

# Using nested classes as associated types.

Authors omitted for double-bind review.

Unspecified Institution.

## Abstract

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Praesent convallis orci arcu, eu mollis dolor. Aliquam eleifend suscipit lacinia. Maecenas quam mi, porta ut lacinia sed, convallis ac dui. Lorem ipsum dolor sit amet, consectetur adipiscing elit. Suspendisse potenti.

**2012 ACM Subject Classification** Dummy classification

**Keywords and phrases** Dummy keyword

**Digital Object Identifier** 10.4230/LIPIcs.CVIT.2016.23

## 1 Introduction

Associated types are a powerful form of generics, now integrated in both Scala and Rust. They are a new kind of member, like methods fields and nested classes. Associated types behave as 'virtual' types: they can be overridden, can be abstract and can have a default. However, the user has to specify those types and their concrete instantiations manually; that is, the user have to provide a complete mapping from all virtual type to concrete instantiation. When the number of associated types is small this poses no issue, but it hinders designs where the number of associated types is large. In this paper we examine the possibility of completing a partial mapping in a desirable way, so that the resulting mapping is sound and also robust with respect to code evolution.

The core of our design is to reuse the concept of nested classes instead of relying of a new kind of member for associated types. An operation, call Redirect, will redirect some nested classes in some external types. To simplify our formalization and to keep the focus on the core of our approach, we present our system on top of a simple Java like languages, with only final classes and interfaces, when code reuse is obtained by trait composition instead of conventional inheritance. We rely on a simple nominal type system, where subtyping is induced only by implementing interfaces; in our approach we can express generics without having a polymorphic type system. To simplify the treatment of state, we consider fields to be always instance private, and getters and setters to be automatically generated, together with a `static` method `of(..)` that would work as a standard constructor, taking the value of the fields and initializing the instance. In this way we can focus our presentation to just (static) methods, nested classes and implements relationships. Expanding our presentation to explicitly include visible fields, constructors and sub-classing would make it more complicated without adding any conceptual underpinning. In our proposed setting we could write:

```
String=...
SBox={String inner;
  method String inner(){..} //implicit
  static method SBox of(String inner){..} //implicit
myTtrait={
  Box={Elem inner} //implicit Box(Elem inner) and Elem inner()
  Elem={Elem concat(Elem that)}
  static method Box merge(Box b, Elem e){return Box.of(b.inner().concat(e));}
}
Result=myTrait<Box=SBox> //equivalent to trait<Box=SBox, Elem=String>
...Result.merge(SBox.of("hello "), "world");//hello world
```



© Authors omitted for double-bind review.;  
licensed under Creative Commons License CC-BY

42nd Conference on Very Important Topics (CVIT 2016).

Editors: John Q. Open and Joan R. Access; Article No. 23; pp. 23:1–23:??

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

## 23:2 Using nested classes as associated types.

48 Here class **SBox** is just a container of **Strings**, and **myTrait** is code encoding **Boxes** of any kind  
49 of **Elem** with a **concat** method. By instantiating **myTrait<Box=SBox>**, we can infer **Elem=String**,  
50 and obtain the following flattened code, where **Box** and **Elem** has been removed, and their  
51 occurrences are replaced with **SBox** and **String**.

```
52 Result={static method SBox merge(SBox b,String e){  
53     return SBox.of(b.inner().concat(e));}  
54 }  
55
```

56 Note how **Result** is a new class that could have been written directly by the programmer,  
57 there is no trace that it has been generated by **myTrait**. We will represent trait names with  
58 lower-case names and class/interface names with upper-case names. Traits are just units of  
59 code reuse, and do not induce nominal types.

60 We could have just written **Result=myTrait<Elem=String>**, obtaining

```
61 Result={  
62     Box={String inner}  
63     static method Box merge(Box b,String e){  
64         return Box.of(b.inner().concat(e));}  
65 }
```

67 Note how in this case, class **Result.Box** would exist. Thanks to our decision of using nested  
68 classes as associated types, the decision of what classes need to be redirected is not made  
69 when the trait is written, but depends on the specific redirect operation. Moreover, our  
70 redirect is not just a way to show the type system that our code is correct, but it can change  
71 the behaviour of code calling static methods from the redirected classes.

72 This example shows many of the characteristics of our approach:

- 73 ■ (A) We can redirect mutually recursive nested classes by redirecting them all at the  
74 same time, and if a partial mapping is provided, the system is able to infer the complete  
75 mapping.
- 76 ■ (B) **Box** and **Elem** are just normal nested classes inside of **myTrait**; indeed any nested  
77 class can be redirected away. In case any of their (static) methods was implemented, the  
78 implementation is just discarded. In most other approaches, abstract/associated/generic  
79 types are special and have some restriction; for example, in Java/Scala static methods  
80 and constructors can not be invoked on generic/associated types. With redirect, they are  
81 just normal nested classes, so there are no special restrictions on how they can be used.  
82 In our example, note how **merge** calls **Box.of(..)**.
- 83 ■ (C) While our example language is nominally typed, nested classes are redirected over  
84 types satisfying the same structural shape. We will show how this offers some advantages  
85 of both nominal and structural typing.

