

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 du. 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, called 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. As in many trait languages, we support **This** to refer to the current class. It is needed so that a method inside of a trait can refer to its eventual type. 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 and constructors to be always private. In our code examples we assume standard getters and setters to be automatically declared, together with a **static** method `of(..)` that would contain a standard constructor call, 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()=this.inner//implicit
  static method This of(String inner)=new This(inner)//implicit
myTtrait={
  Box={Elem inner}//implicit This of(Elem inner) and Elem inner()
  Elem={This concat(This that)}
  static method Box merge(Box b,Elem e){return Box.of(b.inner().concat(e));}
}
```



© 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. 23:1–23:20

Leibniz International Proceedings in Informatics



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

23:2 Using nested classes as associated types.

```
47 Result=myTrait<Box=SBox>//equivalent to trait<Box=SBox, Elem=String>
48 ...Result.merge(SBox.of("hello "), "world");//hello world
```

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

```
54 Result={static method SBox merge(SBox b,String e){
55     return SBox.of(b.inner().concat(e));}}
56
```

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

Redirect could be applied in other ways; **Result2=myTrait<Elem=String>** for example would flatten into:

```
64 Result2={
65     Box={String inner}
66     static method Box merge(Box b,String e){
67         return Box.of(b.inner().concat(e));}}
68
```

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

This example shows many of the characteristics of our approach:

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

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

The formal core of our work is in defining

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

Before diving in the formal details, we show an example motivating that expanding the redirect map is not trivial when subtyping is taken into consideration. Consider an interface **ColorPoint** implementing **Point** and **Root**, **Left**, **Right** and **Merge** forming a diamond interface implementation, where method `m` return type is refined in **Right**, and thus stay refined in **Merge**:

```

104 Point=interface{ ...}
105 ColorPoint=interface{ implements Point ...}
106 Root=interface{Point m()}
107 Left={interface implements EA Point m()}
108 Right={interface implements EA ColorPoint m()}
109 Merge={implements Left, Right ColorPoint m()}
110 C={ Merge bind()}
111

```

Trait `t` contains **Target** with a method returning a **Result**, that implements an interface **I** with a method returning a **ColorPoint**. We include an abstract method `show` reporting in its signature **Target**, **Result** and **I**, so we can see where are they redirected to.

```

116 t={
117   I=interface{ColorPoint m()}
118   Result=interface{implements I ColorPoint m()}
119   Target={Result bind()}
120   Target show(Result r, I i)
121 }
122 Res=t<Target=C>
123

```

The big question is, what is the complete mapping inferred from `t<Target=C>`? Naively, if **Target=C**, since both **Target** and **C** have a method `bind`, we could connect their result types: **Result=Merge**. This is not acceptable, since **Result** is an interface while **Merge** is not, and more (possibly private) members inside `t` may be currently implementing **Result**, even if such members are not present now, it would be reasonable if they were added in the future, and we want our inferred map to be stable to such additions. Note however that is safe to redirect result to any interface implemented by **Merge**. Thus we have three possibilities: **Left**, **Right** and indirectly **Root**. The only possibility is **Result=Right**, since the method `m` need to return a **ColorPoint**. However, **Result** implements **I**, so also **I** need to be redirected, but to what? all possible supertypes of **Right** are a possible option, so in this case **Root** and **Right** itself. The only option here is **Right**, again method `m` need to return a **ColorPoint**. Thus, the final mapping is **Target=C, Result=Right, I=Right** and the flattening result would be `Res={C show(Right r, Right i)}`. Subtyping 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 taken into account. This is strongly connected with ontology matching and will be discussed in the technical core of the paper later on.

We first formally define our core language, then we define our redirect operator and its formal properties. Finally we motivate our model showing how many interesting examples of generics and associated types can be encoded with redirect. Finally, as an extreme application, 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. 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
$L ::= \{ \text{interface } Tz; M\bar{s} \mid \{ Tz; Mz; K \}$	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 . By using a larger then needed n , there could be multiple different types referring to the same class. We require all types to be in the form where the smallest possible n is used.

Code literals L serve the role of class/trait 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 to well formedness: all the fields in the state K 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.

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 underpinning. 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

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 $E \leftarrow Cs = T$.

2.1 Compilation process/flattening

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 `private? 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 flattened 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.

2.1.1 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()
```

23:6 Using nested classes as associated types.

```
247 This0 cons(Int e)=new This0(Cons.of(e, this.impl)
248 private Impl={interface Bool isEmpty() Cons asCons()}
249 private Empty={/*as before*/}
250 private Cons={implements This1
251     Bool isEmpty()=false Cons asCons()=this
252     Int elem Impl tail }
253 Impl impl
254 }//everywhere there was "Elem", now there is "Int"
255
```

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

```
258 class ShapeGroup<T extends Shape>{
259     List<T> shapes;
260     ..}
261 //alternative implementation
262 class ShapeGroup<T extends Shape,L extends List<T>>{
263     L shapes;
264     ..}
265
266
```

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

```
269 shapeGroup={
270     MyShape={implements Shape}
271     List=list<Elem=MyShape>
272     List shapes
273     ..}
274
275
```

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

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

```
283 //Triangulation=shapeGroup<MyShape=Triangle>
284 Triangulation={
285     List=/*list with Triangle instead of Elem*/
286     List shapes
287     ..}
288
289 //Triangulation=shapeGroup<List=Triangles>
290 //expands to shapeGroup<List=Triangles,MyShape=Triangle>
291 Triangulation={
292     Triangles shapes
293     ..}
294
295
```

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

297 Redirect can not always succeed. For example, if we was to attempt `shapeGroup<List=Int>`
298 the flattening process would fail with an error similar to a invalid generic instantiation.

