

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. 23:1–23:14

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  
66
```

67 Note how in this case, class **Result.Box** would exists. 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 show 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 applies to such redirect, we will show how
88 supporting (A) greatly improve their value.

89 The formal core of our work is in defining

- 90 ■ **ValidRedirect**, a computable predicate telling if a mapping respect the structural shapes
91 and nominal subtype relations.
- 92 ■ **BestRedirect**, a formal definition of what properties a procedure expanding a partial
93 mapping into a 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

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*.

$e ::= x \mid e.m(es) \mid T.m(es) \mid e.x \mid \text{new } T(es)$	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
$K ::= (Txz)?$	state	$id ::= C \mid t$	class/trait id
$E ::= L \mid t \mid E_1 <+ E_2 \mid E <R>$	Code Expr.	$v ::= \text{new } T(vs)$	value
$R ::= Cs_1 = T_1 \dots Cs_n = T_n$	redirect map	$LV ::= \dots$	

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: This_0 is the current class, This_1 is the enclosing class, 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 This_n . Thus, 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. To simplify our formalism, we delegate some sanity checks well formedness, and we assume all the fields in the state K to have different names; no two methods or nested classes with the same name (m or C) are declared in a code literal, and no nested class is named This_n for any number n ; in any method headers, all parameters have different names, and no parameter is named **this**.

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. We assume a form of alpha-rename for private nested classes, that will consistently rename all the paths of form $\text{This}_n.C.Cs'$, where $\text{This}_n.C$ refer to such private nested class. The trivial definition of such alpha rename is given in appendix.

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 This0(xs)** in method bodies and of form v in the main expression. Those restrictions greatly simplify reasoning about code reuse, since they require different

140 classes to only communicate by calling (static) methods. Supporting unrestricted fields
 141 and constructors would make the formalism much more involved without adding much of a
 142 conceptual difficulty. Values are of form `new T(vs)`.

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

147 Finally, we examine the shape of a nested class: `private? C=E`. The right hand side
 148 is not just a code literal but a code composition expression E . In trait composition, the
 149 code expression will be reduced/flattened to a code literal L during compilation. Code
 150 expressions denote an algebra of code composition, starting from code literal L and trait
 151 names t , referring to a literal declared before by $t=E$. We consider two operators: conventional
 152 preferential sum $E_1 \lt+ E_2$ and our novel redirect $E \lt Cs = T \gt$.

153 2.1 Compilation process/flattening

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

168 2.1.1 Redirect

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

```
172 list={
173   Elem={}
174   static This0 empty()= new This0(Empty.of())
175   boolean isEmpty()= this.impl().isEmpty()
176   Elem head()= this.impl.asCons().tail()
177   This0 tail()=this.impl.asCons().tail()
178   This0 cons(Elem e)=new This0(Cons.of(e, this.impl)
179   private Impl={interface Bool isEmpty() Cons asCons()}
180   private Empty={implements This1
181     Bool isEmpty()=true Cons asCons()=../*error*/
182     ()//() means no fields
183   private Cons={implements This1
184     Bool isEmpty()=false Cons asCons()=this
185     Elem elem Impl tail }
186     Impl impl
187   }
188 }
189 IntList=list<Elem=Int>
```

```

190 ...
191 IntList.Empty.of().push(3).top()==4 //example usage
192

```

193 This would flatten into

```

194 list={/*as before*/
195 //IntList=list<Elem=Int>
196 IntList={
197   //Elem={} no more nested class Elem
198   static This0 empty()= new This0(Empty.of())
199   boolean isEmpty()= this.impl().isEmpty()
200   Int head()= this.impl.asCons().tail()
201   This0 tail()=this.impl.asCons().tail()
202   This0 cons(Int e)=new This0(Cons.of(e, this.impl)
203   private Impl={interface Bool isEmpty() Cons asCons()}
204   private Empty={/*as before*/}
205   private Cons={implements This1
206     Bool isEmpty()=false Cons asCons()=this
207     Int elem Impl tail }
208   Impl impl
209 } //everywhere there was "Elem", now there is "Int"
210

```

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

```

214 class ShapeGroup<T extends Shape>{
215   List<T> shapes;
216   ..}
217 //alternative implementation
218 class ShapeGroup<T extends Shape,L extends List<T>>{
219   L shapes;
220   ..}
221

```

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

```

225 shapeGroup={
226   Shape={implements Shape}
227   List=list<Elem=Shape>
228   List shapes
229   ..}
230

```

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

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

