Simple Dependent Types: Concord*

Paul Jolly[1], Sophia Drossopoulou[1], Christopher Anderson[1], and Klaus Ostermann[2]

[1] Imperial College London
[2] University of Technology Darmstadt

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 introduce motivating and exploratory examples, 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 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`. 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 [7], in the programming language Scala [14] and in [4]. In this paper we present Concord, a simple approach to the problem inspired by ideas from [4], less powerful than Scala. 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.

Concord is based on the following 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[3].
- Groups may extend other groups; classes defined in a group $\mathtt{g}'$ (the supergroup) are *further bound* in any subgroup `g`.
- A class `g.c` may extend another class $\mathtt{gr.c}'$ (`gr` is a group reference and is either a group name or `MyGrp`), in which case it is a subclass of $\mathtt{gr.c}'$.
- Subclasses and further binding induce inheritance: If `g.c` is a subclass of $\mathtt{gr.c}'$, then it inherits all members (fields and methods) from $\mathtt{gr.c}'$, replacing occurrences of `MyGrp` in the member definitions by `gr` in the process. If `g.c` further binds $\mathtt{g}'\mathtt{.c}$, then it inherits all members from $\mathtt{g}'\mathtt{.c}$, but instead leaves occurrences of `MyGrp` in member definitions unmodified. Furthermore, subclasses create subtypes (i.e. if `g.c` is a subclass of $\mathtt{gr.c}'$ then `g.c` is a subtype of $\mathtt{gr.c}'$), while further binding does not[4]. Classes may both further bind and subclass other classes.

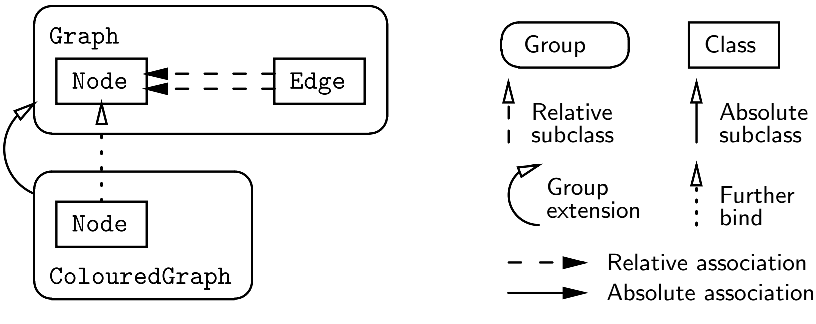
* This work is partly supported by DART, European Commission Research Directorates, IST-01-6-1A.

[3] 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.

[4] Thus, subclasses correspond to inheritance in Java; further binding is as per suggestions in e.g. [20]

```
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 } // further bind Graph.Node
  // ColouredGraph.Edge further binds Graph.Edge
}
```

Listing 1 — CONCORD encoding of the graph example



(a) The graph example (b) Diagram legend

Fig. 1 — Graphical representation of the graph example encoded in CONCORD. Further binding arrows are drawn only in the case where redefinition/extension of the further bound class occurs in the subgroup.

A CONCORD encoding of the graph example is presented in Listing 1 (a graphical representation can be seen in Fig. 1(a)). 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)`; for `n1`, `n2` of type `Graph.Node` and `cn1`, `cn2` of type `ColouredGraph.Node`, the terms `n1.connect(n2)` and `cn1.connect(cn2)`, which return objects of types `Graph.Edge` and `ColouredGraph.Edge` respectively, are type correct, whereas `n1.connect(cn1)` and `cn1.connect(n1)` are type incorrect.

CONCORD classes differ from JAVA inner classes: firstly, inner classes cannot express intra-group relations and secondly, objects of CONCORD classes do not hold references to objects of the enclosing group [21].

Furthermore, CONCORD's relative types are more powerful than simply distinguishing inheritance from subtypes. In the latter case, for example, the expression `cn1.connect(cn2)` would have type `Node.Edge` (rather than the more precise `ColouredGraph.Edge`).

Finally, CONCORD's relative types are more powerful than references to the type of self (e.g. in SATHER [15], EIFFEL [16, 17], and in [4]); the former may refer to any class within the receiver's group, whereas the latter references are restricted to the same class as the receiver.

In this paper, we cite the graph example as motivation for CONCORD. In [12] we present more examples and compare our approach to others in the literature through familiar problems, i.e. two- and three-dimensional points, the cow and food example, and the expression problem.

The development of CONCORD posed the following challenges: the exact meaning of relative types, the difference between absolute and relative superclasses, the use of relative types as the type of `this`, and also in typing expressions whose subexpressions have relative types. The remainder

prog	::=	*group**
group	::=	`group g` $\ll$ `g` { *class** }
class	::=	`class c` <: *type* { *field** *meth** }
field	::=	*type* `f`
meth	::=	*type* `m` (*type* `x`) { *exp* }
exp	::=	`null` | `new` *type* | `this` | *exp* . `f` | *exp* . `m` (*exp*) | *exp* . `f` = *exp*
type	::=	`gr . c`
`gr`	::=	`g` | `MyGrp`
`ta`	::=	`g . c`

Fig. 2 — CONCORD syntax

of this paper is mainly dedicated to clarifying how we tackled these issues. As requested by our reviewers, we added some intuition as to the reasons behind our solutions; however, a fully intuitive explanation, although very worthwhile, was impossible to achieve within the time constraints.