86 A variation of redirect, able to only redirect a single nested class, was already presented  
87 in literature. While points (B) and (C) already apply to such redirect, we will show how  
88 supporting (A) greatly improves their value.

89 The formal core of our work is in defining

- 90 ■ **ValidRedirect**, a computable predicate telling if a mapping respects the structural shapes  
91 and nominal subtype relations.
- 92 ■ A formal definition of what properties a procedure expanding a partial mapping into a  
93 complete one should respect.
- 94 ■ **ChoseRedirect**, an efficient algorithm respecting those properties.

95 We first formally define our core language, then we define our redirect operator and its  
96 formal properties. Finally we motivate our model showing how many interesting examples of  
97 generics and associated types can be encoded with redirect. Finally, as an extreme application,  
98 we show how a whole library can be adapted to be injected in a different environment.

## 2 Language grammar and well formedness

$e ::= x \mid e.m(es) \mid T.m(es) \mid e.x \mid \text{new } T(es)$	expression	$T ::= \text{This}n.Cs$	types
$L ::= \{\text{interface } Tz; Mz \mid \{Tz; Mz; K\}$	code literal	$Tx ::= T \ x$	parameter
$M ::= \text{static? } T \ m(Txs) \ e? \mid \text{private? } C=E$	member	$D ::= id=E$	declaration
$K ::= \langle Txs \rangle?$	state	$id ::= C \mid t$	class/trait id
$E ::= L \mid t \mid E_1 <+ E_2 \mid E < Cs = T \rangle$	Code Expr.	$v ::= \text{new } T(vs)$	value

We apply our ideas on a simplified object oriented language with nominal typing and (nested) interfaces and final classes. Instead of inheritance, code reuse is obtained by trait composition, thus the source code would be a sequence of top level declarations  $D$  followed by a main expression; a lower-case identifier  $t$  is a trait name, while an upper case identifier  $C$  is a class name. To simplify our terminology, instead of distinguishing between nested classes and nested interfaces, we will call *nested class* any member of a code literal named by a class identifier  $C$ . Thus, the term *class* may denote either an *interface class* (interface for short) or a *final class*.

In the context of nested classes, types are paths. Syntactically, we represent them as relative paths of form  $\text{This}n.Cs$ , where the number  $n$  identify the root of our path:  $\text{This}0$  is the current class,  $\text{This}1$  is the enclosing class,  $\text{This}2$  is the enclosing enclosing class and so on.  $\text{This}n.Cs$  refers to the class obtained by navigating throughout  $Cs$  starting from  $\text{This}n$ . Thus,  $\text{This}0$  is just the type of the directly enclosing class. By using a larger then needed  $n$ , there could be multiple different types referring to the same class. Here we expect all types to be in the normalized form where the smallest possible  $n$  is used.

Code literals  $L$  serve the role of class/interface bodies; they contain the set of implemented interfaces  $Tz$ , the set of members  $Mz$  and their (optional) state. In the concrete syntax we will use **implements** in front of a non empty list of implemented interfaces and we will omit parenthesis around a non empty set of fields. A class member  $M$  can be a (private) nested class or a (static) method. Abstract methods are just methods without a body. Well formed interface methods can only be abstract and non-static. To facilitate code reuse, classes can have (static) abstract methods, code composition is expected to provide an implementation for those or, as we will see, redirect away the whole class. We could easily support private methods too, but to simplify our formalism we consider private only for nested classes. In a well formed code literal, in all types of form  $\text{This}n.Cs.C.Cs'$ , if  $C$  denotes a private nested class, then  $Cs$  is empty.

Expressions are used as body of (static) methods and for the main expression. They are variables  $x$  (including **this**) and conventional (static) method calls. Field access and **new** expressions are included but with restricted usage: well formed field accesses are of form **this**. $x$  in method bodies and  $v.x$  in the main expression, while well formed **new** expressions have to be of form **new This**0( $xs$ ) in method bodies and of form  $v$  in the main expression. Those restrictions greatly simplify reasoning about code reuse, since they require different classes to only communicate by calling (static) methods. Supporting unrestricted fields and constructors would make the formalism much more involved without adding much of a conceptual difficulty. Values are of form **new**  $T(vs)$ .

For brevity, in the concrete syntax we assume a syntactic sugar declaring a static of method (that serve as a factory) and all fields getters; thus the order of the fields would induce the order of the factory arguments. In the core calculus we just assume such methods to be explicitly declared.