299 2.1.2 Preferential sum; sum and redirect working together

300 The sum of two traits is conceptually a trait with the sum of the traits members, and the
301 union of the implemented interfaces. If the two traits both define a method with the same
302 name, some resolution strategy is applied. In the symmetric sum[] the two methods need
303 to have the same signature and at least one of them need to be abstract. With preferential
304 sum (sometimes called override), if they are both implemented, the right implementation

is chosen and the left one is discarded. 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:

```

316 listComp=list<+{
317   Elem:{ Int geq(This e)}// -1/0/1 for smaller, equals, greater
318   static Elem max2(Elem e1, Elem e2)=if e1.geq(e2)>0 then e1, else e2
319   Elem max(Elem candidate)=
320     if This.isEmpty() then candidate
321     else this.tail().max(This.max2(this.head(), candidate))
322   Elem min(Elem candidate)=...
323   This0 sort()=...
324 }
325
326

```

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

```

331 class ListComp<Elem extends Comparable<Elem>> extends LinkedList<Elem>{
332   ../*body as before*/
333 }
334
335

```

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:

```

339 set:{
340   Elem:{
341     List=list<Elem=Elem>
342     static This0 empty()= new This0(List.empty())
343     Bool contains(Elem e)=../*uses eq and hash*/
344     Int size()=..
345     This add(Elem e)=...
346     This remove(Elem e)=...
347     Bool eq(Elem e1,Elem e2)//abstract
348     Int hash(Elem e)//abstract
349     List asList //to allow iteration
350   }
351   eqElem={
352     Elem={ Bool equals(Elem e)//abstract*/}
353     Bool eq(Elem e1,Elem e2)=e1.equals(e2)
354   }
355   hashElem={
356     Elem={ Int hash(Elem e)//abstract*/}
357     Int hash(Elem e)=e.hash()
358   }
359   Strings=(set<+eqElem<+eqHash)<Elem=String>
360

```

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


```

361 LongStrings=(set<+eqElem><Elem=String> <+{
362   Int hash(String e)=e.size()
363 }//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 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 written, 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 DVs 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

```

389 A={()}
390 t={ B={()}   This1.A m(This0.B b)}
391 C={D={E=t}}
392 H=t<B=A>

```

the flattened body of `C.D.E` will be `{ B={()} 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

```

398 A={()}
399 t={ B={()}   This1.A m(This0.B b)}
400 C={D={E=t}};
401 C={D={E=t}}; //this means, we entered in C
402 D={E=t} //this means, we entered in D

```

In order to fetch the code literals corresponding to t , we define notation $p[t]$ ($=\{ B={()} This3.A m(This0.B b) \}$). Such notation transforms the types so that they keep referring to the same nested classes. We also rely on the notation $p[T]$, to extract just methods and the list of implemented interfaces, in a form where they are useful for direct comparison with T . for example, if the program contains `{B={} This0 m(This0.B x)}` in position `This2.A`, $p[\text{This2.A}]$ would be `{This2.A m(This2.A.B x)}`.

We now present formal definition for those operations. We will use members Mz as a function containing both method names m and class names C in its domain; thus we will

assume notation $dom(Mz)$, $Mz(m)$, $Mz(C)$ with the usual meaning. Under here, we define useful auxiliary notations to access literals L with functional notation with the intent of accessing their members. We define notations $L[Cs = E] = L'$ and $Mz[C = E] = Mz'$ serving the role of function update. We use those notations to define $p(T) = LV$ accessing a program p as function. We also define operations on programs: $p.\text{push}(D) = p'$, allowing to work with programs as if they was stacks, and $p.\text{min}(T) = T'$, denoting the shortest type T' referring to the same nested class of T . We define $T.\text{from}(T',j)$ and $L.\text{from}(T,j)$; we omit all the trivial propagation cases of form $M.\text{from}(T,j)$, $K.\text{from}(T,j)$ and $e.\text{from}(T,j)$.

$(DLs; DVs).\text{push}(id=L) = id=L, DLs; DVs$	$(Mz, \text{private}?C = _)[C = E] = Mz, \text{private}?C = E$
$(_, C=L, _)(\text{This}_0.C.Cs) = L(Cs)$	$LV(\emptyset) = LV$
$p.\text{push}(_ = L)(\text{This}_0.Cs) = L(Cs)$	$L(C.Cs) = L(C)(Cs)$
$p.\text{push}(_)(\text{This}_{n+1}.Cs) = p(\text{This}_n.Cs)$	$L[\text{empty} = E] = E$
$\text{members}(\text{interface}? \{ _, Mz; _ \}) = Mz$	$\text{interface}? \{Tz; Mz; K?\}[C.Cs = E] =$
$L(m) = \text{members}(L)(m)$	$\text{interface}? \{Tz; Mz[C = Mz(C)][Cs = E]\}; K?\}$
$L(C) = \text{members}(L)(C)$	$p.\text{min}(\text{This}_{n+1}.id_n.Cs) = p.\text{min}(\text{This}_n.Cs)$
$dom(L) = dom(\text{members}(L))$	where $p = id_0=L_0 \dots id_n=L_n _; Ds$
$m\text{dom}(L) = \{m \in dom(L)\}$	otherwise $p.\text{min}(T) = T$
$\text{This}_n.Cs.\text{from}(T,j) = \text{This}_n.Cs$ with $n < j$	
$\text{This}_{n+j}.Cs.\text{from}(\text{This}_m.C_1 \dots C_k, j) = \text{This}_{m+j}.C_1 \dots C_{k-n}$ with $n \leq k$	
$\text{This}_{n+j}.Cs.\text{from}(\text{This}_m.C_1 \dots C_k, j) = \text{This}_{m+j+n-k}.C_1 \dots C_{k-n}Cs$ with $n > k$	
$\{\text{interface}?Tz; Mz; K\}.\text{from}(T,j-1) = \{\text{interface}?Tz.\text{from}(T,j); Mz.\text{from}(T,j); K.\text{from}(T,j)\}$	
$(DL_1 \dots DL_n; _, t=LV)[t] = p.\text{min}(LV.\text{from}(\text{This}_n, 0))$	
$p[T] = p.\text{min}(\text{interface}? \{Tz.\text{from}(T,0); Mz.\text{from}(T,0); \})$ where $p(T) = \text{interface}? \{Tz; Mz; K?\}$	

The type system and the reduction of the main program are in appendix. They are very straight forward: 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.

3 Flattening

Flattening 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 This_0 , and we use to the fact that refers to the same nested class as $\text{This}_1.id$. 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. We will present those two rules in the two sections below. Note how we require their input to be already in the *minimized* form, that is, all the T uses the shortest way to refer to their corresponding nested class. This prevents the programmer from expressing some difficult cases. Consider for example using two different ways to refer to A , redirect A and then adding it back:

