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.

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
```

Here class **SBox** is just a container of **String**s, and **myTrait** is code encoding **Box**es 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**.

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

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.

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

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

Note how in this case, class **Result.Box** would exists. 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 show 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 most other approaches, abstract/associated/generic types are special and have some restriction; 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 respect 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.

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. 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*.

$$e ::= x \mid e.m(es) \mid T.m(es) \mid e.x \mid \texttt{new } T(es) \qquad \text{expression} \qquad T ::= \texttt{This} n.Cs \qquad \text{types}$$

$$L ::= \{\texttt{interface } Tz;\ Ms\} \mid \{Tz;\ Mz\ ;\ K\} \qquad \text{code literal} \qquad Tx ::= T\ x \qquad \text{parameter}$$

$$M ::= \texttt{static}?\ T\ m(Txs)\ e? \mid \texttt{private}?\ C\texttt{=}E \qquad \text{member} \qquad D ::= id\texttt{=}E \qquad \text{declaration}$$

$$K ::= (Txz)? \qquad \text{state} \qquad id ::= C \mid t \qquad \text{class/trait id}$$

$$E ::= L \mid t \mid E_1 \texttt{<+} E_2 \mid E\texttt{<}R\texttt{>} \qquad \text{Code Expr.} \qquad v ::= \texttt{new } T(vs) \qquad \text{value}$$

$$R ::= Cs_1\texttt{=}T_1 \ldots Cs_n\texttt{=}T_n \qquad \text{redirect map} \qquad LV ::= \ldots$$

In the context of nested classes, types are paths. Syntactically, we represent them as relative paths of form $\texttt{This}_n.Cs$, where the number n identify the root of our path: $\texttt{This0}$ is the current class, $\texttt{This1}$ is the enclosing class, $\texttt{This2}$ is the enclosing enclosing class and so on. $\texttt{This}_n.Cs$ refers to the class obtained by navigating throughout Cs starting from $\texttt{This}_n$. Thus, $\texttt{This}_0$ 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 $\texttt{implements}$ in front of a non empty list of implemented interfaces and we will omit parenthesis around a non empty set of fields. To simplifiy 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 $\texttt{This}_n$ for any number n; in any method headers, all parameters have different names, and no parameter is named $\texttt{this}$.

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 $\texttt{This}_n.Cs.C.Cs'$, if C denotes a private nested class, then Cs is empty. We assume a form of alpha-reaming for private nested classes, that will consistently rename all the paths of form $\texttt{This}_n.C.Cs'$, where $\texttt{This}_n.C$ 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 $\texttt{this}$) and conventional (static) method calls. Field access and $\texttt{new}$ expressions are included but with restricted usage: well formed field accesses are of form $\texttt{this}.x$ in method bodies and $v.x$ in the main expression, while well formed $\texttt{new}$ expressions have to be of form $\texttt{new This0}(xs)$ in method bodies and of form v in the main expression. Those restrictions greatly simply 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 `<+` E_2 and our novel redirect E`<`Cs=T`>`.

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 LVs: 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[] 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:

```
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>
```

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

This would flatten into

```
194
195  list={/*as before*/
196  //IntList=list<Elem=Int>
197  IntList={
198    //Elem={} no more nested class Elem
199    static This0 empty()= new This0(Empty.of())
200    boolean isEmpty()= this.impl().isEmpty()
201    Int head()= this.impl.asCons().tail()
202    This0 tail()=this.impl.asCons().tail()
203    This0 cons(Int e)=new This0(Cons.of(e, this.impl)
204    private Impl={interface   Bool isEmpty()  Cons asCons()}
205    private Empty={/*as before*/}
206    private Cons={implements This1
207      Bool isEmpty()=false   Cons asCons()=this
208      Int elem Impl tail }
209    Impl impl
210  }//everywhere there was "Elem", now there is "Int"
211
```

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

```
214
215  class ShapeGroup<T extends Shape>{
216    List<T> shapes;
217    ..}
218  //alternative implementation
219  class ShapeGroup<T extends Shape,L extends List<T>>{
220    L shapes;
221    ..}
222
```

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

```
225
226  shapeGroup={
227    Shape={implements Shape}
228    List=list<Elem=Shape>
229    List shapes
230    ..}
231
```

With redirect, `shapeGroup` follow both roles of the two Java examples; indeed there are two reasonable ways to reuse this code
   Triangolation=shapeGroup<**Shape=Triangle**>, if we have a **Triangle** class and we would like the concrete list type used inside to be local to the **Triangolation**, or **Triangolation**=shapeGroup<**List=Triangles**>, if we have a preferred implementation for the list of triangles that is going to be used by our **Triangolation**. Those two versions would flatten as follow:

```
238
239  //Triangolation=shapeGroup<Shape=Triangle>
240  Triangolation={
241    List=/*list with Triangle instead of Elem*/
242    List shapes
243    ..}
244
245  //Triangolation=shapeGroup<List=Triangles>
246  //exapands to shapeGroup<List=Triangles,Shape=Triangle>
247  Triangolation={
248    Triangles shapes
249    ..}
250
```

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

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

2.1.2   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:

```
set:{
  Elem:{}
  List=list<Elem=Elem>
  static This0 empty()= new This0(List.empty())
  Bool contains(Elem e)=../*uses eq and hash*/
  Int size()=..
  This add(Elem e)=...
  This remove(Elem e)=...
  Bool eq(Elem e1,Elem e2)//abstract
  Int hash(Elem e)//abstract
  List asList //to allow iteration
  }
eqElem={
  Elem={ Bool equals(Elem e)/*abstract*/}
  Bool eq(Elem e1,Elem e2)=e1.equals(e2)
  }
hashElem={
  Elem={ Int hash(Elem e)/*abstract*/}
  Int hash(Elem e)=e.hash()
  }
Strings=(set<+eqElem<+eqHash)<Elem=String>
LongStrings=(set<+eqElem)<Elem=String> <+{
  Int hash(String e)=e.size()
  }//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<Ele` 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 is 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 $D_0 \ldots D_n$ 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

```
A={()}
t={ B={()}    This1.A m(This0.B b)}
```

```
355  C={D={E=t}}
356  H=t<B=A>
357
```

the flattened version for `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

```
361
362  A={()}
363  t={ B={()}     This1.A m(This0.B b)}
364  C={D={E=t}};
365  C={D={E=t}},//this means, we entered in C
366  D={E=t}//this means, we entered in D
367
```

In order to fetch code literals form 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 consitently 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 contex 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
propegates 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 that they are 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:

```
391
392  B=...
393  X={ A:{}     Void m(This1.X.A p1, This0.A p2)} <A=B> <+ {A:{}}
394  //should flattening redirect only p2
395  X={ A:{}     Void m(This1.X.A p1, This1.B p2)}
396  //or both occurrences?
397  X={ A:{}     Void m(This1.B p1, This1.B p2)}
398
```

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

3.1   Sum

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

Def: $L_1 \text{<+} L_2 = \texttt{interface}? \{Tz_1 \cup Tz_2;\ Mz \text{<+} Mz', Mz_1, Mz_2;\ K?\}$

$\quad L_1 = \texttt{interface}? \{Tz_1;\ Mz, Mz_1;\ K?_1\}, \qquad L_2 = \texttt{interface}? \{Tz_2;\ Mz', Mz_2;\ K?_2\}$

$\quad \{empty, K?_1, K?_2\} = \{empty, K?\}$

$\quad$ if $\texttt{interface}? = \texttt{interface}$ then $mdom(L_1) = mdom(L_2)$

Def: $Tm(Txs)e? \text{<+} Tm(Txs)e = Tm(Txs)e$

Def: $Tm(Txs)e? \text{<+} Tm(Txs) = Tm(Txs)e?$

Def: $(C\text{=}L) \text{<+} (C\text{=}L') = C\text{=}\ L \text{<+} L,$

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 recursivelly 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 $M_1 \text{<+} M_2$ 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 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 $M_1 \text{<+} M_2$ uses the implementation from the right hand side, if available, otherwise if the right hand side is abstract, the body is took from the left side. Composing nested classes, not how they can not be $\texttt{private}$; 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.

3.2 Redirect

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

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

In addition of adding 1 to all the types provided in the redirect map, since they was relative to p and not p', it also checks that R actually refers to types external of LV, by preventing types of form $\texttt{This}_0.id.\_\_$.

Notation $p.\textbf{redirectSet}(R)$ 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 easly computable.

DO WE NEED TO ADD NESTEDS HERE?

$dom(R) \subseteq p.\textbf{redirectSet}(R)$

$\textbf{internals}(\textbf{reachables}(p[\texttt{This}_0.Cs])) \subseteq p.\textbf{redirectSet}(R) \quad$ with $Cs \in p.\textbf{redirectSet}(R)$

$\textbf{reachables}(\texttt{interface}? \{Tz;\ Mz;\ K?\}) = Tz, \textbf{reachables}(Mz)$

$\textbf{reachables}(\texttt{static}?T_0m(T_1x_1 \ldots T_nx_n)e?) = T_0 \ldots T_n$

$\textbf{internals}(Tz) = \{Cs \mid \texttt{This}_0.Cs \in Tz\}$

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

■ **Figure 1** Flattening

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

(TOP)
$$\frac{\begin{array}{c} DVz \subseteq DVs \\ DVz \vdash \mathbf{Ok} \\ empty; DVz; id \vdash E \Rightarrow E' \end{array}}{DVs\ id\text{=}EDs \Rightarrow DVs\ id\text{=}E'Ds}$$