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

In [12] we present complete formal details, more examples and explanations. A smaller version is available in [11].

2 Syntax

Fig. 2 introduces CONCORD syntax where `g`, `c`, `f` and `m` represent group, class, field and method identifiers respectively.

A CONCORD program comprises group definitions which, in turn, comprise class definitions. Classes are similar to those in JAVA or C♯ with the difference that CONCORD types comprise a group reference (`gr`) and a class identifier (`c`). The group reference may be a group name (e.g. `g1`) or the keyword `MyGrp`. The former introduces an *absolute* type, anchored to the named class defined within the named group. The latter introduces a *relative* type, the target of which changes when inherited. Every group extends another group and every class extends another class; `GlobalGroup` and `GlobalGroup.c` are therefore at the top of the hierarchy.

The CONCORD program in Listing 2 (visualised in Fig. 3) will serve as our running example[5].

In class `g2.A`, notice that field `f1` has an absolute type, whereas field `f2` has a relative type. Similarly, method `m1` has relative argument and result types, while method `m2` has absolute argument and result types. Furthermore, the superclass of `g2.B` is relative, whereas the superclass of `g2.C` is absolute. In Section 3, we shall see how the relative types are inherited.

3 Inheritance

As we said earlier, both further binding and subclasses induce inheritance but in slightly different ways: 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). Therefore, in our example, the type of `f2` in `g3.A` is `MyGrp.A` and the type of `f2` in `g2.C` is `g2.A`. However, the type of `f2` in `g2.B` is `MyGrp.A`, because although `g2.B` is a subclass, it has a *relative* superclass.

In the rest of this section, we describe the mechanisms behind these issues in more detail.

[5] Much like JAVA, we omit supergroup and superclass references in group and class definitions when those references are `GlobalGroup` or `GlobalGroup.c` respectively.

```
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 { ... }   // relative subclass
  class C <: g2.A { ... }      // absolute subclass
}
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!
    }
  }
}
```

Listing 2 — Our running example

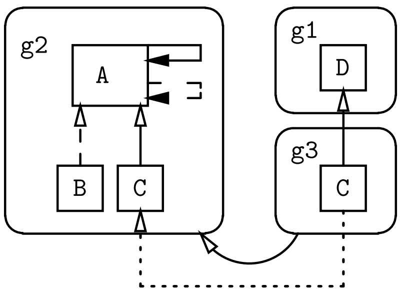


Fig. 3 — Graphical representation of our running example. Refer to legend in Fig. 1(b).

All classes from a certain group are further bound in a subgroup. For example, classes `g3.A` and `g3.B` further bind `g2.A` and `g2.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 $\mathtt{c}$, defined via $\mathtt{group}\ \ \mathtt{g} \ll \mathtt{g}'\,\{\ldots\mathtt{class}\ \ \mathtt{c} <: \mathtt{gr}\,.\,\mathtt{c}'\,\{\ldots\}\ldots\}$, further binds $\mathtt{g}'\,.\,\mathtt{c}$ (if such a class exists), and is a subclass of $\mathtt{gr}\,.\,\mathtt{c}'$. Therefore, $\mathtt{g}\,.\,\mathtt{c}$ inherits from both $\mathtt{g}'\,.\,\mathtt{c}$ and $\mathtt{gr}\,.\,\mathtt{c}'$. The exact process is described via the definitions of functions $\mathcal{F}$ and $\mathcal{M}$ in Fig. 4; the operator $\oplus$ is defined in Appendix B.

Through further binding, $\mathtt{g}\,.\,\mathtt{c}$ inherits all members from $\mathtt{g}'\,.\,\mathtt{c}$ unmodified, i.e. $\mathcal{F}(\mathtt{g}\,.\,\mathtt{c}, \mathtt{f}) = \ldots \oplus \ldots \oplus \mathcal{F}(\mathtt{g}'\,.\,\mathtt{c}, \mathtt{f})$. Inheritance though subclassing is more intricate: First the superclass is determined by replacing any intra-group reference in $\mathtt{gr}\,.\,\mathtt{c}'$ by the current group, $\mathtt{g}$. Then, any intra-group references in the member definition are replaced by the reference to the supergroup, $\mathtt{gr}$: $\mathcal{F}(\mathtt{g}\,.\,\mathtt{c}, \mathtt{f}) = \ldots \oplus \mathcal{F}(\mathtt{gr}[\mathtt{g}]\,.\,\mathtt{c}', \mathtt{f})[\mathtt{gr}] \oplus \ldots$.