```

443 B = ...
444 X = { A: { }      Void m(This1.X.A p1, This0.A p2)} <A=B> <+ {A: {}}
445 //should flattening redirect only p2 or also p1
446

```

```

447 X={ A:{ }      Void m(??? p1, This1.B p2) }

```

449 The complete L42 language solves those issues, but here we present a simplified version.

450 3.1 Sum

451 Rule (SUM) just delegate the work on the auxiliary notation defined below:

$$\begin{array}{l}
L_1 \ltimes L_2 = \text{interface? } \{Tz_1 \cup Tz_2; Mz \ltimes Mz', Mz_1, Mz_2; K?\} \\
L_1 = \text{interface? } \{Tz_1; Mz, Mz_1; K?_1\} \quad L_2 = \text{interface? } \{Tz_2; Mz', Mz_2; K?_2\} \\
\{empty, K?_1, K?_2\} = \{empty, K?\} \\
\text{if interface?} = \text{interface then } mdom(L_1) = mdom(L_2) \\
\hline
Tm(Txs)e? \ltimes Tm(Txs)e = Tm(Txs)e \\
Tm(Txs)e? \ltimes Tm(Txs) = Tm(Txs)e? \\
(C=L) \ltimes (C=L') = C = L \ltimes L,
\end{array}$$

453 As usual in definitions of sum operators, the implemented interfaces is the union of
454 the interfaces of L_1 and L_2 , the members with the same domain are recursively composed
455 while the members with disjoint domains are directly included. Since method and nested
456 class identifiers must be unique in a well formed L and $M_1 \ltimes M_2$ being defined only if the
457 identifier is the same, our definition forces $dom(Mz) = dom(Mz')$ and $dom(Mz_1)$ disjoint
458 $dom(Mz_2)$. For simplicity here we require at most one class to have a state; if both have
459 no state, the result will have no state, otherwise the result will have the only present state
460 (the set $\{empty, K?\}$ mathematically express this requirement in a compact way); we also
461 allow summing only interfaces with interfaces and final classes with final classes. When
462 two interfaces are composed both sides must define the same methods. This is because
463 other nested classes inside L_1 may be implementing such interface, and adding methods
464 to such interface would require those classes to somehow add an implementation for those
465 methods too. In literature there are expressive ways to soundly handle merging different
466 state, composing interfaces with final classes and adding methods to interfaces, but they are
467 out of scope in this work.

468 Member composition $M_1 \ltimes M_2$ uses the implementation from the right hand side, if
469 available, otherwise if the right hand side is abstract, the body is took from the left side.
470 Composing nested classes, note how they can not be **private**; it is possible to sum two literals
471 only if their private nested classes have different private names. This constraint can always
472 be obtained by alpha-renaming them: we assume a form of alpha-reameing for private nested
473 classes, that will consistently rename all the paths of form **This_n.C.Cs'**, where **This_n.C** refer
474 to such private nested class. The trivial definition of such alpha renaming is given in the
475 appendix.

476 3.2 Redirect

477 Rule (REDIRECT) is the centre of our interest for this work. As for sum we check that the
478 LV is in minimized form. Moreover, to have a single data structure p' where all the types
479 correctly points to the corresponding nested classes, we add the L to the top of our current
480 program. Notation R/id is defined as

$$481 \overline{Cs_0 = \text{This}_n.C.Cs = Cs_0 = \text{This}_{n+1}.C.Cs, \text{ where either } C \neq id \text{ or } n > 0}$$

482 In addition of adding 1 to all the types provided in the redirect map, since they was
483 relative to p and not p' , it also checks that R actually refers to types external of LV , by
484 preventing types of form **This₀.id.**

Notation $p.\text{redirectSet}(R)$ computes the set of nested classes that need to be redirected if R is redirected. This is information depend just from LV (the top of the program) and the domain of R . RedirectSet is easily computable.

$$\begin{array}{l} \text{dom}(R) \subseteq p.\text{redirectSet}(R) \\ \text{internals}(\text{exposedTypes}(p[\text{This}_0.Cs])) \subseteq p.\text{redirectSet}(R) \quad \text{with } Cs \in p.\text{redirectSet}(R) \\ \text{exposedTypes}(\text{interface? } \{Tz; Mz; K?\}) = Tz, \text{exposedTypes}(Mz) \\ \text{exposedTypes}(\text{static? } T_0 m(T_1 x_1 \dots T_n x_n) e?) = T_0 \dots T_n \\ \text{internals}(Tz) = \{Cs \mid \text{This}_0.Cs \in Tz\} \end{array}$$

The intuition behind redirectSet is that if the signature of a nested class mention another nested class, they must be redirected together. Consider the following simple example:

```
t={A={B size()} B={} ...}
Res=t<A=String>
```

If we were to redirect **A**, we would need to redirect also **B**: the type **B** is nested inside **t**, thus **String** would not be able to reach it. The only reasonable solution is to redirect **A** and **B** together.

For our redirection (and $p'.\text{bestRedirection}()$) to be well defined, we need to check that $p.\text{redirectable}(Cs)$ This is again a check local to the LV (the top of the program) and is also easily computable.

$$\begin{array}{l} \text{redirectable}(p, Cs) \text{ iff} \\ \text{empty} \notin Cs \\ \text{if } Cs \in Cs \text{ then } \text{This}_0.Cs \in \text{dom}(p) \\ \text{if } Cs \in Cs \text{ and } C \in \text{dom}(p(\text{This}_0.Cs)) \text{ then } Cs.C \in Cs \\ \text{if } Cs.C \in Cs \text{ then } p(\text{This}_0.Cs) = \text{interface? } \{ _ ; C=L _ ; _ \} \end{array}$$

That is, the empty path is not redirectable, every nested class of a redirect path must be redirected away, and all paths must traverse only non-private C .

Finally, $p.\text{bestRedirection}(R)$, given a p and an R (that are valid input for redirection as defined above) can denote the best complete map, mapping any element of Cs into a suitable type in p . This is the centerpiece of our formal framework and his definition will be the main topic of the next section.

Given the complete mapping R' , to produce the flattened result we first remove all the elements of Cs from LV , and then we apply R' as a rename, renaming all internal paths $Cs \in Cs$ to the corresponding external type $R'(Cs)$. Those two notations are formally defined as following:

$$\begin{array}{l} LV.\text{remove}(Cs_1 \dots Cs_n) = LV.\text{remove}(Cs_1) \dots \text{remove}(Cs_n) \\ LV[C.C = _].\text{remove}(Cs.C) = LV \text{ where } Cs.C \notin \text{dom}(LV) \\ R(L) = R_{\text{empty}}(L) \\ R_{Cs}(\text{interface? } \{Tz; Mz; K?\}) = \text{interface? } \{R_{Cs}(Tz); R_{Cs}(Mz); R_{Cs}(K?)\} \\ R_{Cs}(C=L) = C = R_{Cs.C}(L) \\ R_{Cs}(M), R_{Cs}(e), R_{Cs}(K) \quad \text{simply propagate on the structure until } T \text{ is reached} \\ R_{C_1 \dots C_n}(T) = \text{This}_{n+k+1}.Cs' \quad \text{where } T.\text{from}(\text{This}_0.C_1 \dots C_n) = \text{This}_0.Cs, R(Cs) = \text{This}_k.Cs' \\ \text{otherwise } R_{Cs}(T) = T \end{array}$$

Rename must keep track of the explored Cs in order to distinguish internal paths that need to be renamed, and the mapped type need to look out of the whole explored Cs and the top level code literal (thus $n + k + 1$).

4 BestRedirect

Best redirection balance three aspects:

■ Figure 1 Flattening

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

(TOP)

$$\frac{DVz \subseteq DVs \quad DVz \vdash \mathbf{Ok} \quad \frac{empty; DVz; id \vdash E \Rightarrow E'}{DVz \text{ id}=EDs \Rightarrow DVz \text{ id}=E'Ds}}{}$$

(L-ENTER)

$$\frac{p.\mathbf{push}(id=L[C=E]); C \vdash E \Rightarrow E'}{p; id \vdash L[C=E] \Rightarrow L[C=E']}$$

(TRAIT)

$$\frac{}{p; id \vdash t \Rightarrow p[t]}$$

(REDIRECT)

$$\begin{aligned} LV &= p.\mathbf{min}(id=LV) \\ p' &= p.\mathbf{push}(id=LV) \\ Csz &= p'.\mathbf{redirectSet}(R/id) \\ p'.\mathbf{redirectable}(Csz) \\ R' &= p'.\mathbf{bestRedirection}(R/id) \end{aligned}$$

(SUM)

$$\frac{LV_i = p.\mathbf{min}(id=LV_i) \quad LV_1 <+ LV_2 = LV}{p; id \vdash LV_1 <+ LV_2 \Rightarrow LV}$$

$$\frac{}{p; id \vdash LV < R > \Rightarrow R'(LV.\mathbf{remove}(Csz))}$$

- 518 ■ Validity: if the mapping is applied to well typed code (as in the rule (REDIRECT)) then
 519 the result is still well typed.