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

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

(SUM)
$$\frac{\begin{array}{c} LV_i = p._{\mathbf{min}(id\text{=}LV_i)} \\ LV_1\text{<+}LV_2 = LV \end{array}}{p; id \vdash LV_1\text{<+}LV_2 \Rightarrow LV}$$

(REDIRECT)
$$\frac{\begin{array}{c} LV = p._{\mathbf{min}(id\text{=}LV)} \\ p' = p._{\mathbf{push}(id\text{=}LV)} \\ Csz = p'._{\mathbf{redirectSet}(R/id)} \\ p'._{\mathbf{redirectable}(Csz)} \\ R' = p'._{\mathbf{bestRedirection}(R/id)} \end{array}}{p; id \vdash LV\text{< }R\text{>} \Rightarrow R'(L._{\mathbf{remove}(Csz)})}$$

```
439
440  t={A={B size()} B={} ...}
441  Res=t<A=String>
442
```

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

446 For $p'._{\mathbf{bestRedirection}()}$ to be well defined, we need to check that $p._{\mathbf{redirectable}(Csz)}$ This
447 is again a check local to the LV (the top of the program) and is also easly computable.

redirectable(p, Csz)iff

$empty \notin Csz$

448 if $Cs \in Csz$ then $\mathtt{This}_0.Cs \in dom(p)$

if $Cs \in Csz$ and $C \in dom(p(\mathtt{This}_0.Cs))$ then $Cs.C \in Csz$

if $Cs.C.\_\_ \in Csz$ then $p(\mathtt{This}_0.Cs) = \mathtt{interface}? \{\_\_;\ C\text{=}L\_\_;\ \_\}$

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

451 We use $p'._{\mathbf{bestRedirection}()}$ in order to complete the provided mapping R into a complete
452 mapping. Such mapping must be of the correct ..... Something does not work: either we
453 check dom(R')==Csz or we use dom(R') instead of Csz. What is the right one?

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

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

459 ## 4   Appendix?

460
$\mathcal{E}_V ::= \Box \,|\, \mathcal{E}_V \text{ <+ } E \,|\, LV \text{ <+ } \mathcal{E}_V \,|\, \mathcal{E}_V \text{< } Cs \text{=} T \text{>}$ context of library-evaluation

$\mathcal{E}_v ::= \Box \,|\, \mathcal{E}_v.\, m\text{(es)} \,|\, v.\, m\text{(vs } \mathcal{E}_v \text{ es)} \,|\, T.\, m\text{(vs } \mathcal{E}_v \text{ es)}$

5 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 convential, it involves typing judgments of the form $p; Txs \vdash e : T$, with the usual program p and variable environement 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 finanly check check that it's constructor only referenced existing types,

```
Define p |- Ok
===============================================================

D1; Ds |- Ok ... Dn; Ds|- Ok
(Ds ok) ----------------------------- Ds = D1 ... Dn
Ds |- Ok


p |- M1 : Ok .... p |- Mn : Ok
p |- P1 : Implemented .... p |- Pn : Implemented
p |- implements(Pz; Ms) /*WTF?*/                  if K? = K: p.exists(K.Txs.Ts)
(p ok) ------------------------------------------ p.top() = interface? {P1...Pn; M1, ..., Mn; K?
p |- Ok


p.minimize(Pz) subseteq p.minimize(p.top().Pz)
amt1 _ in p.top().Ms ... amtn _ in p.top().Ms
(P implemented) ---------------------------------------------- p[P] = interface {Pz; amt1 ... am
p |- P : Implemented


(amt-ok) ------------------- p.exists(T, Txs.Ts)
p |- T m(Tcs) : Ok


p; ThisO this, Txs |- e : T
(mt-ok) ---------------------------- p.exists(T, Txs.Ts)
p |- T m(Tcs) e : Ok


C = L, p |- Ok
(cd-Ok) -------------------
p |- C = L : OK

```

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 mensioned in the method signature exist, and ofcourse for the latter case, that the body respects this signature.