In our example, `g3.A` further binds `g2.A`, therefore inheriting the fields `f1` and `f2` with *unmodified* types. On the other hand, `g2.C` is a subclass of `g2.A` and so inherits the fields `f1` and `f2` *after* substitution of intra-group references in their type by `g2`. Finally, `g2.B` is a subclass of `MyGrp.A` and therefore inherits the fields `f1` and `f2` from `g2.A` with *unmodified* types, i.e. after replacing `MyGrp` by `MyGrp`.

$$\frac{\mathcal{CW}(\texttt{g.c}) = \texttt{class c} <: \ldots \{ \ldots \texttt{t m} (\texttt{t}_1 \ \texttt{x}) \{ \texttt{e} \} \ldots \}}{\mathcal{MW}(\texttt{g.c}, \texttt{m}) = \texttt{t m} (\texttt{t}_1 \ \texttt{x}) \{ \texttt{e} \}}$$

$$\frac{}{\mathcal{MW}(\texttt{GlobalGroup.c}, \texttt{m}) = \mathcal{U}df}$$

$$\frac{\begin{array}{c}\mathcal{G}(\texttt{g}) = \texttt{group g} \ll \texttt{g}' \{ \ldots \} \\ \mathcal{C}(\texttt{g.c}) = \texttt{class c} <: \texttt{gr.c}' \{ \ldots \}\end{array}}{\mathcal{M}(\texttt{g.c}, \texttt{m}) = \mathcal{MW}(\texttt{g.c}, \texttt{m}) \oplus \mathcal{M}(\texttt{gr}[\texttt{g}].\texttt{c}', \texttt{m})[\texttt{gr}] \oplus \mathcal{M}(\texttt{g}'.\texttt{c}, \texttt{m})}$$

$$\frac{\mathcal{CW}(\texttt{g.c}) = \texttt{class c} <: \ldots \{ \ldots \texttt{t f} \ldots \}}{\mathcal{FW}(\texttt{g.c}, \texttt{f}) = \texttt{t}} \qquad \frac{}{\mathcal{FW}(\texttt{GlobalGroup.c}, \texttt{f}) = \mathcal{U}df}$$

$$\frac{\begin{array}{c}\mathcal{G}(\texttt{g}) = \texttt{group g} \ll \texttt{g}' \{ \ldots \} \\ \mathcal{C}(\texttt{g.c}) = \texttt{class c} <: \texttt{gr.c}' \{ \ldots \}\end{array}}{\mathcal{F}(\texttt{g.c}, \texttt{f}) = \mathcal{FW}(\texttt{g.c}, \texttt{f}) \oplus \mathcal{F}(\texttt{gr}[\texttt{g}].\texttt{c}', \texttt{f})[\texttt{gr}] \oplus \mathcal{F}(\texttt{g}'.\texttt{c}, \texttt{f})}$$

$$\frac{}{\mathcal{F}s(\texttt{g.c}) = \{ \texttt{f} \mid \mathcal{F}(\texttt{g.c}, \texttt{f}) \neq \mathcal{U}df \}}$$

Fig. 4 — Method and field lookup functions

The following table demonstrates field and method inheritance in our example:

subclassing — fields
$\mathcal{F}(\texttt{g2.A}, \texttt{f1}) = \mathcal{F}(\texttt{g2.B}, \texttt{f1}) = \mathcal{F}(\texttt{g2.C}, \texttt{f1}) = \texttt{g2.A}$
$\mathcal{F}(\texttt{g2.A}, \texttt{f2}) = \mathcal{F}(\texttt{g2.B}, \texttt{f2}) = \texttt{MyGrp.A}$
$\mathcal{F}(\texttt{g2.C}, \texttt{f2}) = \texttt{g2.A}$
$\mathcal{F}(\texttt{g3.C}, \texttt{f3}) = \texttt{g1.D}$

further binding — fields
$\mathcal{F}(\texttt{g3.A}, \texttt{f1}) = \mathcal{F}(\texttt{g3.B}, \texttt{f1}) = \mathcal{F}(\texttt{g3.C}, \texttt{f1}) = \texttt{g2.A}$
$\mathcal{F}(\texttt{g3.A}, \texttt{f2}) = \mathcal{F}(\texttt{g3.B}, \texttt{f2}) = \texttt{MyGrp.A}$
$\mathcal{F}(\texttt{g3.C}, \texttt{f2}) = \texttt{g2.A}$

subclassing — methods
$\mathcal{M}(\texttt{g2.A}, \texttt{m1}) = \mathcal{M}(\texttt{g2.B}, \texttt{m1}) = \texttt{MyGrp.A m1 (MyGrp.A x)} \{ \texttt{new MyGrp.A} \}$
$\mathcal{M}(\texttt{g2.C}, \texttt{m1}) = \texttt{g2.A m1 (g2.A x)} \{ \texttt{new g2.A} \}$

4 Execution

We define execution in terms of large step semantics whereby an expression and store are mapped onto a value and store. The store, σ, represents the stack and heap. It maps `this` and the method parameter `x` onto addresses and addresses onto objects. Objects $[\![\texttt{g.c} \ \| \ \texttt{f}_l : \texttt{v}_l]\!]$ contain their runtime type (or absolute type, `ta`) and the values of their fields. This can be seen in Fig. 5.