- 520 ■ Stability: changing little details on the code base (as for example adding a new unrelated
 521 nested class) do not change the selected map. This applies to both *LV* itself (internal
 522 stability) and the rest of the program (external stability).
- 523 ■ Specificity: when multiple options are available, the most specific is chosen.

524 To better divide the various aspect, we will use functions of form $(p, R) \rightarrow Rz$, producing
 525 valid mappings for any program p and starting map R . All of those functions will respect
 526 **possibleRedirections**. Rule REDIRECT ensures **possibleRedirections** for the input mapping,
 527 here we check that is also verified for the complete mapping.

$$\frac{\begin{aligned} &R' \in \mathbf{possibleRedirections}(p, R) \text{ if} \\ &R \subseteq R' \\ &dom(R') = \mathbf{redirectSet}(p, R) \\ &(p, R') \in \mathbf{validProblems} \end{aligned}}{(p, Cs_1 = T_1 \dots Cs_n = T_n) \in \mathbf{validProblems} \text{ iff } \forall i \in 1..n : \begin{aligned} &p.\mathbf{minimize}(T_i) = T_i \\ &T_i \text{ not of form } \mathbf{This}_0. _ \\ &p \vdash p[T] : \mathbf{OK} \\ &\mathbf{redirectable}(p, \mathbf{redirectSet}(p, R)) \end{aligned}}}$$

529 We now define **validRedirections** as one of such functions. This is the most complete
 530 function achieving both validity and internal stability. It is based on the judgement $p \vdash T \subseteq L$
 531 to be read as: under the program p , T is structurally a subtype of the literal L . Some more
 532 auxiliary notation is used: the obvious **isInterface** and the more interesting **superClasses** and
 533 method subtyping $p \vdash M \leq M'$. In **superClasses** we add T so that F-Bound polymorphism
 534 may work as expected, so that is possible to redirect **{implements Foo}** not only to any class
 535 implementing **Foo** but also to **Foo** itself. Method subtyping is given in the expressive form
 536 where the return type can be more specific, and the parameter types can be more general.

$$\begin{array}{l}
R' \in \text{validRedirections}(p, R) \text{ iff} \\
R' \in \text{possibleRedirections}(p, R) \\
\forall Cs \in \text{dom}(R') \quad p \vdash p[R'(Cs)] : R'(Cs) \subseteq R'(p[Cs]) : Cs \\
\hline
p \vdash P \subseteq \text{interface? } \{Tz; Mz; _ \} \text{ iff} \\
Tz \subseteq \text{superClasses}(p, P) \\
\forall m \in \text{dom}(Mz) : p \vdash p[P](m) \leq Mz(m) \\
\text{if } \text{interface?} = \text{interface} \text{ then } \forall m \in \text{dom}(p[P]) \quad p \vdash Mz(m) \leq p[P](m) \\
\text{if } \text{interface}(p[P]) \text{ then } \text{staticTm}(Tx) _ \notin Mz \text{ else } \text{interface?} = \text{empty} \\
\hline
\text{isInterface}(L) \text{ iff } L = \{ \text{interface } _ ; _ \} \\
\hline
\text{superClasses}(p, T) = \{T\} \cup \text{superClasses}(T_1) \cup \dots \cup \text{superClasses}(T_n) \\
\text{with } p[T] = \text{interface? } \{T_1 \dots T_n; _ ; _ \} \\
\hline
p \vdash \text{static? } T'_0 m(T_1 x_1 \dots T_n x_n) _ \leq \text{static? } T_0 m(T'_1 x'_1 \dots T'_n x'_n) _ \\
\text{with } T_0 \in \text{superClasses}(p, T'_0) \dots T_n \in \text{superClasses}(p, T'_n)
\end{array}$$

Note how **validRedirections**, while mathematically sound, is incredibly hard to compute: while it is easy to check if a certain $R' \in \text{validRedirections}(p, R)$, finding naively all such R' would require examining every possible permutation. In particular, subtyping allows for redirections to be conceptually took out of thin-air. Consider the following example:

```

542 I=interface {..}
543
544 A= {method A m(I x)}
545 C={implements I ..}
546 t={B: {} T: {method T m(B x)}}
547 Res=t<T=A>
548

```

Clearly, selecting **C** as a candidate to complete the map is a valid choice but is also an arbitrary choice that should not be made while automatically completing the mapping. What if type **D**={implements I ..} was introduced while maintaining the program? the completed redirect map may change unpredictably. As you can see from the former example, stability is an important requirement to allow for code maintainability. To model stability, we define the concept of similar programs:

$$DLs; DVz \quad DVz' \in \text{similarPrograms}(DLs; DVz)$$

Note how we just add new declarations at the outermost level. We will later prove that this is sufficient to ensure that adding/removing unrelated classes anywhere in the program would still not change the selected completed mapping. Finally, we have all formal tools to define **bestRedirection**, representing the high level specification of what correctly completing a mapping means.

$$\begin{array}{l}
\text{bestRedirection}(p, R) = \text{stableMostSpecific}(p, R, \text{validRedirections}) \\
\text{stableMostSpecific}(p, R, f) = R_0 \text{ iff } \forall p' \in \text{similarPrograms}(p) \\
R_0 \in f(p', R) \text{ and } \forall R_1 \in f(p', R) \quad \text{moreSpecific}(p, R_0, R_1) \\
\hline
\text{moreSpecific}(p, Cs_1 = T_1 \dots Cs_n = T_n, Cs_1 = T'_1 \dots Cs_n = T'_n) \\
T'_1 \in \text{superClasses}(p, T_1) \dots T'_n \in \text{superClasses}(p, T_n)
\end{array}$$

The best redirection is a **validRedirection** that is the most specific across all similar programs. While **bestRedirection** in the current form is not practically computable, it is clear from the formulation a good stepping stone to obtain a computable algorithm would be to replace **validRedirections** with an computable algorithm producing a subset of **validRedirections** and behaving identically for all the **similarPrograms**.

If multiple solutions are available, providing one of those non deterministically would clearly break stability. In this case, instead of just refusing to complete the mapping, we attempt to find the most specific solution: a solution where every individual mapping maps to the most specific type with respect to all the other available mappings. Choosing the

23:14 Using nested classes as associated types.

most specific solution is the desired solution in many practical cases; for example consider this variation of the former example, where **B** is the return type instead of an argument type:

```
573 I=interface { ..}  
574 C={implements I ..}  
575 A={C m()=..}  
576 t={B={ } T={method B m()} ..}  
577 Res=t<T=A>  
578
```

bestRedirection complete this mapping as **<T=A, B=C>** thanks to choosing the most specific, since also **B=I** is a valid option. In a language with a global supertype like **Any/Object**, that would be yet another option. Indeed, an alternative version selecting the least specific option may complete the mapping selecting **Any/Object** every time a nested was declared with empty body. That in turn is very common since it is the Java equivalent of not requiring any **extends T** constraints on a generic type.

5 Properties of **bestRedirection**

5.1 Internal/external stability

5.2 Meta-Level soundness

6 A computable **bestRedirection**: **choseRedirection**

7 Redirect applications

7.1 Graph example

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**.

```
595 graphUtils={  
596   Edges:list<+{Node start() Node end()}  
597   Node:{Edges connections()}  
598   Nodes:set<Elem=Node> //note that we do not specify equals/hash  
599   static Bool isConnected(Nodes nodes)=  
600     if(nodes.size()==0) then true  
601     else getConnected(nodes.asList().head()).size()==nodes.size()  
602   static Nodes getConnected(Node node)=getConnected(node,Nodes.empty())  
603   static Nodes getConnected(Node node,Nodes collected)=  
604     if(collected.contains(node)) then collected  
605     else connectEdges(node.connections(),collected.add(node))  
606   static Nodes connectEdges(Edges e,Nodes collected)=  
607     if( e.isEmpty()) then collected  
608     else connectEdges(e.tail(),collected.add(e.head().end()))  
609   }  
610 }  
611
```

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.

```
619 Map:{  
620   Street:{City start, City end, Int size}  
621
```

```

622 City: {}
623 Streets: priorityQueue<Elem=Street><+{
624     Int geq(Street e1, Street e2)=e1.size()-e2.size()}
625 }<+{
626     Streets: {}
627     City: {Streets connections, Int index} //index identify the node
628     Cities: set<Elem=City><+{
629         Bool eq(City e1, City e2) e1.index==e2.index
630         Int hash(City e) e.index
631     }
632     Cities cities
633     //more methods
634 }
635 MapUtils=graphUtils<Nodes=Map.Cities>
636 //infers Nodes.List, Node, Edges, Edge
637

```

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

640 7.2 Loading libraries

641 Most languages have a standard library. The standard library have two goals:

- 642 ■ Be a set of useful features for programmers to use.
- 643 ■ Be a starting point for third party libraries.

644 While the first point is quite obvious, the second one is a little surprising: third party libraries
 645 will communicate with the user code mostly by using standard library types: **Strings**,
 646 **Collections** and if we are in a pure OO language, also **Booleans**, **Integers**, **Doubles** and
 647 so on. The number of types involved in just take input and produce output is much larger
 648 then one could expect, since all kind of errors need to be considered too, and if the language
 649 support reflection, all those classes and their errors may end up being transitively required.
 650 The goal of the standard library is. For example, assume a simple library taking in input a
 651 **String** and producing a **String**: What are its dependencies?

652 **String** has an **isEmpty():Boolean** method, the **Boolean** has a **toString():String** method,
 653 a circular dependency. **String** has a **size():Integer** method, and a **getChar(Integer pos):Char**
 654 method. another typical method of strings is **@split(String regex):ListString@**, returning a
 655 list, that extends some general collection, and so on. If reflection is not implemented with
 656 Mirrors[], any of those object would have a **class():Class** method, and **Class** would have
 657 methods to connect with most other reflection classes.

658 Those dependencies are usually not a problem because we assume the standard library
 659 to be fixed and always available. Or, if you prefer, we are forced to design programming
 660 languages together with their standard library because those dependencies are too hard to
 661 manage directly.

662 With Redirect we can get free from this chain, and every third party library can just
 663 declare a the set of dependencies that are really needed. A single redirect application can
 664 then “load” the library in the current scope, where a variation of the standard library is
 665 available, but not necessarily exactly the library used to develop such third party library.

666 For example, a library may only need pass indexes around, without directly doing
 667 arithmetic, and may never use ..