Finally, we examine the shape of a nested class: **private?**  $C=E$ . The right hand side is not just a code literal but a code composition expression  $E$ . In trait composition, the

## 23:4 Using nested classes as associated types.

code expression will be reduced/flattened to a code literal  $L$  during compilation. Code expressions denote an algebra of code composition, starting from code literal  $L$  and trait names  $t$ , referring to a literal declared before by  $t=E$ . We consider two operators: conventional preferential sum  $E_1 \leftarrow E_2$  and our novel redirect  $E \leftarrow C_s = T$ .

The compilation process consists in flattening all the  $E$  into  $L$ , starting from the innermost leftmost  $E$ . This means that sum and redirect work on  $LV$ s: a kind of  $L$ , where all the nested classes are of form  $C=LV$ . The execution happens after compilation and consist in the conventional execution of the main expression  $e$  in the context of the fully reduced declarations, where all trait composition has been flatted away. Thus, execution is very simple and standard and behaves like a variation of FJ with interfaces instead of inheritance, and where nested classes are just a way to hierarchically organize code names. On the other side, code composition in this setting is very interesting and powerful, where nested classes are much more than name organization: they support in a simple and intuitive way expressive code reuse patterns. To flatten an  $E$  we need to understand the behaviour of the two operators, and how to load the code of a trait: since it was written in another place, the syntactic representation of the types need to be updated. For each of those points we will first provide some informal explanation and then we will proceed formalizing the precise behaviour.

### 2.1 Redirect

Redirect takes a library literal and produce a modified version of it where some nested classes has been removed and all the types referencing such nested classes are now referring to an external type. It is easy to use this feature to encode a generic list:

```
list={
  Elem={}
  static This0 empty()= new This0(Empty.of())
  boolean isEmpty()= this.impl().isEmpty()
  Elem head()= this.impl.asCons().tail()
  This0 tail()=this.impl.asCons().tail()
  This0 cons(Elem e)=new This0(Cons.of(e, this.impl)
  private Impl={interface Bool isEmpty() Cons asCons()}
  private Empty={implements This1
    Bool isEmpty()=true Cons asCons()=../*error*/
    ()//() means no fields
  private Cons={implements This1
    Bool isEmpty()=false Cons asCons()=this
    Elem elem Impl tail }
  Impl impl
}
IntList=list<Elem=Int>
...
IntList.Empty.of().push(3).top()=4 //example usage
```

This would flatten into

```
list=/*as before*/
//IntList=list<Elem=Int>
IntList={
  //Elem={} no more nested class Elem
  static This0 empty()= new This0(Empty.of())
  boolean isEmpty()= this.impl().isEmpty()
  Int head()= this.impl.asCons().tail()
  This0 tail()=this.impl.asCons().tail()
  This0 cons(Int e)=new This0(Cons.of(e, this.impl)
  private Impl={interface Bool isEmpty() Cons asCons()}
  private Empty=/*as before*/}
```

```

198 private Cons={implements This1
199     Bool isEmpty()=false Cons asCons()=this
200     Int elem Impl tail }
201 Impl impl
202 }//everywhere there was "Elem", now there is "Int"

```

Redirect can be propagated in the same way generics parameters are propagate: For example, in Java one could write code as below,

```

206 class ShapeGroup<T extends Shape>{
207     List<T> shapes;
208     ..}
209 //alternative implementation
210 class ShapeGroup<T extends Shape,L extends List<T>>{
211     L shapes;
212     ..}
213

```

to denote a class containing a list of a certain kind of **Shapes**. In our approach, one could write the equivalent

```

217 shapeGroup={
218     Shape={implements Shape}
219     List=list<Elem=Shape>
220     List shapes
221     ..}
222

```

With redirect, `shapeGroup` follow both roles of the two Java examples; indeed there are two reasonable ways to reuse this code

**Triangulation**=`shapeGroup<Shape=Triangle>`, if we have a **Triangle** class and we would like the concrete list type used inside to be local to the **Triangulation**, or **Triangulation**=`shapeGroup<List=Triangles>`, if we have a preferred implementation for the list of triangles that is going to be used by our **Triangulation**. Those two versions would flatten as follow:

```

230 //Triangulation=shapeGroup<Shape=Triangle>
231 Triangulation={
232     List=/*list with Triangle instead of Elem*/
233     List shapes
234     ..}
235
236 //Triangulation=shapeGroup<List=Triangles>
237 //expands to shapeGroup<List=Triangles,Shape=Triangle>
238 Triangulation={
239     Triangles shapes
240     ..}
241

```

As you can see, with redirect we do not decide a priori what is generic and what is not in a class.