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

$$\mathcal{M}yGrp(\sigma) = \texttt{g} \quad \text{where} \quad \sigma(\sigma(\texttt{this})) = [\![\texttt{g.c} \ \| \ \ldots]\!]$$

In Fig. 6 we define the operational semantics. All rules are straightforward and similar to those in [6], with the exception of the rule for object creation. The runtime type of the object being

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

Fig. 5 — Stores of and operations on objects o; store σ and identifier or address z.

created may depend on the runtime type of the current receiver. For example, with a receiver of runtime type `g3.A`, execution of `new MyGrp.C` will create an object of dynamic type `g3.C`.

5 Types

We roughly follow the Java approach, whereby subclasses introduce subtypes. However, further binding does not introduce subtypes. Therefore, in our example, `g2.C` is a subtype of `g2.A`, but `g3.A` is not a subtype of `g2.A`. With relative types things become more complex, as their meaning (and thus also the meaning of the subtype relationship) is context dependent. 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`).

Typing of expressions takes place in the context of a method body, in a given class within a given group, say `g.c`. The question arises as to what the type of `this` should be. Since the method may be inherited by any class which further binds the current class, it makes sense to consider `this` to have a type `MyGrp.c`, and the context to be `g`. Therefore, in our example, within class `g3.A`, the receiver, `this` has type `MyGrp.A`.

The next issue concerns the type of member access. A member may have an absolute or relative type; equally, the receiver may have an absolute or relative type. If the member has an absolute type `ta`, then member access has type `ta` (irrespective of the receiver type): e.g. in class `g3.A`, the receiver, `this`, has type `MyGrp.A` and `this.f1` has type `g2.A`. If the member has a relative type, then intra-group dependencies are replaced by the group reference `gr` in the receiver's type: e.g. `(new g3.A).f2` has type `g3.A`, whereas `this.f2` (in the context of `g3.A`) has type `MyGrp.A`.

In the rest of this section, we describe the mechanisms behind the above issues in more detail. Expressions are typed in the context of an environment, Γ, which assigns types to the receiver, `this`, and the method parameter, `x`. $\Gamma(id)$ returns the type of id in Γ. For $\Gamma = \texttt{t x}, \texttt{g.c this}$, we define $\Gamma(id)$ to be `t` if $id = \texttt{x}$, `g` if $id = \texttt{MyGrp}$, `MyGrp.c` if $id = \texttt{this}$ and $\mathcal{U}\!\mathit{df}$ otherwise. $\Gamma(\texttt{this})$ always has the form `MyGrp.c`.

The functions $\mathcal{M}yGrp(\texttt{t})$ and $\mathcal{M}yGrp(\Gamma)$ extract the group of the type `t` and the receiver of Γ respectively. The operations $\texttt{e}[\texttt{g}]$ and $\texttt{t}[\texttt{t}']$ replace any occurrence of `MyGrp` in an expression and type respectively.

$$
\begin{array}{ll}
\mathcal{M}yGrp(\Gamma) & = \Gamma(\texttt{MyGrp}) \\
\mathcal{M}yGrp(\texttt{t}) & = \texttt{gr} \quad \text{where } \texttt{t} = \texttt{gr.c} \\
\texttt{e}[\texttt{g}] & = \texttt{e}\,[\,\texttt{g}\,/\,\texttt{MyGrp}\,] \\
\texttt{t}[\Gamma] & = \texttt{t}\,[\,\mathcal{M}yGrp(\Gamma)\,/\,\texttt{MyGrp}\,] \\
\texttt{t}[\sigma] & = \texttt{t}\,[\,\mathcal{M}yGrp(\sigma)\,/\,\texttt{MyGrp}\,] \\
\texttt{t}[\texttt{t}'] & = \texttt{t}\,[\,\mathcal{M}yGrp(\texttt{t}')\,/\,\texttt{MyGrp}\,] \\
(\texttt{t m}\,(\texttt{t}'\ \texttt{x})\,\{\,\texttt{e}\,\})[\texttt{g}] & = \texttt{t}[\texttt{g}]\ \texttt{m}\,(\texttt{t}'[\texttt{g}]\ \texttt{x})\,\{\,\texttt{e}[\texttt{g}]\,\}
\end{array}
$$

$$\frac{}{\texttt{v},\ \sigma\ \leadsto\ \texttt{v},\ \sigma}\ (\mathsf{val}) \qquad \frac{}{\begin{array}{l}\texttt{x},\ \sigma\ \leadsto\ \sigma(\texttt{x}),\ \sigma\\ \texttt{this},\ \sigma\ \leadsto\ \sigma(\texttt{this}),\ \sigma\end{array}}\ (\mathsf{var})$$