```

238 //Triangulation=shapeGroup<Shape=Triangle>
239 Triangulation={
240   List=/*list with Triangle instead of Elem*/
241   List shapes
242   ..}
243
244 //Triangulation=shapeGroup<List=Triangles>
245 //expands to shapeGroup<List=Triangles,Shape=Triangle>
246 Triangulation={
247   Triangles shapes
248   ..}
249

```

251 As you can see, with redirect we do not decide a priori what is generic and what is not in a
 252 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.

2.1.2 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[] 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.¹ 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 then the equivalent Java

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

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

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:

```

303 set:{
304   Elem:{
305     List=list<Elem=Elem>
306     static This0 empty()= new This0(List.empty())
307     Bool contains(Elem e)=../*uses eq and hash*/
308     Int size()=..
309     This add(Elem e)=...
310     This remove(Elem e)=...
311     Bool eq(Elem e1,Elem e2)//abstract
312     Int hash(Elem e)//abstract
313     List asList //to allow iteration
314   }
315   eqElem={
316     Elem={ Bool equals(Elem e)/*abstract*/
317           Bool eq(Elem e1,Elem e2)=e1.equals(e2)
318         }
319     hashElem={
320       Elem={ Int hash(Elem e)/*abstract*/
321             Int hash(Elem e)=e.hash()
322           }
323     }
324   Strings=(set<+eqElem<+eqHash><Elem=String>
325   LongStrings=(set<+eqElem><Elem=String> <+{
326     Int hash(String e)=e.size()
327   }//for very long strings, size is a faster hash

```

Note how `(set<+eqElem<+eqHash><Elem=String>)` is equivalent to `set<Elem=String> <+eqElem<Elem=String> <+eqHash<Elem=String>`. 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 define both if `String.equals` signature has a `String` or an `Object` parameter. EXPAND on method subtyping.

2.2 Moving traits around in the program

It is not trivial to formalize the way types like `This1.A.B` have to be adapted so that when code is moved around in different depths of nesting the refereed classes stay the same. This is needed during flattening, when a trait t is reused, but also during reduction, when a method body is inlined in the main expression, and during typing, where a method body is typed depending on the signature of other methods in the system.

To this aim we define a concept of program $p ::= Ds; DVz$ where $DV ::= id=LV$; as a representation of the code as seen from a certain point inside of the source code. It is the most interesting form of the grammar, used for virtually all reduction and typing rules. On the left of the ‘;’ is a stack representing which (nested) declaration is currently being processed, the bottom of the stack (rightmost) D represents the top level declaration of the source-program that is currently being processed, while the other elements of the stack are nested classes nested inside of each other. The right of the ‘;’ represents the top-level declarations that have already been compiled, this is necessary to look up top-level classes and traits. Summarizing, each of the $D_0 \dots D_n$ represents the outer nested level $0..n$, while the DV s component represent the already flattened portion of the program top level, that is the outer nested level $n + 1$. Thus, for example in the program

```

352 A={()}
353 t={ B={()}   This1.A m(This0.B b)}
354

```


23:8 Using nested classes as associated types.

```

355 C={D={E=t}}
356 H=t<B=A>

```

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

```

361 A={()}
362 t={ B={()} This1.A m(This0.B b)}
363 C={D={E=t}};
364 C={D={E=t}}, //this means, we entered in C
365 D={E=t} //this means, we entered in D

```

In order to fetch code literals from the program, while transforming the types so that they keep referring to the same nested classes, we rely on notations $p[T]$ and $p[t]$. Those notations extract a class or a trait from a program while consistently transforming types. We also use notation $L[C = E]$ to update the code expression in C to E . For space reasons, those notations are defined in the appendix.

3 Flattening

Aside from the redirect operation itself, compilation/flattening is the most interesting part, it is defined by reduction arrow $Ds \Rightarrow Ds'$, where eventually Ds' is going to reach form DVs and $p; id \vdash E \Rightarrow E'$, where eventually E' is going to reach form LV . The id represents the identifier 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. Rule (TOP) selects the leftmost $id=E$ where E is not of form LV and DVz : a well typed subset of the preceeding declarations. E is flattened in the context of such DVz , thus by rule (TRAIT) DVz must contain all the trait names used in E . In the judgement $p; id \vdash E \Rightarrow E'$ id is only used in order to grow the program p in rule (L-ENTER), and p itself is only needed for (REDIRECT). The (CTXV) rule is the standard context, the (L-ENTER) rule propagates compilation inside of nested-classes, (TRAIT) merely evaluates a trait reference to it's defined body, finally (SUM) and (REDIRECT) perform our two meta-operations by propagating to corresponding auxiliary definitions. Rule (SUM) just delegate the work on the auxiliary notation defined below:

```

367 Def:  $L_1 \lt+ L_2 = \text{interface? } \{Tz_1 \cup Tz_2; Mz \lt+ Mz', Mz_1, Mz_2; K?\}$ 
       $L_1 = \text{interface? } \{Tz_1; Mz, Mz_1; K?_1\}$        $L_2 = \text{interface? } \{Tz_2; Mz', Mz_2; K?_2\}$ 
       $\{empty, K?_1, K?_2\} = \{empty, K?\}$ 
368   if  $\text{interface?} = \text{interface}$  then  $mdom(L_1) = mdom(L_2)$ 
      Def:  $Tm(Txs)e? \lt+ Tm(Txs)e = Tm(Txs)e$ 
      Def:  $Tm(Txs)e? \lt+ Tm(Txs) = Tm(Txs)e?$ 
      Def:  $(C=L) \lt+ (C=L') = C= L \lt+ L,$ 

```

As usual in definitions of sum operators, the implemented interfaces is the union of the interfaces of L_1 and L_2 , the members with the same domain are recursively composed while the members with disjoint domains are directly included. Since method and nested class identifiers must be unique in a well formed L and $M_1 \lt+ M_2$ being defined only if the identifier is the same, our definition forces $dom(Mz) = dom(Mz')$ and $dom(Mz_1)$ disjoint $dom(Mz_2)$. For simplicity here we require at most one class to have a state; if both have no state, the result will have no state, otherwise the result will have the only present state (the set $\{empty, K?\}$ mathematically express this requirement in a compact way); we also allow summing only interfaces with interfaces and final classes with final classes. When two interfaces are composed both sides must define the same methods. This is because

■ **Figure 1** Flattening

Def: $Ds \Rightarrow Ds'$ and $p; id \vdash E \Rightarrow E'$, where $\mathcal{E}_V ::= \square \mid \mathcal{E}_V <+ E \mid LV <+ \mathcal{E}_V \mid \mathcal{E}_V < Cs = T >$
(TOP)

$$\begin{array}{c}
 DVz \subseteq DVs \\
 DVz \vdash \mathbf{Ok} \\
 \hline
 empty; DVz; id \vdash E \Rightarrow E' \\
 \hline
 DVs \text{ id}=EDs \Rightarrow DVs \text{ id}=E'Ds
 \end{array}
 \quad
 \begin{array}{c}
 \text{(L-ENTER)} \\
 \hline
 p \cdot \mathbf{push}(id=L[C=E]); C \vdash E \Rightarrow E' \\
 \hline
 p; id \vdash L[C=E] \Rightarrow L[C=E']
 \end{array}
 \quad
 \begin{array}{c}
 \text{(TRAIT)} \\
 \hline
 p; id \vdash t \Rightarrow p[t]
 \end{array}$$

$$\begin{array}{c}
 \text{(REDIRECT)} \\
 \hline
 p' = p \cdot \mathbf{push}(id=LV) \\
 R' = p' \cdot \mathbf{bestRedirection}(R[\mathbf{This}_n = \mathbf{This}_{n+1}]) \\
 Csz = p' \cdot \mathbf{redirectSet}(R) \\
 p' \cdot \mathbf{redirectable}(Csz) \\
 L = p' \cdot \mathbf{minimize}(LV) \cdot \mathbf{remove}(Csz) \\
 \hline
 p; id \vdash LV < R > \Rightarrow R'(L)
 \end{array}$$

$$\begin{array}{c}
 \text{(SUM)} \\
 \hline
 L_i = p \cdot \mathbf{push}(id=LV_i)[\mathbf{This}_0] \\
 L_1 <+ L_2 = LV \\
 \hline
 p; id \vdash LV_1 <+ LV_2 \Rightarrow LV
 \end{array}$$

other nested classes inside L_1 may be implementing such interface, and adding methods to such interface would require those classes to somehow add an implementation for those methods too. In literature there are expressive ways to soundly handle merging different state, composing interfaces with final classes and adding methods to interfaces, but they are out of scope in this work.

Member composition $M_1 <+ M_2$ uses the implementation from the right hand side, if available, otherwise if the right hand side is abstract, the body is taken from the left side. Composing nested classes, not how they can not be **private**; it is possible to sum two literals only if their private nested classes have different private names. This constraint can always be obtained by alpha-renaming them.

Rule (REDIRECT) is the center of our interest for this work. To have a single data structure p where all the types correctly point to the corresponding nested classes, we add the L to the top of our current program, and we add 1 to all the types provided in the redirect map, since they were relative to p and not p' . We use $p' \cdot \mathbf{bestRedirection}()$ in order to complete the provided mapping R into a complete mapping. Such mapping must be of the correct Something does not work: either we check $\text{dom}(R') = \text{Csz}$ or we use $\text{dom}(R')$ instead of Csz . What is the right one?

We will comment on the formal definitions used in (REDIRECT) in the next section, that is going to be the formal core of the paper.

For space reason, the type system and the reduction of the main program are in appendix. They are very straightforward: thanks to flattening, they are a simple nominal type system and reduction over a FJ-like language, with no generics or special method dispatch rules.

4 Redirect in the details

5 Appendix?

$\mathcal{E}_V ::= \square \mid \mathcal{E}_V <+ E \mid LV <+ \mathcal{E}_V \mid \mathcal{E}_V < Cs = T >$ context of library-evaluation
 $\mathcal{E}_v ::= \square \mid \mathcal{E}_v \cdot m(es) \mid v \cdot m(vs \ \mathcal{E}_v \ es) \mid T \cdot m(vs \ \mathcal{E}_v \ es)$

6 Type System

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