Redirect can not always succeed. For example, if we was to attempt `shapeGroup<List=Int>` the flattening process would fail with an error similar to a invalid generic instantiation. Subtype is a fundamental feature of object oriented programming. Our proposed redirect operator do not require the type of the target to perfectly match the structural type of the internal nested classes; structural subtyping is sufficient. This feature adds a lot of flexibility to our redirect, however completing the mapping (as happens in the example above) is a challenging and technically very interesting task when subtyping is took into account. This is strongly connected with ontology matching and will be discussed in the technical core of the paper later on.

## 2.2 Preferential sum and examples of sum and redirect working together

The sum of two traits is conceptually a trait with the sum of the traits members, and the union of the implemented interfaces. If the two traits both define a method with the same name, some resolution strategy is applied. In the symmetric sum<sup>1</sup> the two methods need to have the same signature and at least one of them need to be abstract. With preferential sum (sometimes called override), if they are both implemented, the left implementation is chosen. Since in our model we have nested classes, nested classes with the same name will be recursively composed.

We chose preferential sum since is simpler to use in short code examples.<sup>1</sup> Since the focus of the paper is the novel redirect operator, instead of the well known sum, we will handle summing state and interfaces in the simplest possible way: a class with state can only be summed with a class without state, and an interface can only be summed with another interface with identical methods signatures.

In literature it has been shown how trait composition with (recursively composed) nested classes can elegantly handle the expression problem and a range of similar design challenges. Here we will show some examples where sum and redirect cooperate to produce interesting code reuse patterns:

```
listComp=list<+{
  Elem:{ Int geq(Elem e)}//-1/0/1 for smaller, equals, greater
  static Elem max2(Elem e1, Elem e2)=if e1.geq(e2)>0 then e1, else e2
  Elem max(Elem candidate)=
    if This.isEmpty() then candidate
    else this.tail().max(This.max2(this.head(),candidate))
  Elem min(Elem candidate)=...
  This0 sort()=...
}
```

As you can see, we can *extends* our generic type while refining our generic argument: **Elem** of **listComp** now needs a **geq** method.

While this is also possible with conventional inheritance and F-Bound polymorphism, we think this solution is logically simpler than the equivalent Java

```
class ListComp<Elem extends Comparable<Elem>> extends LinkedList<Elem>{
  ../*body as before*/
}
```

Another interesting way to use sum is to modularize behaviour delegation: consider the following (not efficient for the sake of compactness) implementation of **set**, where the way to compare elements is not fixed:

```
set:{
  Elem:{}
  List=list<Elem=Elem>
  static This0 empty()= new This0(List.empty())
  Bool contains(Elem e)=../*uses eq and hash*/
  Int size()=..
  This add(Elem e)=...
  This remove(Elem e)=...
  Bool eq(Elem e1,Elem e2)//abstract
  Int hash(Elem e)//abstract
  List asList //to allow iteration
}
```

<sup>1</sup> symmetric sum is often presented in conjunction with a restrict operator that makes some methods abstract.

```

307 }
308 eqElem={
309   Elem={ Bool equals(Elem e)/*abstract*/}
310   Bool eq(Elem e1,Elem e2)=e1.equals(e2)
311 }
312 hashElem={
313   Elem={ Int hash(Elem e)/*abstract*/}
314   Int hash(Elem e)=e.hash()
315 }
316 Strings=(set<+eqElem<+eqHash)<Elem=String>
317 LongStrings=(set<+eqElem<Elem=String> <+{
318   Int hash(String e)=e.size()
319 }//for very long strings, size is a faster hash
320

```

321 Note how `(set<+eqElem<+eqHash)<Elem=String>` is equivalent to `set<Elem=String> <+eqElem<Elem=String> <+eqHash<Elem=String>`.

322 Consider now the signature `Bool equals(Elem e)`. This is different from the common signature `Bool equals(Object e)`. What is the best signature for `equals` is an open research question, where most approaches advise either the first or the second one. Our `eqElem`, as I wrote, can support both: `Strings` would be correctly defined both if `String.equals` signature has a `String` or an `Object` parameter. EXPAND on method subtyping.

327 We now consider an example where `Redirect` simplifies the code quite a lot: We have a `Node` and `Edge` concepts for a graph. The `Node` have a list of `Edges`. A `isConnected` function takes a list of `Nodes`. A `getConnected` function takes `Node` and return a set of `Nodes`.

```

330 graphUtils={
331   Edges:list<+{Node start() Node end()}
332   Node:{Edges connections()}
333   Nodes:set<Elem=Node>//note that we do not specify equals/hash
334   static Bool isConnected(Nodes nodes)=
335     if(nodes.size()==0) then true
336     else getConnected(nodes.asList().head()).size()==nodes.size()
337   static Nodes getConnected(Node node)=getConnected(node,Nodes.empty())
338   static Nodes getConnected(Node node,Nodes collected)=
339     if(collected.contains(node)) then collected
340     else connectEdges(node.connections(),collected.add(node))
341   static Nodes connectEdges(Edges e,Nodes collected)=
342     if( e.isEmpty()) then collected
343     else connectEdges(e.tail(),collected.add(e.head().end()))
344 }
345

```

347 We have shown the full code instead of omitting implementations to show that the code inside of an highly general code like the former is pretty conventional. Just declare nested classes as if they was the concrete desired types. Note how we can easily create a new `Nodes@` by doing `Nodes.empty()`.

