  belongs in an expression  $e$ , then we obtain the same type as if we replaced all occurrences of **MyGrp** by the group to which the receiver of  $\Gamma$  belongs in the original type of  $e$ . Therefore, inheritance of methods by subtypes preserves the types of the method body modulo the necessary substitutions of **MyGrp**.

**Definition 1**  $\Gamma' <: \Gamma$  iff:  
 $\Gamma = g.c \text{ this}, t \ x$  and  $\Gamma' = g'.c' \text{ this}, t' \ x$  and  
 $(\Gamma' \vdash \text{MyGrp}.c' <: \text{gr}.c$  or  $\Gamma' \vdash t' <: t[g])$  and  $g = \text{gr}[g']$

**Lemma 5**  $\Gamma \vdash e : t, \Gamma' <: \Gamma \implies \Gamma' \vdash e[\Gamma] : t[\Gamma]$

Thus, we can prove that in a well formed program (well formed programs,  $\vdash P \diamond$ , are defined in Appendix C), the body of any method in an absolute type  $\text{ta}$ , even if inherited through subclassing or through further binding, has a type in accordance with its type in  $\text{ta}$ :

**Theorem 1** *In a well formed program:*

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

## 6 Soundness and Decidability

In Fig. 7 we define agreement between addresses and types,  $\sigma \vdash \iota : \text{ta}$ , whereby an address agrees with the runtime type of the corresponding object, or any supertype, and **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  $g.c$  have values which agree with their types in  $g.c$  where **MyGrp** is replaced by  $g$ . A store is well-formed,  $\Gamma \vdash \sigma \diamond$ , if all addresses correspond to well formed objects, and if **this** and **x** contain addresses which agree with their types in  $\Gamma$ .

We can now prove soundness of our type system.

**Theorem 2 (Soundness)**

$$\left. \begin{array}{l} \vdash P \diamond \\ \Gamma \vdash \sigma \diamond \\ \Gamma \vdash e : t \\ e, \sigma \rightsquigarrow \iota, \sigma' \\ e[\Gamma] = e[\sigma] \end{array} \right\} \implies \begin{array}{l} \Gamma' \vdash \sigma \diamond \\ \sigma' \vdash \iota : t[\Gamma] \end{array}$$

We have a hand-written proof of the theorem, through induction on the derivation of the type of  $e$ . The requirement that  $e[\Gamma] = e[\sigma]$  guarantees that either there are no occurrences of **MyGrp** in  $e$ , or that  $\sigma(\text{MyGrp}) = \Gamma(\text{MyGrp})$ .

$$\begin{array}{c}
\frac{\sigma(\iota) = \llbracket \mathbf{ta} \parallel \dots \rrbracket}{\sigma \vdash \iota : \mathbf{ta}} \quad \frac{\sigma \vdash \iota : \mathbf{ta} \quad \vdash \mathbf{ta} <: \mathbf{ta}'}{\sigma \vdash \iota : \mathbf{ta}'} \quad \frac{}{\sigma \vdash \mathbf{null} : \mathbf{ta}} \\
\\
\frac{\sigma(\iota) = \llbracket \mathbf{g} . \mathbf{c} \parallel \dots \rrbracket \quad \mathcal{F}(\mathbf{g} . \mathbf{c}, \mathbf{f}) = \mathbf{t} \implies \sigma \vdash \sigma(\iota)(\mathbf{f}) : \mathbf{t}[\mathbf{g}]}{\sigma \vdash \iota} \\
\\
\frac{\sigma(\iota) \neq \mathcal{U}f \implies \sigma \vdash \iota \quad \sigma \vdash \sigma(\mathbf{x}) : \Gamma(\mathbf{x})[\Gamma] \quad \sigma \vdash \sigma(\mathbf{this}) : \Gamma(\mathbf{this})[\Gamma]}{\Gamma \vdash \sigma \diamond}
\end{array}$$

**Fig. 7** — Agreement between programs, stores and environments

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  $\mathcal{F}$  and  $\mathcal{M}$  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.

## 7 Related and Further Work and Conclusions

As we said earlier, there are many approaches to the expression of relationships across collections of classes. `TypeGroups` were used in [4], combined with matching and `MyType`. The full system has not yet been formalised and proven sound, while a subset, comprising `MyType` (in the form of `ThisClass`) but not `TypeGroups` is presented and proven sound for a FEATHERWEIGHT JAVA extension in [3]. The main difference between `MyGrp` and `ThisClass` is that `MyGrp` refers to the group enclosing the current class whereas `ThisClass` refers to the current class itself. We show an encoding of `ThisClass` in Appendix D.