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

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

```
Define p; Txs |- e : T
=====================================
(var)
--------------------- T x in Txs
p;  Txs |- x : T

(call)
p; Txs |- e0 : T0
...
p; Txs |- en : Tn
---------------------------------- T' m(T1 x1 ... Tn xn) _ in p[T0].Ms
p; Txs |- e0.m(e1 ... en) : T'

(field)
p; Txs |- e : T
-------------------------------------- p[T].K = constructor(_ T' x _)
p; Txs |- e.x : T'


(new)
p; Txs |- e1 : T1 ... p; Txs |- en : Tn
------------------------------------------ p[T].K = constructor(T1 x1 ... Tn xn)
p; Txs |- new T(e1 ... en)


(sub)
p; Txs |- e : T
---------------------------------- T' in p[T].Pz
p; Txs |- e : T'


(equiv)
p; Txs |- e : T
---------------------------------- T =p T'
p; Txs |- e : T'
```

6   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 **Edge**s. A `isConnected` function takes a list of **Node**s. A `getConnected` function takes **Node** and return a set of **Node**s.

```
graphUtils ={
  Edges:list <+{Node start() Node end ()}
```

```
554   Node:{Edges connections()}
555   Nodes:set<Elem=Node>//note that we do not specify equals/hash
556   static Bool isConnected(Nodes nodes)=
557     if(nodes.size()=0) then true
558     else getConnected(nodes.asList().head()).size()==nodes.size()
559   static Nodes getConnected(Node node)=getConnected(node,Nodes.empty())
560   static Nodes getConnected(Node node,Nodes collected)=
561     if(collected.contains(node)) then collected
562     else connectEdges(node.connections(),collected.add(node))
563   static Nodes connectEdges(Edges e,Nodes collected)=
564     if( e.isEmpty()) then collected
565     else connectEdges(e.tail(),collected.add(e.head().end()))
566   }
567
```

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 easly 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.

```
576   Map:{
577     Street:{City start,City end, Int size}
578     City:{}
579     Streets:priorityQueue<Elem=Street><+{
580       Int geq(Street e1,Street e2)=e1.size()-e2.size()}
581     }<+{
582     Streets:{}
583     City:{Streets connections, Int index}//index identify the node
584     Cities:set<Elem=City><+{
585       Bool eq(City e1,City e2) e1.index==e2.index
586       Int hash(City e) e.index
587       }
588     Cities cities
589     //more methods
590     }
591   MapUtils=graphUtils<Nodes=Map.Cities>
592   //infers Nodes.List, Node, Edges, Edge
593
```

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:

$$(DLs; DVs)_{\textbf{.push}(id\text{=}L)} \coloneqq id\text{=}L, DLs; DVs$$

$$(;\_, C\text{=}L, \_)(\texttt{This}_0.C.Cs) \coloneqq L(Cs)$$

$$p_{\textbf{.push}(\_\text{=}L)}(\texttt{This}_0.Cs) \coloneqq L(Cs)$$

$$p_{\textbf{.push}(\_)}(\texttt{This}_{n+1}.Cs) \coloneqq p(\texttt{This}_n.Cs)$$

$$LV(\emptyset) \coloneqq LV$$

$$\texttt{interface}? \ \{\_; \ \_, C\text{=}L_0, \_; \ \_\}(C.Cs) \coloneqq L_0(Cs)$$

where $L = a$

This notation just fetch the referred LV without any modification. To adapt the paths we define $T_{0.\textbf{from}(T_1,j)}$, $L_{.\textbf{from}(T,j)}$ and $p_{.\textbf{minimize}(T)}$ as following:

$$\text{This}_n.Cs_{\mathbf{.from}(T,j)} := \text{This}_n.Cs \quad \text{with } n < j$$

$$\text{This}_{n+j}.Cs_{\mathbf{.from}(\text{This}_m.C_1...C_k,j)} := \text{This}_{m+j}.C_1...C_{k-n} \quad \text{with } n \leq k$$

$$\text{This}_{n+j}.Cs_{\mathbf{.from}(\text{This}_m.C_1...C_k,j)} := \text{This}_{m+j+n-k}.C_1...C_{k-n}Cs \quad \text{with } n > k$$

$$\{\text{interface}?Tz;\ Mz;\ K\}_{\mathbf{from}(T,j-1)} := \{\text{interface}?Tz_{\mathbf{.from}(T,j)};\ Mz_{\mathbf{.from}(T,j)};\ K_{\mathbf{.from}(T,j)}\}$$

$$p_{\mathbf{.minimize}(T)} := T'....$$

Finally, we we combine those to notation for the most common task of getting the value of a literal, in a way that can be understand from the current location: $p[t]$ and $p[T]$:

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

$$p[T] := p_{\mathbf{.minimize}(p(T)_{\mathbf{.from}(T)})}$$

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

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

7   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 inheritence (to instantiate the paramaters). However, when parametrizing multiple types, the user to specify the full mapping. For example in Java interface A<B> 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 fundational model for genericty 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

647   EB: EA1 a1 EA1 a1

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

649   EI2 b // A1 -> EA1, A2 -> EA2 c

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

651   `hi` **`Hi`** `class`

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

652   aa`hi`**`Hi`**`class`qaq $a ::= b \quad c$

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

653   `}}][()]`

$$(\text{TOP})$$

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

654

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