351 Here we show how to instantiate `graphUtils` to a graph representing cities connected by streets, where the streets are annotated with their length, and `Edges` is a priority queue, to optimize finding the shortest path between cities.

```

354 Map:{
355   Street:{City start, City end, Int size}
356   City:{}
357   Streets:priorityQueue<Elem=Street><+{
358     Int geq(Street e1,Street e2)=e1.size()-e2.size()
359   }<+{
360   Streets:{}
361   City:{Streets connections, Int index}//index identify the node
362   Cities:set<Elem=City><+{
363     Bool eq(City e1, City e2) e1.index==e2.index
364     Int hash(City e) e.index
365   }
366   Cities cities
367

```



## 23:8 Using nested classes as associated types.

```

368 //more methods
369 }
370 MapUtils=graphUtils<Nodes=Map.Cities>
371 //infers Nodes.List, Node, Edges, Edge

```

373 In Appending 2 we will show our best attempt to encode this graph example in Java,  
 374 Rust and Scala. In short, we discovered...

### 375 2.3 Moving traits around in the program

376 It is not trivial to formalize the way types like **This1.A.B** have to be adapted so that when  
 377 code is moved around in differed depths of nesting the refereed classes stay the same. To  
 378 this aim we define a concept of program  $p$ , as a representation of the code as seen from a  
 379 certain point inside of the source code. Each of the  $DV_0 \dots DV_n$  represents the outer nested  
 380 level  $0..n$ , while the  $DL$ s component represent the program top level, that is the outer nested  
 381 level  $n + 1$

$$p ::= DLs; DVs \quad \text{program}$$

$$DL ::= id=L \quad \text{partially-evaluated-declaration}$$

$$DV ::= id=LV \quad \text{evaluated-declaration}$$

$$Mid ::= C \mid m \quad \text{member-id}$$

383 Thus, for example in the program

```

384 A={()}
385 t={ B={()} This1.A m(This0.B b)}
386 C={D={E=t}}
387

```

389 the flattened version for **C** will be  $C=\{D=\{E=\{ B=\{()\} \text{This3.A m(This0.B b)}\}\}$ , where the path  
 390 **This1.A** is now **This3.A** while the path **This0.B** stays the same. The program  $p$  in the  
 391 observation point  $D=t$  is

```

392 A={()}
393 t={ B={()} This1.A m(This0.B b)}
394 C={D={E=t}};
395 C={D={E=t}},
396 D={E=t}
397

```

399 We will use  $p$  as a function, so we will write ..... $p(T)$  and  $p(T.m)$  we define now From, ....  
 $\mathcal{E}_V ::= \square \mid \mathcal{E}_V \leftarrow E \mid LV \leftarrow \mathcal{E}_V \mid \mathcal{E}_V \leftarrow Cs = T > \quad \text{context of library-evaluation}$

400  $\mathcal{E}_v ::= \square \mid \mathcal{E}_v.m(es) \mid v.m(vs \mathcal{E}_v es) \mid T.m(vs \mathcal{E}_v es)$

401 We use  $t$  and  $C$  to syntactically distinguish between trait and class names. An  $E$  is a  
 402 top-level class expression, which can contain class-literals, references to traits, and operations  
 403 on them, namely our sum  $E < +E$  and redirect  $e(Cs = T)$ . A declaration  $D$  is just an  
 404  $id = E$ , representing that  $id$  is declared to be the value of  $E$ , we also have  $CD, CV, DL$ ,  
 405 and  $DV$  that constrain the forms of the LHS and RHS of the declaration. A literal  $L$  has 4  
 406 components, an optional interface keyword, a list of implemented interfaces, a list of members,  
 407 and an optional constructor. For simplicity, interfaces can only contain abstract-methods  
 408 ( $amt$ ) as members, and cannot have constructors. A member  $M$ , is either an (potentially  
 409 abstract) method  $mt$  or a nested class declaration ( $CD$ ). A member value  $MV$ , is a member  
 410 that has been fully compiled. An  $mid$  is an identifier, identifying a member. Constructors,  $K$ ,  
 411 contain a  $Txs$  indicating the type and names of fields. An  $e$  is normal fetherweight-java style  
 412 expression, it has variables  $x$ , method calls  $e.m(es)$ , field accesses  $e.x$  and object creation  
 413  $newes$ .  $CtxV$  is the evaluation context for class-expressions  $E$ , and  $ctxv$  is the usuall one for  
 414  $e$ 's.