Families of classes are suggested in [6, 7] mostly through extensions of the language `gbeta`; a formalisation is planned.

SCALA [9] combines functional and object oriented programming and contains several features to support reuse. It is more powerful than CONCORD, as 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 `Graph` type, thus forbidding mixing components from two different graph objects even if those objects have the same type. SCALA has an implementation and extensive documentation, and a formalisation with soundness proof [10] through  $\nu$ OBJ. The correspondence between  $\nu$ OBJ and SCALA is not immediate. It is unclear whether subtyping is decidable, not necessarily due to SCALA's treatment of dependent types.

The problem of relationships across collections of classes can also be addressed through virtual types [12] and its connection with generics has been explored in [13]. The famous “Cow Example” can be seen in Appendix E.

The connection between virtual types and dependent types is explored in [8]; soundness has not yet been demonstrated. The expression problem, posed originally by Reynolds and later suggested by Wadler in the JAVA-genericity mailing list [15], is an example of such dependence across classes. Solutions using virtual types and generics are explored in [14], and using dependent types in SCALA [16]. The expression problem is shown in Appendix F.

Powerful versions of dependent types have been suggested in [1], decidability has not yet been proven.

Thus, we believe that CONCORD is the first work that combines a *simple* solution, an imperative model, a decidable system and a sketch of the proof of soundness.

In the full paper we will compare SCALA’s treatment of the issues addressed by CONCORD and will explore the relationship between CONCORD and [4]. In further work we plan to write up the proofs and consider mapping CONCORD onto GJ [2]. We also want to work on an implementation and explore several extensions of CONCORD: allow any amount of nesting in groups, groups as parameters, the amalgamation of groups and classes, and allow `MyGrp` to refer to an object’s identity.

## References

- [1] C. Anderson and K. Ostermann. VC - foundations for virtual classes with dependent types. URL <http://www.binarylord.com/work/vc.pdf>. Submitted for publication, 2003.
- [2] G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. *GJ: Extending the Java Programming Language with type parameters*, August 1998.
- [3] K. B. Bruce and J. N. Foster. LOOJ: Weaving LOOM into Java. In *ECOOP 2004*, 2004.
- [4] K. B. Bruce and J. C. Vanderwaart. Semantics-driven language design: Statically type-safe virtual types in object-oriented languages. In *Fifteenth Conference on the Mathematical Foundations of Programming Semantics*, 1999.
- [5] S. Drossopoulou, F. Damiani, M. Dezani-Ciancaglini, and P. Giannini. Fickle: Dynamic Object Re-classification. In *ECOOP’01*, volume 2072 of *LNCS*, pages 130–149. Springer-Verlag, 2001.
- [6] E. Ernst. Family polymorphism. In J. L. Knudsen, editor, *ECOOP 2001 – Object-Oriented Programming*, LNCS 2072, pages 303–326, Heidelberg, European Conference in Germany, 2001. Springer-Verlag. ISBN 3-540-42206-4.
- [7] E. Ernst. Higher-order hierarchies. In L. Cardelli, editor, *Proceedings ECOOP 2003*, LNCS 2743, pages 303–329, Heidelberg, Germany, July 2003. Springer-Verlag.

- [8] A. Igarashi and B. C. Pierce. Foundations for virtual types. In *ECOOP 1999*, 1999.
- [9] LAMP/EPFL. Scala website. URL <http://scala.epfl.ch/>.
- [10] M. Odersky, V. Cremet, C. Röckl, and M. Zenger. A Nominal Theory of Objects with Dependent Types. In *Proc. ECOOP'03*, Springer LNCS, July 2003.
- [11] K. Ostermann. Dynamically composable collaborations with delegation layers. In *Proceedings of the 16th European Conference on Object-Oriented Programming (ECOOP)*, Malaga, Spain, 2002.
- [12] K. K. Thorup. Genericity in Java with virtual types. *Lecture Notes in Computer Science*, 1241:444–, 1997.
- [13] K. K. Thorup and M. Torgersen. Unifying genericity — combining the benefits of virtual types and parameterized classes. In *ECOOP 99*. Springer-Verlag, 1999.
- [14] M. Torgersen. The expression problem revisited four new solutions using generics. In *ECOOP 2004*, 2004.
- [15] Various. Java-Genericity email list. List emails collated to form a webpage of links to each correspondance, 1997–2000. URL <http://www.cis.ohio-state.edu/~gb/cis888.07g/java-genericity/>.
- [16] M. Zenger and M. Odersky. Independently extensible solutions to the expression problem. URL <http://scala.epfl.ch/docu/related.html>.