```

436 Define p |- Ok
437 =====
438
439 D1; Ds |- Ok ... Dn; Ds|- Ok
440 (Ds ok) ----- Ds = D1 ... Dn
441 Ds |- Ok
442
443 p |- M1 : Ok .... p |- Mn : Ok
444 p |- P1 : Implemented .... p |- Pn : Implemented
445 p |- implements(Pz; Ms) /*WTF?*/                if K? = K: p.exists(K.Txs.Ts)
446 (p ok) ----- p.top() = interface? {P1...Pn; M1, ..., Mn
447 p |- Ok
448
449 p.minimize(Pz) subseteq p.minimize(p.top().Pz)
450 amt1 _ in p.top().Ms ... amtn _ in p.top().Ms
451 (P implemented) ----- p[P] = interface {Pz; amt1 ..
452 p |- P : Implemented
453
454 (amt-ok) ----- p.exists(T, Txs.Ts)
455 p |- T m(Tcs) : Ok
456
457 p; This0 this, Txs |- e : T
458 (mt-ok) ----- p.exists(T, Txs.Ts)
459 p |- T m(Tcs) e : Ok
460
461 C = L, p |- Ok
462 (cd-Ok) -----
463 p |- C = L : OK
464

```

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

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

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

```

475 Define p; Txs |- e : T
476 =====
477 (var)
478 ----- T x in Txs
479 p; Txs |- x : T
480
481 (call)
482 p; Txs |- e0 : T0
483 ...
484 p; Txs |- en : Tn
485 ----- T' m(T1 x1 ... Tn xn) _ in p[T0].Ms
486 p; Txs |- e0.m(e1 ... en) : T'
487
488 (field)
489 p; Txs |- e : T
490 ----- p[T].K = constructor(_ T' x _)
491 p; Txs |- e.x : T'
492
493
494 (new)
495 p; Txs |- e1 : T1 ... p; Txs |- en : Tn
496 ----- p[T].K = constructor(T1 x1 ... Tn xn)
497 p; Txs |- new T(e1 ... en)
498
499
500 (sub)
501 p; Txs |- e : T
502 ----- T' in p[T].Pz
503 p; Txs |- e : T'
504
505
506 (equiv)
507 p; Txs |- e : T
508 ----- T =p T'
509 p; Txs |- e : T'

```

510 7 Graph example

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

```

514 graphUtils={
515   Edges:list<+{Node start() Node end()}
516

```

23:12 Using nested classes as associated types.

```

517 Node:{Edges connections()}
518 Nodes:set<Elem=Node> //note that we do not specify equals/hash
519 static Bool isConnected(Nodes nodes)=
520   if(nodes.size()==0) then true
521   else getConnected(nodes.asList().head()).size()==nodes.size()
522 static Nodes getConnected(Node node)=getConnected(node,Nodes.empty())
523 static Nodes getConnected(Node node,Nodes collected)=
524   if(collected.contains(node)) then collected
525   else connectEdges(node.connections(),collected.add(node))
526 static Nodes connectEdges(Edges e,Nodes collected)=
527   if( e.isEmpty()) then collected
528   else connectEdges(e.tail(),collected.add(e.head().end()))
529 }

```

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().

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.