$$\frac{\begin{array}{l}\texttt{e},\ \sigma\ \leadsto\ \iota,\ \sigma''\\ \texttt{e}',\ \sigma''\ \leadsto\ \texttt{v},\ \sigma'''\\ \sigma'''(\iota)(\texttt{f}) \neq \mathcal{U}df\\ \sigma' = \sigma'''[\,\iota \mapsto \sigma'''(\iota)[\texttt{f} \mapsto \texttt{v}]\,]\end{array}}{\texttt{e.f} = \texttt{e}',\ \sigma\ \leadsto\ \texttt{v},\ \sigma'}\ (\mathsf{fldAss}) \qquad \frac{\begin{array}{l}\texttt{e},\ \sigma\ \leadsto\ \iota,\ \sigma'\\ \sigma'(\iota)(\texttt{f}) \neq \mathcal{U}df\end{array}}{\texttt{e.f},\ \sigma\ \leadsto\ \sigma(\iota)(\texttt{f}),\ \sigma'}\ (\mathsf{fld})$$

$$\frac{\begin{array}{l}\texttt{g} = \texttt{gr}[\mathcal{M}yGrp(\sigma)]\\ \mathcal{F}s(\texttt{g.c}) = \{\texttt{f}_1, \ldots, \texttt{f}_r\}\\ \iota \text{ is new}\end{array}}{\texttt{new gr.c},\ \sigma\ \leadsto\ \iota,\ \sigma[\,\iota \mapsto [\![\,\texttt{g.c} \parallel \texttt{f}_1 : \texttt{null}, \ldots, \texttt{f}_r : \texttt{null}\,]\!]\,]}\ (\mathsf{new})$$

$$\frac{\begin{array}{l}\texttt{e}_0,\ \sigma\ \leadsto\ \iota,\ \sigma_0\\ \texttt{e}_1,\ \sigma_0\ \leadsto\ \texttt{v}_1,\ \sigma_1\\ \sigma_1(\iota) = [\![\,\texttt{ta} \parallel \ldots]\!]\\ \mathcal{M}(\texttt{ta},\texttt{m}) = \texttt{t m}\,(\texttt{t}_1\ \texttt{x})\,\{\,\texttt{e}\,\}\\ \sigma'' = \sigma_1[\,\texttt{this} \mapsto \iota\,][\,\texttt{x} \mapsto \texttt{v}_1\,]\\ \texttt{e},\ \sigma''\ \leadsto\ \texttt{v},\ \sigma'\end{array}}{\texttt{e}_0\,.\,\texttt{m}\,(\,\texttt{e}_1\,),\ \sigma\ \leadsto\ \texttt{v},\ \sigma'[\,\texttt{this} \mapsto \sigma(\texttt{this}), \texttt{x} \mapsto \sigma(\texttt{x})\,]}\ (\mathsf{methCall})$$

Fig. 6 — Operational semantics. We omit rules for throwing and propagation of exceptions; they are standard

Fig. 7 defines the subtype relationship $\texttt{g} \vdash \texttt{t} <: \texttt{t}'$, whereby a type $\texttt{t}$ is a subtype of another type $\texttt{t}'$ in the context of a specific group $\texttt{g}$. The group context is necessary when $\texttt{t}$ or $\texttt{t}'$ reference MyGrp. For example:

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

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 $\texttt{g}$, $\texttt{g}'$ with $\vdash \texttt{g}' \ll \texttt{g}$: $\texttt{g} \vdash \texttt{t}' <: \texttt{t} \Longrightarrow \texttt{g}' \vdash \texttt{t}' <: \texttt{t}$ and $\vdash \texttt{g.c} <: \texttt{ta} \Longrightarrow \vdash \texttt{g}'\texttt{.c} <: \texttt{ta}$.

We can also prove that any absolute type, $\texttt{g}'\texttt{.c}'$, inherits every member from its supertype $\texttt{g.c}$; the members are inherited unmodified if the two groups $\texttt{g}$ and $\texttt{g}'$ are equal, otherwise occurrences of MyGrp are replaced by g.

Fig. 7 also defines the type rules $\Gamma \vdash \texttt{e} : \texttt{t}$, whereby an expression $\texttt{e}$ has type $\texttt{t}$ in the context of an environment Γ. The rules (VarThis), (NewNull) and (Subsump) are standard. The rule (Fld) is more interesting for two reasons. Firstly, it looks up the field $\texttt{f}$ in $\texttt{t}[\Gamma]$ (through $\texttt{t}' = \mathcal{F}(\texttt{t}[\Gamma], \texttt{f})$), the absolute type found by replacing occurrences of MyGrp in t by the group containing the current method definition. Secondly, it replaces occurrences of MyGrp in $\texttt{t}'$ by the group to which the receiver, e, belongs. For example:

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

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

$$\frac{\mathcal{C}(\mathtt{g.c}) = \mathtt{class\ \ c} <: \mathtt{t}\,\{\dots\}}{\mathtt{g} \vdash \mathtt{MyGrp.c} \ <: \ \mathtt{t}} \qquad \frac{\mathtt{g} \vdash \mathtt{t} \ <: \ \mathtt{t}'}{\mathtt{g}' \vdash \mathtt{t[g]} \ <: \ \mathtt{t}'\mathtt{[g]}}$$

$$\frac{\begin{array}{l}\mathtt{g} \vdash \mathtt{t} \ <: \ \mathtt{t}' \\ \vdash \mathtt{g}' \ll \mathtt{g}\end{array}}{\mathtt{g}' \vdash \mathtt{t} \ <: \ \mathtt{t}'} \qquad \frac{\begin{array}{l}\mathtt{g} \vdash \mathtt{t} \ <: \ \mathtt{t}'' \\ \mathtt{g} \vdash \mathtt{t}'' \ <: \ \mathtt{t}'\end{array}}{\mathtt{g} \vdash \mathtt{t} \ <: \ \mathtt{t}'} \qquad \frac{\mathtt{g} \vdash \mathtt{ta} \ <: \ \mathtt{ta}'}{\vdash \mathtt{ta} \ <: \ \mathtt{ta}'}$$

$$\frac{\begin{array}{l}\Gamma \vdash \mathtt{e} : \mathtt{t} \\ \mathcal{M}yGrp(\Gamma) \vdash \mathtt{t} \ <: \ \mathtt{t}'\end{array}}{\Gamma \vdash \mathtt{e} : \mathtt{t}'}\ (\mathsf{Subsump}) \qquad \frac{\begin{array}{l}\vdash \Gamma \diamond \\ \Gamma \vdash \mathtt{t} \diamond_t\end{array}}{\begin{array}{l}\Gamma \vdash \mathtt{null} : \mathtt{t} \\ \Gamma \vdash \mathtt{new\ t} : \mathtt{t}\end{array}}\ (\mathsf{NewNull})$$

$$\frac{\vdash \Gamma \diamond}{\begin{array}{l}\Gamma \vdash \mathtt{x} : \Gamma(\mathtt{x}) \\ \Gamma \vdash \mathtt{this} : \Gamma(\mathtt{this})\end{array}}\ (\mathsf{VarThis}) \qquad \frac{\begin{array}{l}\Gamma \vdash \mathtt{e}_0 : \mathtt{t}_0 \\ \mathcal{M}(\mathtt{t}_0[\Gamma], \mathtt{m}) = \mathtt{t\ m}\,(\mathtt{t}_1\ \mathtt{x})\,\{\dots\} \\ \Gamma \vdash \mathtt{e}_1 : \mathtt{t}_1[\mathtt{t}_0]\end{array}}{\Gamma \vdash \mathtt{e}_0 . \mathtt{m}\,(\,\mathtt{e}_1\,) : \mathtt{t}[\mathtt{t}_0]}\ (\mathsf{Meth})$$

$$\frac{\begin{array}{l}\Gamma \vdash \mathtt{e} : \mathtt{t} \\ \mathcal{F}(\mathtt{t}[\Gamma], \mathtt{f}) = \mathtt{t}'\end{array}}{\Gamma \vdash \mathtt{e.f} : \mathtt{t}'[\mathtt{t}]}\ (\mathsf{Fld}) \qquad \frac{\begin{array}{l}\Gamma \vdash \mathtt{e} : \mathtt{t} \\ \mathcal{F}(\mathtt{t}[\Gamma], \mathtt{f}) = \mathtt{t}' \\ \Gamma \vdash \mathtt{e}' : \mathtt{t}'[\mathtt{t}]\end{array}}{\Gamma \vdash \mathtt{e.f} = \mathtt{e}' : \mathtt{t}'[\mathtt{t}]}\ (\mathsf{FldAss})$$

Fig. 7 — Subtypes and the type system

prove that, given $\Gamma \vdash \mathtt{e} : \mathtt{t}$, if we replace the type of the receiver in the environment Γ with a subtype (to give Γ'), and substitute occurrences of `MyGrp` in an expression `e` by $\mathcal{M}yGrp(\Gamma)$ (to give $\mathtt{e}[\Gamma]$), then, in the context of the environment Γ', the expression $\mathtt{e}[\Gamma]$ has type $\mathtt{t}[\Gamma]$, the type obtained by replacing all occurrences of `MyGrp` with $\mathcal{M}yGrp(\Gamma)$ in `t`. Therefore, inheritance of methods by subclasses preserves method body types, modulo the necessary substitutions of `MyGrp`.

Thus, we can prove that in a well formed program (well formed programs, $\vdash \mathtt{P} \diamond$, are described in Appendix A), 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`:

Theorem 1 In a well formed program:

$$\mathcal{M}(\mathtt{ta}, \mathtt{m}) = \mathtt{t}_1\ \mathtt{m}\,(\mathtt{t}_2\ \mathtt{x})\,\{\,\mathtt{e}\,\} \Longrightarrow \mathtt{ta\ this}, \mathtt{t}_2\ \mathtt{x} \vdash \mathtt{e} : \mathtt{t}_1$$

6 Soundness and Decidability

In Fig. 8 we define agreement between addresses and types, $\sigma \vdash \iota : \mathtt{ta}$, whereby an address agrees with the runtime type (or any supertype) of the corresponding object, and `null` agrees with all types. An address ι 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 Γ.