## A Unique definitions

The judgement demanding unique definitions can be seen in Fig. 8. It requires that:

- Group names are unique. Member class names are unique within a group. Groups cannot be nested<sup>3</sup>, hence there can be no naming clash.
- Field and method names are unique within member classes, even where subclassing and further binding is involved.

$$\begin{array}{l}
 \forall g : P = P_1 \text{ group } g \ll \text{sg} \{ \dots \} P_2 \text{ and } P = P_3 \text{ group } g \ll \text{sg}' \{ \dots \} P_4 \\
 \quad \implies P_1 = P_3 \text{ and } P_2 = P_4 \text{ and } \text{sg} = \text{sg}' \\
 \forall c : P = P_1 \text{ group } g \ll \text{sg} \{ \text{gbody}_1 \text{ class } c <: \text{sc} \{ \dots \} \text{gbody}_2 \} P_2 \text{ and} \\
 \quad P = P_1 \text{ group } g \ll \text{sg} \{ \text{gbody}_3 \text{ class } c <: \text{sc}' \{ \dots \} \text{gbody}_4 \} P_2 \\
 \quad \implies \text{gbody}_1 = \text{gbody}_3 \text{ and } \text{gbody}_2 = \text{gbody}_4 \text{ and } \text{sc} = \text{sc}' \\
 \forall f : P = P_1 \text{ group } g \ll \text{sg} \{ \\
 \quad \text{gbody}_1 \text{ class } c <: \text{sc} \{ \text{defs}_1 \text{ t f } \text{defs}_2 \} \text{gbody}_2 \\
 \quad \} P_2 \text{ and} \\
 \quad P = P_1 \text{ group } g \ll \text{sg} \{ \\
 \quad \text{gbody}_1 \text{ class } c <: \text{sc} \{ \text{defs}_3 \text{ t}' \text{ f } \text{defs}_4 \} \text{gbody}_2 \\
 \quad \} P_2 \\
 \quad \implies \text{defs}_1 = \text{defs}_3 \text{ and } \text{defs}_2 = \text{defs}_4 \\
 \forall m : P = P_1 \text{ group } g \ll \text{sg} \{ \\
 \quad \text{gbody}_1 \text{ class } c <: \text{sc} \{ \text{defs}_1 \text{ t m (t}_1 \text{ x) } \{ \text{e} \} \text{defs}_2 \} \text{gbody}_2 \\
 \quad \} P_2 \text{ and} \\
 \quad P = P_1 \text{ group } g \ll \text{sg} \{ \\
 \quad \text{gbody}_1 \text{ class } c <: \text{sc} \{ \text{defs}_3 \text{ t}' \text{ m (t}'_1 \text{ x) } \{ \text{e}' \} \text{defs}_4 \} \text{gbody}_2 \\
 \quad \} P_2 \\
 \quad \implies \text{defs}_1 = \text{defs}_3 \text{ and } \text{defs}_2 = \text{defs}_4 \\
 \hline
 \vdash P \diamond_u \\
 \\
 \forall g, g' : \vdash g \ll g' \text{ and } \vdash g' \ll g \implies g = g' \\
 \hline
 \vdash P \diamond_{ag} \\
 \\
 \forall t, t' : g \vdash t <: t' \text{ and } g \vdash t' <: t \implies t = t' \\
 \hline
 \vdash P \diamond_{at}
 \end{array}$$

Fig. 8 — Unique definitions and acyclic groups and types

<sup>3</sup>The lack of group nesting follows from the syntax

## B Group and Class Lookup Functions

$$\begin{array}{c}
 \frac{P = P' \text{ group } g \ll g' \{ gbody \} P''}{\mathcal{G}(g) = \text{group } g \ll g' \{ gbody \}} \\
 \\
 \frac{\mathcal{G}(g) = \text{group } g \ll sg \{ \dots \text{class } c <: t \{ cbody \} \dots \}}{\mathcal{CW}(g.c) = \text{class } c <: t \{ cbody \}} \\
 \\
 \frac{}{\mathcal{CW}(\text{GlobalGroup}.c) = \mathcal{U}lf} \qquad \frac{\mathcal{CW}(g.c) \neq \mathcal{U}lf}{\mathcal{C}(g.c) = \mathcal{CW}(g.c)} \\
 \\
 \frac{\mathcal{CW}(g.c) = \mathcal{U}lf}{\mathcal{G}(g) = \text{group } g \ll g' \{ \dots \}} \\
 \frac{}{\mathcal{C}(g.c) = \mathcal{C}(g'.c)}
 \end{array}$$

Fig. 9 — Group and class lookup functions.