415 An  $S$  represents what the top-level source-code form of our language is, it's just a sequence  
 416 of declarations and a main expression. The most interesting form of the grammar is a  $p$ , it is  
 417 a 'program', used as the context for many reductions and typing rules, on the LHS of the ;  
 418 is a stack representing which (nested) declaration is currently being processed, the bottom  
 419 (rightmost)  $DL$  represents the  $D$  of the source-program that is currently being processed.  
 420 The RHS of the ; represents the top-level declarations that have already been compiled, this  
 421 is necessary to look up top-level classes and traits.

422 To look up the value of a type in the program we will use the notation  $p(T)$ , which is defined  
 423 by the following, but only if the RHS denotes an  $LV$ :

$$(\_; \_, C=L, \_)(\text{This}0.C.Cs) := L(Cs)$$

$$424 \quad (id=L, p)(\text{This}0.Cs) := L(Cs)$$

$$425 \quad (id=L, p)(\text{This}n+1.Cs) := p(\text{This}n.Cs)$$

426 To get the relative value of a trait, we define  $p[t]$ :

$$427 \quad (DLs; \_, t=LV, \_)[t] := LV[\text{This}\#DLs]$$

428

429 To get the value of a literal, in a way that can be understood from the current location  
 430 ( $\text{This}0$ ), we define:

$$431 \quad p[T] := p(T)[T]$$

432

433 And a few simple auxiliary definitions:

$$Ts \in p := \forall T \in Ts \bullet p(T) \text{ is defined}$$

$$L(\emptyset) := L$$

$$434 \quad L(C.Cs) := L(Cs) \text{ where } L = \text{interface? } \{ \_ ; \_, C=L, \_ ; \_ \}$$

$$L[C=E'] := \text{interface? } \{ Tz ; MVs C=E' Ms ; K \}$$

$$435 \quad \text{where } L = \text{interface? } \{ Tz ; MVs C=\_ Ms ; K \}$$

## 23:10 Using nested classes as associated types.

We have two-top level reduction rules defining our language, of the form  $Dse^{\sim\sim} > Ds'e$  which simply reduces the source-code. The first rule (*compile*) ‘compiles’ each top-level declaration (using a well-typed subset of already compiled top-level declarations), this reduces the defining expression. The second rule, (*main*) is executed once all the top-level declarations have compiled (i.e. are now fully evaluated class literals), it typechecks the top-level declarations and the main expression, and then proceeds to reduce it. In principle only one-typechecking is needed, but we repeat it to avoid declaring more rules.

```
443 Define Ds e --> Ds' e'
444 =====
445 DVs' |- Ok
446 empty; DVs'; id | E --> E'
447 (compile)----- DVs' subsetof DVs
448 DVs id = E Ds e --> DVs id = E' Ds e
449
450 DVs |- Ok
451 DVs |- e : T
452 DVs |- e --> e'
453 (main)----- for some type T
454 DVs e --> DVs e'
```

### 3 Compilation

Aside from the redirect operation itself, compilation is the most interesting part, it is defined by a reduction arrow  $p; id | E \rightarrow E'$ , the *id* represents the id of the type/trait that we are currently compiling, it is needed since it will be the name of *This0*, and we use that fact that that is equal to *This1.id* to compare types for equality. The (*CtxV*) rule is the standard context, the (*L*) rule propagates compilation inside of nested-classes, (*trait*) merely evaluates a trait reference to its defined body, (*sum*) and (*redirect*) perform our two meta-operations.

```
462 Define p; id | E --> E'
463 =====
464 p; id | E --> E'
465 (CtxV) -----
466 p; id | CtxV[E] --> CtxV[E']
467
468 id = L[C = E], p; C | E --> E'
469 (L) ----- // TODO use fresh C?
470 p; id | L[C = E] --> L[C = E']
471
472 (trait) -----
473 p; id | t -> p[t]
474
475 LV1 <+p' LV2 = LV3 p' = C' = LV3, p
476 (sum) ----- for fresh C'
477 p; id | LV1 <+ LV2 --> LV3
478
479 // TODO: Inline and de-42 redirect formalism
480 (redirect) -----LV'=redirect(p, LV, Cs, P)
481 p; id | LV(Cs=P) -> LV'
```

## 4 The Sum operation

The sum operation is defined by the rule  $L1 < +p L2 = L3$ , it is unconventional as it assumes we already have the result ( $L3$ ), and simply checks that it is indeed correct. We believe (but have not proved) that this rule is unambiguous, if  $L1 < +p L2 = L3$  and  $L1 < +p L2 = L3'$ , then  $L3 = L3'$  (since the order of members does not matter for  $Ls$ ).