```

668 library={//this code is fully self contained
669     N={
670     C={ Bool equal(C x)}
671     S={N size(), C getChar(N index)}
672     S myLibFunction(S x)=...
673     Map=interface{
674

```


23:16 Using nested classes as associated types.

```

675   S string()
676   C char()
677   N integer()
678 }
679 }
680 ...
681 Int={..}
682 Char={..}
683 String={..}
684 Map=interface{String string() ... }
685 LoadedLib=library<Map=Map>
686

```

Since the code of *t* is self contained (do not refer to any class in the outer program, it is possible to just ship it independently of the standard library of the target. The code of *library* can be typechecked once, and then any other program may load it as shown. Any program defining a **Map** interface with some types that we expect libraries to rely upon can be used in conjunction with *library*. In this example, if **Char** is not a valid structural subtype of **C**, the redirect would fail with a meaningful error message.

By the stability theorem, we get a good formal characterization of what are the acceptable shapes of the program so that the redirect would succeed. However, even if the program do not match the expectations of the library, it could be possible to tweak the code to make it work. ...

8 Related work

The operator redirect fist emerged with DeepFJig[], and was then explored more in MetaFJig[]; however, in this line of research, it was only able to redirect one nested class at a time, thus it was impossible to redirect mutually recursive classes.

We apply redirect over a trait composition language[sharly]. We are particularly inspired by Ferruccio, MetaFJig (and UseReuse). The main difference in the compilation model with respect to DeepFJig is an improved treatment of **This_n**: in MetaFJig **This_n** wrote in class **A** is not equivalent to **This_(n+1).A**. Moreover, the formalism is simpler and more compact.

Redirect supports a similar expressive power with respect to Abstract Types, and in some sense is proposed as an alternative to them. A good formal discussion about Abstract types can be found in [Foundations of Path-Dependent Types].

9 Appendix?

PUT LATER?However, the type system of the language is more restrictive when it comes to refine an interface method, allowing only return type refinement. This is not just to align our calculus with existing languages like Java/C# and C++, but is required to make reasoning about parameter types influential while expanding redirect mappings.END PUT LATER

$$\mathcal{E}_V ::= \square \mid \mathcal{E}_V \leftarrow E \mid LV \leftarrow \mathcal{E}_V \mid \mathcal{E}_V \leftarrow Cs = T > \quad \text{context of library-evaluation}$$

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

10 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

719 simply push each declaration onto a program and typecheck the resulting program. Rule *pok*
 720 typechecks a program by check the topmost class literal: we type check each of it's members
 721 (including all nested classes), check that it properly implements each interface it claims to,
 722 does something weird, and finanly check check that it's constructor only referenced existing
 723 types,

724

725

```
726 Define p |- Ok
```

```
727 =====
```

728

```
729 D1; Ds |- Ok ... Dn; Ds|- Ok
```

```
730 (Ds ok) ----- Ds = D1 ... Dn
```

```
731 Ds |- Ok
```

732

```
733 p |- M1 : Ok .... p |- Mn : Ok
```

```
734 p |- P1 : Implemented .... p |- Pn : Implemented
```

```
735 p |- implements(Pz; Ms) /*WTF?*/ if K? = K: p.exists(K.Txs.Ts)
```

```
736 (p ok) ----- p.top() = interface? {P1...Pn; M1, ..., Mn; K?}
```

```
737 p |- Ok
```

738

```
739 p.minimize(Pz) subseq p.minimize(p.top().Pz)
```

```
740 amt1 _ in p.top().Ms ... amtn _ in p.top().Ms
```

```
741 (P implemented) ----- p[P] = interface {Pz; amt1 ... am
```

```
742 p |- P : Implemented
```

743

```
744 (amt-ok) ----- p.exists(T, Txs.Ts)
```

```
745 p |- T m(Tcs) : Ok
```

746

```
747 p; This0 this, Txs |- e : T
```

```
748 (mt-ok) ----- p.exists(T, Txs.Ts)
```

```
749 p |- T m(Tcs) e : Ok
```

750

```
751 C = L, p |- Ok
```

```
752 (cd-Ok) -----
```

```
753 p |- C = L : OK
```

754

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

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

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

```
765 Define p; Txs |- e : T
```

23:18 Using nested classes as associated types.