### B.1 Definition of $\oplus$

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

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

## C Well-formedness

$$\begin{array}{c}
 \frac{\vdash P \diamond_u \quad \mathcal{G}(g) = \text{group } g \ll \dots \{ \dots \} \quad \mathcal{C}(g.c) = \text{class } c <: \dots \{ \dots \}}{\vdash g \diamond_g \quad \vdash g.c \diamond_c \quad \vdash g.c \diamond_t} \qquad \frac{\vdash g.c \diamond_c \quad \Gamma(\text{MyGrp}) = g}{\Gamma \vdash \text{MyGrp}.c \diamond_t} \\
 \\
 \frac{\vdash \Gamma(x) \diamond_t \quad \vdash \Gamma(\text{MyGrp}) \diamond_g \quad \vdash \Gamma(\text{this}) \diamond_c}{\vdash \Gamma \diamond} \qquad \frac{}{\vdash \text{GlobalGroup}.c \diamond_c \quad \vdash \text{GlobalGroup}.c \diamond_t \quad \vdash \text{GlobalGroup} \diamond_g}
 \end{array}$$

Fig. 10 — Well-formed environments

$$\begin{array}{c}
\mathcal{MW}(g.c,m) = \text{t m}(t_1 \text{ x}) \{ e \} \\
\vdash t \diamond_t \\
\vdash t_1 \diamond_t \\
t_1 \text{ x, g.c this} \vdash e : t_{ret} \\
\hline
g \vdash t_{ret} <: t \\
\hline
g.c \vdash m \diamond \\
\\
\mathcal{FW}(g.c,f) = t_0 \implies \vdash t_0 \diamond_t \\
\mathcal{MW}(g.c,m) = \text{t m}(t_1 \text{ x}) \{ e \} \implies g.c \vdash m \diamond \\
\hline
\vdash g.c \diamond_{general} \\
\\
\mathcal{G}(g) = \text{group } g \ll g' \{ \dots \} \text{ and } \mathcal{C}(g.c) = \text{class } c <: \text{gr.c} \{ \dots \} \implies \\
\mathcal{FW}(g.c,f) \neq \mathcal{Ulf} \implies \mathcal{F}(g'.c,f) = \mathcal{Ulf} \\
\mathcal{MW}(g.c,m) = \text{t m}(t_1 \text{ x}) \{ \dots \} \text{ and } \mathcal{M}(g'.c,m) \neq \mathcal{Ulf} \implies \\
\mathcal{M}(g'.c,m) = \text{t m}(t_1 \text{ x}) \{ e' \} \\
\hline
\vdash g.c \diamond_{fb} \\
\\
\mathcal{C}(g.c) = \text{class } c <: \text{gr.c}' \{ \dots \} \text{ and } g' = \text{gr}[g] \implies \\
\mathcal{FW}(g.c,f) \neq \mathcal{Ulf} \implies \mathcal{F}(g'.c',f) = \mathcal{Ulf} \\
\mathcal{MW}(g.c,m) = \text{t m}(t_1 \text{ x}) \{ \dots \} \text{ and } \mathcal{M}(g'.c,m) = t' \text{ m}(t'_1 \text{ x}) \{ \dots \} \implies \\
t = t'[g'] \\
t_1 = t'_1[g'] \\
\hline
\vdash g.c \diamond_{inh} \\
\\
\mathcal{G}(g) = \text{group } g \ll g' \{ \dots \} \\
\mathcal{C}(g.c) = \text{class } c <: \text{gr.c}' \{ \dots \} \\
\mathcal{F}(g'.c,f) \neq \mathcal{Ulf} \implies \mathcal{F}(\text{gr}[g].c',f) = \mathcal{Ulf} \\
\mathcal{M}(g'.c,m) \neq \mathcal{Ulf} \implies \mathcal{M}(\text{gr}[g].c',m) = \mathcal{Ulf} \\
\hline
\vdash g.c \diamond_{uniq} \\
\\
\vdash P \diamond_{ag} \quad \vdash P \diamond_{ac} \quad \vdash g.c \diamond_{general} \quad \vdash g.c \diamond_{fb} \quad \vdash g.c \diamond_{inh} \quad \vdash g.c \diamond_{uniq} \\
\hline
\vdash g.c \diamond \\
\\
\vdash P \diamond_{ag} \\
\forall c: \mathcal{C}(g.c) = \text{class } c <: \text{gr.c}' \{ \dots \} \implies \vdash g.c \diamond \\
\hline
\vdash g \diamond \\
\\
\forall g: \mathcal{G}(g) \neq \mathcal{Ulf} \implies \vdash g \diamond \\
\hline
\vdash P \diamond
\end{array}$$