```

538 Map:{
539   Street:{City start, City end, Int size}
540   City:{}
541   Streets:priorityQueue<Elem=Street><+{
542     Int geq(Street e1,Street e2)=e1.size()-e2.size()
543   }>+{
544     Streets:{}
545     City:{Streets connections, Int index} //index identify the node
546     Cities:set<Elem=City><+{
547       Bool eq(City e1,City e2) e1.index==e2.index
548       Int hash(City e) e.index
549     }
550     Cities cities
551     //more methods
552   }
553   MapUtils=graphUtils<Nodes=Map.Cities>
554   //infers Nodes.List, Node, Edges, Edge
555 }

```

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

FROM and minimize that will go in the appendix:

To fetch a trait form a program, we will use notation $p(t) = LV$, to fetch a class we will use $p(T)$.

To look up the definition of a class in the program we will use the notation $p(T) = LV$, which is defined by the following:

$$\begin{aligned}
 (DLs; DVs).push(id=L) &:= id=L, DLs; DVs \\
 (; _, C=L, _)(\text{This}_0.C.Cs) &:= L(Cs) \\
 p.push(_=L)(\text{This}_0.Cs) &:= L(Cs) \\
 p.push(_)(\text{This}_{n+1}.Cs) &:= p(\text{This}_n.Cs) \\
 LV(\emptyset) &:= LV \\
 L(C.Cs) &:= L_0(Cs) \text{ where } L = \text{interface? } \{ _; _, C=L_0, _; _ \}
 \end{aligned}$$

where $L = a$

This notation just fetch the referred LV without any modification. To adapt the paths we define $T_0.\text{from}(T_1,j)$, $L.\text{from}(T,j)$ and $p.\text{minimize}(T)$ as following:

$$\begin{aligned} \text{This}_n.Cs.\text{from}(T,j) &:= \text{This}_n.Cs \quad \text{with } n < j \\ \text{This}_{n+j}.Cs.\text{from}(\text{This}_m.C_1 \dots C_{k,j}) &:= \text{This}_{m+j}.C_1 \dots C_{k-n} \quad \text{with } n \leq k \\ \text{This}_{n+j}.Cs.\text{from}(\text{This}_m.C_1 \dots C_{k,j}) &:= \text{This}_{m+j+n-k}.C_1 \dots C_{k-n}Cs \quad \text{with } n > k \\ \{\text{interface?}Tz; Mz; K\}.\text{from}(T,j-1) &:= \{\text{interface?}Tz.\text{from}(T,j); Mz.\text{from}(T,j); K.\text{from}(T,j)\} \\ p.\text{minimize}(T) &:= T' \dots \\ \text{Finally, we combine those to notation for the most common task of getting the value of a} \\ \text{literal, in a way that can be understood from the current location: } p[t] \text{ and } p[T]: \\ (DL_1 \dots DL_n; _, t=LV, _)[t] &:= LV.\text{from}(\text{This}_n) \\ p[T] &:= p.\text{minimize}(p(T).\text{from}(T)) \end{aligned}$$

```

- towel1:.. //Map: towel2:.. //Map: lib: T:towel1 f1 ... fn
MyProgram: T:towel2 Lib:lib[T=This0.T] ... -

```

8 extra

Features: Structural based generics embedded in a nominal type system. Code is Nominal, Reuse is Structural. Static methods support for generics, so generics are not just a trik to make the type system happy but actually change the behaviour Subsume associate types. After the fact generics; redirect is like mixins for generics Mapping is inferred-> very large maps are possible -> application to libraries

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

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

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

We model a minimal calculus with interfaces and final classes, where implementing an interface is the only way to induce subtyping. We will show how supporting subtyping constitute the core technical difficulty in our work, inducing ambiguity in the mappings. As you can see, we base our generic matches the structor of the type instead of respecting a subtype requirement as in F-bound polymorphis. We can easily encode subtype requirements by using implements: Print=interface method String print(); g= A:implements Print method A printMe(A a1,A a2) if(a1.print().size()>a2.print.size())return a1; return a2; MyPrint=implements Print .. g<A=MyPrint> //instantiation g<A=Print> //works too
————— example showing ordering need to strictly improve EI1: interface EA1: implements EI1
EI2: interface EA2: implements EI2

23:14 Using nested classes as associated types.

```
610      EB: EA1 a1 EA1 a1
611      A1:  A2:  B: A1 a1 A2 a2 [B = EB] // A1 -> EI1, A2 -> EA2 a // A1 -> EA1, A2 ->
612 EI2 b // A1 -> EA1, A2 -> EA2 c
613      a <=b b <=a c<= a,b a <= c
614      hi Hi class
                                a ::= b  c
615      aaHiHi class qq a ::= b  c
                                a ::= b  c
616      }}[(())]
      (TOP)
      a → c  ∀i < 3a ⊢ b : OK
      b
617      -----
      1 + 2 → 3
                                a
                                b
                                c
```