The main rule for summing of non-interfaces, sums the members, unions the implemented interfaces (and uses *minimize* to remove any duplicates), it also ensures that at most one of them has a constructor. For summing an interface with a interface/class we require that an interface cannot 'gain' members due to a sum. The actual L42 implementation is far less restrictive, but requires complicated rules to ensure soundness, due to problems that could arise if a summed nested-interface is implemented. Summing of traits/classes with state is a non-trivial problem and not the focus of our paper, there are many prior works on this topic, and our full L42 language simply uses ordinary methods to represent state, however this would take too much effort to explain here.

```

Define L1 <+p L2 = L3
=====
{Tz1; Mz1; K?1} <+p {Tz2; Mz2; K?2} = {Tz; Mz; K?}
Tz = p.minimize(Tz1 U Tz2)
Mz1 <+p Mz1 = Mz
{empty, K?1, K?2} = {empty, K?} //may be too sophisticated?

interface{Tz1; amtz,amtz';} <+p interface?{Tz2;amtz;} = interface {Tz;amtz,amtz';}
Tz = p.minimize(Tz1 U Tz2)
if interface? = interface then amtz'=empty

```

The rules for summing member are simple, we take two sets of members collect all the ones with unique names, and sum those with duplicates. To sum nested classes we merely sum their bodies, to sum two methods we require their signatures to be identical, if they both have bodies, the result has the body of the RHS, otherwise the result has the body (if present) of the LHS.

```

Define Mz <+p Mz' = Mz"
-----
M, Mz <+p M', Mz' = M <+p M', Mz <+p Mz
//note: only defined when M.Mid = M'.Mid

Mz <+p Mz' = Mz, Mz':
dom(Mz) disjoint dom(Mz')

Define M <+p M' = M"
-----
T' m(Txs') e? <+p T m(Txs) e = T m(Txs) e
T', Txs'.Ts =p Ts, Txs

T' m(Txs') e? <+p T m(Txs) = T m(Txs) e?
T', Txs'.Ts =p Ts, Txs

(C = L) <+p (C = L') = L <+p.push(C) L'

```

528 **5** Type System

529 The type system is split into two parts: type checking programs and class literals, and the  
 530 typechecking of expressions. The latter part is mostly conventional, it involves typing judgments  
 531 of the form  $p; Txs \vdash e : T$ , with the usual program  $p$  and variable environment  $Txs$  (often  
 532 called  $\Gamma$  in the literature). rule  $(Dsok)$  type checks a sequence of top-level declarations by  
 533 simply push each declaration onto a program and typecheck the resulting program. Rule  $pok$   
 534 typechecks a program by check the topmost class literal: we type check each of it's members  
 535 (including all nested classes), check that it properly implements each interface it claims to,  
 536 does something weird, and finanly check check that it's constructor only referenced existing  
 537 types,

538

539

540 Define  $p \vdash Ok$ 

541 =====

542

543  $D1; Ds \vdash Ok \dots Dn; Ds \vdash Ok$ 544  $(Ds\ ok) \text{ ----- } Ds = D1 \dots Dn$ 545  $Ds \vdash Ok$ 

546

547  $p \vdash M1 : Ok \dots p \vdash Mn : Ok$ 548  $p \vdash P1 : Implemented \dots p \vdash Pn : Implemented$ 549  $p \vdash implements(Pz; Ms) \text{ /*WTF?*/} \quad \text{if } K? = K: p.exists(K.Txs.Ts)$ 550  $(p\ ok) \text{ ----- } p.top() = interface? \{P1...Pn; M1, \dots, Mn$ 551  $p \vdash Ok$ 

552

553  $p.minimize(Pz) \text{ subseq } p.minimize(p.top().Pz)$ 554  $amt1\_ \text{ in } p.top().Ms \dots amtn\_ \text{ in } p.top().Ms$ 555  $(P\ implemented) \text{ ----- } p[P] = interface \{Pz; amt1 ..$ 556  $p \vdash P : Implemented$ 

557

558  $(amt-ok) \text{ ----- } p.exists(T, Txs.Ts)$ 559  $p \vdash T\ m(Tcs) : Ok$ 

560

561  $p; This0\ this, Txs \vdash e : T$ 562  $(mt-ok) \text{ ----- } p.exists(T, Txs.Ts)$ 563  $p \vdash T\ m(Tcs)\ e : Ok$ 

564

565  $C = L, p \vdash Ok$ 566  $(cd-Ok) \text{ -----}$ 567  $p \vdash C = L : OK$ 

568

569 Rule  $(Pimplemented)$  checks that an interface is properly implemented by the program-  
 570 top, we simply check that it declares that it implements every one of the interfaces super-  
 571 interfaces and methods. Rules  $(amt-ok)$  and  $(mt-ok)$  are straightforward, they both  
 572 check that types mentioned in the method signature exist, and ofcourse for the latter case,  
 573 that the body respects this signature.