Fig. 11 — Well formed classes, groups and programs

## D Encoding ThisClass

Here we discuss some preliminary ideas on the encoding of `ThisClass` in `CONCORD`. We use the widely known one- and two-dimensional points example, where a method `equal`, defined in classes `Point1D` and `Point2D`, should compare either two `Point1D` objects, or two `Point2D` objects, but never a `Point1D` with a `Point2D` object. In [3], this is described through a parameter of type `ThisClass`, indicating that the argument should have the same class as the receiver. Here, we use the type `MyGrp.Point`:

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

Assuming that `p1` and `p1'` are variables of type `OneDim.Point`, and `p2` and `p2'` are variables of type `TwoDim.Point`, the following expressions type check as follows:

```
type correct:  p1.equal(p1');  p2.equal(p2');
               p1 = p1';      p2 = p2';
type incorrect: p1.equal(p2);  p2.equal(p1);
               p1 = p2;
```

Needless to say, we could also have overwritten the method `equal` in the two dimensions subgroup:

```
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 (if we supported overloading, it would have overloaded the method):

```
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`, for the function:

```

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

```

the call `equalPoints(p1,p1')` is legal, but `equalPoints(p2,p2')` is illegal. As execution of the latter does not fail, it is a shortcoming of `CONCORD`, that the expression is illegal. In fact, in [3] it is legal. In order to overcome this shortcoming, we plan, in further work, to support type and method signatures parameterisation by groups, e.g. to allow a function like:

```

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

```

## E The Cow Example

In this example, animals eat food, and cows eat grass. If we model `Cows` as subclasses of `Animals`, and `Grass`, `Meat` as subclasses of `Food`, and if `Animal` has a field of type `Food`, then the type system cannot prevent cows from eating meat. This examples has been used to motivate the introduction of virtual types. We shall encode the example in `CONCORD`, by using groups of animals, herbivores and carnivores, and thus we shall prevent mixing their eating habits across these groups.

```

group FoodG {
    class Food { int calories; }
    class Grass <: FoodG.Food {
        // stuff to do with grass
    }
    class Meat <: FoodG.Food {
        // stuff to do with meat
    }
}
group AnimalG {
    class Animal {
        MyGrp.Food eat;
    }
    class Food {
        // AnimalG.Food is unrelated to FoodG.Food
        ...
    }
}
group HerbivoreG << AnimalG {
    class Cow <: MyGrp.Animal {}
    class Food <: FoodG.Grass {
        // stuff to do with Grass
    }
}
group CarnivoreG << AnimalG {
    class Food <: FoodG.Meat{
        // stuff to do with Grass
    }
}

```

Now, assuming that `gr`, `mt`, `herbia`, `cow`, `herbif`, `carnia`, `carnif` are variables of types `FoodG.Grass`, `FoodG.Meat`, `HerbivoreG.Animal`, `HerbivoreG.Cow`, `HerbivoreG.Food`, `CarnivoreG.Animal`, `CarnivoreG.Food`, respectively, then the following expressions would type check as follows:

```

type correct:   herbia.eat = herbif;   cow.eat = herbif;
                  carnia.eat = carnif;
type incorrect: herbia.eat = gr;      carnia.eat = mt;
                  carnia.eat = gr;

```

## F The Expression Problem

The expression problem requires the independent extension of an abstract data type with more cases (more subclasses in OO languages) and more functions, and to the possibility to combine these extensions. In `CONCORD` we can express the first but not the second requirement.

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):

```

group BaseGroup {
  class Expr {
    int eval() { 0 } // should be an abstract method
  }
  class Num <: MyGrp.Expr {
    int value;
    int eval() { this.value }
  }
}

```

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

```

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: `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`. A notion of traits, or a notion of virtual superclasses could possibly achieve that.