We can now prove soundness of our type system:

$$\frac{\sigma(\iota) = [\![\, \mathtt{ta} \;\|\; \ldots \,]\!]}{\sigma \;\vdash\; \iota \;:\; \mathtt{ta}} \qquad \frac{\begin{array}{l}\sigma \;\vdash\; \iota \;:\; \mathtt{ta} \\ \vdash\; \mathtt{ta} \;<:\; \mathtt{ta}'\end{array}}{\sigma \;\vdash\; \iota \;:\; \mathtt{ta}'} \qquad \frac{}{\sigma \;\vdash\; \mathtt{null} \;:\; \mathtt{ta}}$$

$$\frac{\begin{array}{l}\sigma(\iota) = [\![\, \mathtt{g.c} \;\|\; \ldots \,]\!] \\ \mathcal{F}(\mathtt{g.c}, \mathtt{f}) = \mathtt{t} \;\Longrightarrow\; \sigma \;\vdash\; \sigma(\iota)(\mathtt{f}) \;:\; \mathtt{t}[\mathtt{g}]\end{array}}{\sigma \;\vdash\; \iota}$$

$$\frac{\begin{array}{l}\sigma(\iota) \neq \mathcal{U}\!\mathit{df} \;\Longrightarrow\; \sigma \;\vdash\; \iota \\ \sigma \;\vdash\; \sigma(\mathtt{x}) \;:\; \Gamma(\mathtt{x})[\Gamma] \\ \sigma \;\vdash\; \sigma(\mathtt{this}) \;:\; \Gamma(\mathtt{this})[\Gamma]\end{array}}{\Gamma \;\vdash\; \sigma \;\diamond}$$

Fig. 8 — Agreement between programs, stores and environments

Theorem 2 (Soundness)

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

We have a hand-written proof of theorem 2 by induction on the derivation of the type of `e`. The requirement that $\mathtt{e}[\Gamma] = \mathtt{e}[\sigma]$ guarantees that either there are no occurrences of `MyGrp` in `e`, or that $\sigma(\mathtt{MyGrp}) = \Gamma(\mathtt{MyGrp})$. 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 $\mathcal{F}$ 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 Work, Further Work and Conclusions

As mentioned earlier, there are many approaches to the expression of relationships between collections of classes. `TypeGroup`s 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 `TypeGroup`s, 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.

Families of classes are suggested in [7, 8] primarily through extensions to the language `gbeta`; a formalisation is planned.

SCALA [14] 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. [14] 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 [19]. 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.

The problem of relationships across collections of classes can also be addressed through virtual types [22] and its connection with generics has been explored in [23].

The connection between virtual types and dependent types is explored in [10]; soundness has not yet been demonstrated. The expression problem, posed originally by Reynolds and later suggested by Wadler in the JAVA-genericity mailing list [25], is an example of such dependence between classes. Solutions using virtual types and generics are explored in [24], and using dependent types in SCALA [26].

EIFFEL [16, 17] and SATHER [15] support references to the current class but do not combine these references with nested classes, and thus do not solve the graph example.

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

Acknowledgements — We are grateful to our anonymous referees for their careful reading, feedback and justified requests for more intuitive explanations, which kept us on our toes.

References

[1] C. Anderson and K. Ostermann. $\mathcal{VC}$ - foundations for virtual classes with dependent types. URL `http://www.binarylord.com/work/vc.pdf`. 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 Odersky [18].

[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] L. Cardelli, editor. *ECOOP 2003 – Object-Oriented Programming*. LNCS 2743. Springer-Verlag, Darmstadt, Germany, 2003. ISBN 3-540-40531-3.

[6] S. Drossopoulou, F. Damiani, M. Dezani-Ciancaglini, and P. Giannini. FICKLE: Dynamic Object Re-classification. In Knudsen [13], pages 130–149. ISBN 3-540-42206-4.

[7] E. Ernst. Family Polymorphism. In Knudsen [13], pages 303–326. ISBN 3-540-42206-4.

[8] E. Ernst. Higher-Order Hierarchies. In Cardelli [5], pages 303–329. ISBN 3-540-40531-3.

[9] R. Guerraoui, editor. *ECOOP 1999 – Object-Oriented Programming*. LNCS 1628. Springer-Verlag, Lisbon, Portugal, 1999. ISBN 3-540-66156-5.

[10] A. Igarashi and B. C. Pierce. Foundations for Virtual Types. In Guerraoui [9]. ISBN 3-540-66156-5.

[11] P. Jolly, S. Drossopoulou, C. Anderson, and K. Ostermann. Simple Dependent Types: CONCORD (FTfJP accepted version). April 2004. URL `http://myitcv.org.uk/papers/concord04.html`.

[12] P. A. Jolly. Simple Dependent Types: CONCORD (Masters Thesis). Master's thesis, Department of Computing, Imperial College of Science, Technology and Medicine, University of London, June 2004. URL `http://myitcv.org.uk/`. *To appear June 16, 2004*.

[13] J. L. Knudsen, editor. *ECOOP 2001 – Object-Oriented Programming*. LNCS 2072. Springer-Verlag, Heidelberg, European Conference in Germany, 2001. ISBN 3-540-42206-4.