574 To typecheck a nested class declaration, we simply push it onto the program and typecheck  
575 the top-of the program as before.

576 The expression typesystem is mostly straightforward and similar to feartherwiegth Java,  
577 notable we we use  $p[T]$  to look up information about types, as it properly ‘from’s paths, and  
578 use a classes constructor definitions to determine the types of fields.

```

579 Define p; Txs |- e : T
580 =====
581 (var)
582 ----- T x in Txs
583 p; Txs |- x : T
584
585 (call)
586 p; Txs |- e0 : T0
587 ...
588 p; Txs |- en : Tn
589 ----- T' m(T1 x1 ... Tn xn) _ in p[T0].Ms
590 p; Txs |- e0.m(e1 ... en) : T'
591
592 (field)
593 p; Txs |- e : T
594 ----- p[T].K = constructor(_ T' x _)
595 p; Txs |- e.x : T'
596
597
598 (new)
599 p; Txs |- e1 : T1 ... p; Txs |- en : Tn
600 ----- p[T].K = constructor(T1 x1 ... Tn xn)
601 p; Txs |- new T(e1 ... en)
602
603
604 (sub)
605 p; Txs |- e : T
606 ----- T' in p[T].Pz
607 p; Txs |- e : T'
608
609
610 (equiv)
611 p; Txs |- e : T
612 ----- T =p T'
613 p; Txs |- e : T'
614
615 - towel1:.. //Map: towel2:.. //Map: lib: T:towel1 f1 ... fn
616   MyProgram: T:towel2 Lib:lib[T=This0.T] ... -

```

## 616 **6** extra

617 Features: Structural based generics embedded in a nominal type system. Code is Nominal,  
618 Reuse is Structural. Static methods support for generics, so generics are not just a trik to

619 make the type system happy but actually change the behaviour Subsume associate types.  
 620 After the fact generics; redirect is like mixins for generics Mapping is inferred-> very large  
 621 maps are possible -> application to libraries

622 In literature, in addition to conventional Java style F-bound polymorphism, there is  
 623 another way to obtain generics: to use associated types (to specify generic paramaters) and  
 624 inheritance (to instantiate the paramaters). However, when parametrizing multiple types,  
 625 the user to specify the full mapping. For example in Java interface A<B> B m(); inteface  
 626 BString f(); class G<TA extends A<TB>, TB>//TA and TB explicitly listed String g(TA  
 627 a TB b)return a.m().f(); class MyA implements A<MyB>.. class MyB implements B ..  
 628 G<MyA,MyB>//instantiation Also scala offers genercs, and could encode the example in  
 629 the same way, but Scala also offers associated types, allowing to write instead....

630 Rust also offers generics and associated types, but also support calling static methods  
 631 over generic and associated types.

632 We provide here a fundational model for genericity that subsume the power of F-bound  
 633 polimorphisms and associated types. Moreover, it allows for large sets of generic parameter  
 634 instantiations to be inferred starting from a much smaller mapping. For example, in our  
 635 system we could just write g= A= method B m() B= method String f() method String g(A a  
 636 B b)=a.m().f() MyA= method MyB m()= new MyB(); .. MyB= method String f()="Hello";  
 637 .. g<A=MyA>//instantiation. The mapping A=MyA,B=MyB

638 We model a minimal calculus with interfaces and final classes, where implementing an  
 639 interface is the only way to induce subtyping. We will show how supporting subtyping  
 640 constitute the core technical difficulty in our work, inducing ambiguity in the mappings.  
 641 As you can see, we base our generic matches the structor of the type instead of respect-  
 642 ing a subtype requirement as in F-bound polymorphis. We can easily encode subtype  
 643 requirements by using implements: Print=interface method String print(); g= A:implements  
 644 Print method A printMe(A a1,A a2) if(a1.print().size())>a2.print.size())return a1; return a2;  
 645 MyPrint=implements Print .. g<A=MyPrint> //instantiation g<A=Print> //works too  
 646 ————— example showing ordering need to strictly improve EI1: interface EA1: imple-  
 647 ments EI1

648 EI2: interface EA2: implements EI2  
 649 EB: EA1 a1 EA1 a1  
 650 A1: A2: B: A1 a1 A2 a2 [B = EB] // A1 -> EI1, A2 -> EA2 a // A1 -> EA1, A2 ->  
 651 EI2 b // A1 -> EA1, A2 -> EA2 c  
 652 a <=b b <=a c<= a,b a <= c

653 **hi Hi class**  

$$a ::= b \quad c$$
  
 654 **aahiHi class**  

$$a ::= b \quad c$$
  

$$a ::= b \quad c$$
  
 655 **}}[(O]**  
 (TOP)  

$$\frac{a \rightarrow c \quad \forall i < 3a \vdash b : \text{OK}}{\frac{1 + 2 \rightarrow 3}{c} \quad a}$$
  
 656