```

766 =====
767 (var)
768 ----- T x in Txs
769 p; Txs |- x : T
770
771 (call)
772 p; Txs |- e0 : T0
773 ...
774 p; Txs |- en : Tn
775 ----- T' m(T1 x1 ... Tn xn) _ in p[T0].Ms
776 p; Txs |- e0.m(e1 ... en) : T'
777
778 (field)
779 p; Txs |- e : T
780 ----- p[T].K = constructor(_ T' x _)
781 p; Txs |- e.x : T'
782
783
784 (new)
785 p; Txs |- e1 : T1 ... p; Txs |- en : Tn
786 ----- p[T].K = constructor(T1 x1 ... Tn xn)
787 p; Txs |- new T(e1 ... en)
788
789
790 (sub)
791 p; Txs |- e : T
792 ----- T' in p[T].Pz
793 p; Txs |- e : T'
794
795
796 (equiv)
797 p; Txs |- e : T
798 ----- T =p T'
799 p; Txs |- e : T'

```

800 FROM and minimize that will go in the appendix:

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

803 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:

805 We will use members Mz as a function containing both method names m and class names C in its domain; thus we will assume notation $dom(Mz)$, $Mz(m)$, $Mz(C)$ with the usual meaning. Under here, we define useful auxiliary notations to access literals L with functional notation with the intent of accessing their members. We define notations $L[Cs = E] = L'$ and $Mz[C = E] = Mz'$ serving the role of function update. We use those notations to define $p(T) = LV$ accessing a program p as function. We also define operations on programs: 811 $p.\mathbf{push}(D) = p'$, allowing to work with programs as if they was stacks, and $p.\mathbf{min}(T) = T'$, 812 denoting the shortest type T' referring to the same nested class of T . We define $T.\mathbf{from}(T',j)$ 813 and $L.\mathbf{from}(T,j)$; we omit all the trivial propagation cases of form $M.\mathbf{from}(T,j)$, $K.\mathbf{from}(T,j)$

814 and $e.\text{from}(T,j)$. Finally, we combine those to notation for the most common task of
 815 getting the value of a literal, in a way that can be understood from the current location: $p[t]$
 816 and $p[T]$:

$\frac{}{(DLs; DVs).\text{push}(id=L) = id=L, DLs; DVs}$ $\frac{}{(_, C=L, _)(\text{This}_0.C.Cs) = L(Cs)}$ $\frac{}{p.\text{push}(_=L)(\text{This}_0.Cs) = L(Cs)}$ $\frac{}{p.\text{push}(_)(\text{This}_{n+1}.Cs) = p(\text{This}_n.Cs)}$	$\frac{}{(Mz, \text{private}?C=_)[C=E] = Mz, \text{private}?C=E}$ $\frac{}{LV(\emptyset) = LV}$ $\frac{}{L(C.Cs) = L(C)(Cs)}$ $\frac{}{L[\text{empty} = E] = E}$
$\frac{}{\text{members}(\text{interface? } \{ _, Mz; _ \}) = Mz}$ $\frac{}{L(m) = \text{members}(L)(m)}$ $\frac{}{L(C) = \text{members}(L)(C)}$ $\frac{}{\text{dom}(L) = \text{dom}(\text{members}(L))}$ $\frac{}{\text{mdom}(L) = \{m \in \text{dom}(L)\}}$	$\frac{}{\text{interface? } \{Tz; Mz; K?\}[C.Cs = E] = \text{interface? } \{Tz; Mz[C = Mz(C)][Cs = E]\}; K\}}$ $\frac{}{p.\text{min}(\text{This}_{n+1}.id_n.Cs) = p.\text{min}(\text{This}_n.Cs)}$ $\text{where } p = id_0=L_0 \dots id_n=L_n; Ds$ $\text{otherwise } p.\text{min}(T) = T$
$\text{This}_n.Cs.\text{from}(T,j) = \text{This}_n.Cs \quad \text{with } n < j$ $\text{This}_{n+j}.Cs.\text{from}(\text{This}_m.C_1 \dots C_k, j) = \text{This}_{m+j}.C_1 \dots C_{k-n} \quad \text{with } n \leq k$ $\text{This}_{n+j}.Cs.\text{from}(\text{This}_m.C_1 \dots C_k, j) = \text{This}_{m+j+n-k}.C_1 \dots C_{k-n}Cs \quad \text{with } n > k$	
$\frac{}{\{ \text{interface? } Tz; Mz; K \}.\text{from}(T,j-1) = \{ \text{interface? } Tz.\text{from}(T,j); Mz.\text{from}(T,j); K.\text{from}(T,j) \}}$ $\frac{}{(DL_1 \dots DL_n; _, t=LV)[t] = p.\text{min}(LV.\text{from}(\text{This}_n, 0))}$ $\frac{}{p[T] = p.\text{min}(\text{interface? } \{Tz.\text{from}(T,0); Mz.\text{from}(T,0); \}) \quad \text{where } p(T) = \text{interface? } \{Tz; Mz; K\}}$	

819 sdgsd

820

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

821

$$\text{interface? } \{ _, _, \text{private? } C=L_0, _, _ \}(C.Cs) := L_0(Cs)$$

822

where $L = a$

823

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

825

$$(DL_1 \dots DL_n; _, t=LV, _)[t] := LV.\text{from}(\text{This}_n)$$

826

$$p[T] := p.\text{minimize}(p(T).\text{from}(T))$$

827

828

829 – towel1:.. //Map: towel2:.. //Map: lib: T:towel1 f1 ... fn

830

MyProgram: T:towel2 Lib:lib[T=This0.T] ... –

831

11 extra

832

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

837

837 In literature, in addition to conventional Java style F-bound polymorphism, there is
 838 another way to obtain generics: to use associated types (to specify generic paramaters) and
 839 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(); interface 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 generics, 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 polymorphisms 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

EB: EA1 a1 EA1 a1

A1: A2: B: A1 a1 A2 a2 [B = EB] // A1 -> EI1, A2 -> EA2 a // A1 -> EA1, A2 -> EI2 b // A1 -> EA1, A2 -> EA2 c

a <= b b <= a c <= a, b a <= c

hi **Hi class**

a ::= b c

aa**hiHi class**qaq a ::= b c

a ::= b c

}}[()]
(TOP)

$a \xrightarrow{b} c \quad \forall i < 3a \vdash b : \text{OK}$

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