[14] LAMP/EPFL. SCALA website. URL `http://scala.epfl.ch/`.

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

[16] B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall, Inc., 1988. ISBN 0136290493.

[17] B. Meyer. *Object-oriented software construction (2nd ed.).* Prentice-Hall, Inc., 1997. ISBN 0-13-629155-4.
[18] M. Odersky, editor. *ECOOP 2004 – Object-Oriented Programming.* Springer-Verlag, Oslo, Norway, 2004.
[19] M. Odersky, V. Cremet, C. Röckl, and M. Zenger. A Nominal Theory of Objects with Dependent Types. In Cardelli [5]. ISBN 3-540-40531-3.
[20] K. Ostermann. Dynamically Composable Collaborations with Delegation Layers. In J. Hernández and A. Moreira, editors, *ECOOP 2002 – Object-Oriented Programming*, LNCS 2548, Málaga, Spain, 2002. Springer-Verlag. ISBN 3-540-43759-2.
[21] M. Smith and S. Drossopoulou. Inner Classes visit Aliasing. Number 408, pages 4–11, 2003.
[22] K. K. Thorup. Genericity in Java with Virtual Types. In M. Aksit and S. Matsuoka, editors, *ECOOP 1997 – Object-Oriented Programming*, LNCS 1241, pages 444–, Jyväskylä, Finland, 1997. Springer. ISBN 3-540-63089-9.
[23] K. K. Thorup and M. Torgersen. Unifying Genericity — Combining the Benefits of Virtual Types and Parameterized Classes. In Guerraoui [9]. ISBN 3-540-66156-5.
[24] M. Torgersen. The Expression Problem Revisited — Four new solutions using generics. In Odersky [18].
[25] Various. Java-Genericity email list. List emails collated to form a webpage of links to each correspondence, 1997–2000. URL `http://www.cis.ohio-state.edu/~gb/cis888.07g/java-genericity/`.
[26] M. Zenger and M. Odersky. Independently Extensible Solutions to the Expression Problem. URL `http://scala.epfl.ch/docu/related.html`.

A Description of auxiliary definitions

In the interests of providing a short paper, we present verbal descriptions of auxiliary definitions required by our formal system.

$\vdash\ \mathtt{g} \ll \mathtt{g}'$ **— subgroups** Given $\mathtt{group}\ \ \mathtt{g} \ll \mathtt{g}'\,\{\,\dots\,\}$, we define $\vdash\ \mathtt{g} \ll \mathtt{g}'$ and $\vdash\ \mathtt{g} \ll \mathtt{g}$. The transitive closure of this relationship completes our definition of subgroups.

$\vdash\ \mathtt{P}\diamond_u$ **— unique definitions** The judgement requiring unique definitions demands that: (a) group names are unique, (b) class names within a group are unique (hence there can be no naming clash) and (c) that field and method names are unique within classes (even where subclassing and further binding are involved).

$\vdash\ \mathtt{P}\diamond_{a\mathtt{g}}$ **and** $\vdash\ \mathtt{P}\diamond_{a\mathtt{t}}$ **— acyclic groups and types** Judgements that require there to be no cycles in the extension of groups or types.

$\mathcal{G}(\mathtt{g})$ **and** $\mathcal{C}(\mathtt{g}\,.\,\mathtt{c})$ **— group and class lookup functions** The group lookup function $\mathcal{G}(\mathtt{g})$ returns the definition $\mathtt{group}\ \ \mathtt{g}{\ll}\mathtt{g}'\{\dots\}$ if such a definition exists within a Concord program. The class lookup function $\mathcal{C}(\mathtt{g}\,.\,\mathtt{c})$ returns the definition $\mathtt{class}\ \ \mathtt{c}{<:}\mathtt{t}\{\dots\}$ if such a definition exists within $\mathtt{g}$ or else if such a definition exists in a supergroup $\mathtt{g}'$ of $\mathtt{g}$.

$\vdash\ \Gamma\diamond$, $\vdash\ \mathtt{g}\,.\,\mathtt{c}\diamond$, $\vdash\ \mathtt{g}\diamond$ **and** $\vdash\ \mathtt{P}\diamond$ **— well formedness** Well formed programs comprise well formed groups which themselves comprise well formed classes. Well formed classes comprise method and field definitions of well formed types. Fields of a class may not be redefined in subclasses or further bound classes; method bodies may be redefined under an identical method signature. Classes that inherit members do so uniquely, i.e. given $\mathtt{group}\ \ \mathtt{g} \ll \mathtt{g}'\,\{\,\dots\,\}$ and $\mathtt{class}\ \ \mathtt{c} <: \mathtt{gr}\,.\,\mathtt{c}'\,\{\,\dots\,\}$, then $\mathcal{F}(\mathtt{g}'\,.\,\mathtt{c},\mathtt{f}) \neq \mathcal{U}\!\mathit{df} \implies \mathcal{F}(\mathtt{gr}[\mathtt{g}]\,.\,\mathtt{c}',\mathtt{f}) = \mathcal{U}\!\mathit{df}$ (similarly for methods).

B 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:

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