Using nested classes as associated types.

Authors omitted for double-bind review.
Unspecified Institution.

Abstract

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

2012 ACM Subject Classification Dummy classification

Keywords and phrases Dummy keyword

Digital Object Identifier 10.4230/LIPIcs.CVIT.2016.23

1 Introduction

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

The core of our design is to reuse the concept of nested classes instead of relying of a new kind of member for associated types. An operation, 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));}
  }
```


42nd Conference on Very Important Topics (CVIT 2016).
Editors: John Q. Open and Joan R. Access; Article No. 23; pp. 23:1–23:19
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

```
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.

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

```
Result2={
  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 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 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.

Before diving in the formal details, we show an example motivating that expanding the redirect map is not trivial when subtyping is took in 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**:

```
Point=interface{ ...}
ColorPoint=interface{ implements Point ...}
Root=interface{Point m()}
Left={interface implements EA Point m()}
Right:{interface implements EA ColorPoint m()}
Merge={implements Left, Right    ColorPoint m()}
C={ Merge bind()}
```

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 method `show` reporting in its signature **Target**, **Result** and **I**, so we can see where are they redirected to.

```
t={
  I=interface{ColorPoint m()}
  Result=interface{implements I    ColorPoint m()}
  Target={Result bind()}
  Target show(Result r, I i)
  }
Res=t<Target=C>
```

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 was 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 tree 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 took 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*.

$e ::= x \mid e.m(es) \mid T.m(es) \mid e.x \mid \texttt{new}\ T(es)$	expression	$T ::= \texttt{This}n.Cs$	types
$L ::= \texttt{\{ interface}\ Tz;\ Ms\texttt{\}} \mid \texttt{\{}Tz;\ Mz\ ;\ K\texttt{\}}$	code literal	$Tx ::= T\ x$	parameter
$M ::= \texttt{static}?\ T\ m(Txs)\ e? \mid \texttt{private}?\ C\texttt{=}E$	member	$D ::= id\texttt{=}E$	declaration
$K ::= (Txz)?$	state	$id ::= C \mid t$	class/trait id
$E ::= L \mid t \mid E_1 \texttt{<+}\ E_2 \mid E\texttt{<}R\texttt{>}$	Code Expr.	$v ::= \texttt{new}\ T(vs)$	value
$R ::= Cs_1 \texttt{=} T_1 \ldots Cs_n \texttt{=} T_n$	redirect map	$LV ::= \ldots$	

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: $\texttt{This}/\texttt{This}_0$ is the current class, $\texttt{This}_1$ is the enclosing class, $\texttt{This}_2$ is the enclosing enclosing class and so on. $\texttt{This}_n.Cs$ refers to the class obtained by navigating throughout Cs starting from $\texttt{This}_n$. 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 $\texttt{This}_n$ 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 $\texttt{This}_n.Cs.C.Cs'$, 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 $\texttt{this}.x$ in method bodies and $v.x$ in the main expression, while well formed `new` expressions have to be of form `new This0(`xs`)` in method bodies and of form v in the main expression. Those restrictions greatly simply 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 $\texttt{new}\ T(vs)$.

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 <+ E_2 and our novel redirect$E$<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 LVs: 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 flatted away. Thus, execution is very simple and standard and behaves like a variation of FJ[] with interfaces instead of inheritance, and where nested classes are just a way to hierarchically organize code names. On the other side, code composition in this setting is very interesting and powerful, where nested classes are much more than name organization: they support in a simple and intuitive way expressive code reuse patterns. To flatten an E we need to understand the behaviour of the two operators, and how to load the code of a trait: since it was written in another place, the syntactic representation of the types need to be updated. For each of those points we will first provide some informal explanation and then we will proceed formalizing the precise behaviour.

2.1.1 Redirect

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

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

This would flatten into

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

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

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

```
class ShapeGroup<T extends Shape>{
  List<T> shapes;
  ..}
//alternative implementation
class ShapeGroup<T extends Shape,L extends List<T>>{
  L shapes;
  ..}
```

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

```
shapeGroup={
  MyShape={implements Shape}
  List=list<Elem=MyShape>
  List shapes
  ..}
```

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

`Triangolation=shapeGroup<MyShape=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:

```
//Triangolation=shapeGroup<MyShape=Triangle>
Triangolation={
  List=/*list with Triangle instead of Elem*/
  List shapes
  ..}

//Triangolation=shapeGroup<List=Triangles>
//expands to shapeGroup<List=Triangles,MyShape=Triangle>
Triangolation={
  Triangles shapes
  ..}
```

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

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.

2.1.2 Preferential sum; 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 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:

```
listComp=list<+{
  Elem:{ Int geq(This 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*/
  }
```

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>
```

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

```
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<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 $DV ::= id\texttt{=}LV$; 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)}
C={D={E=t}}
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

```
A={()}
t={ B={()}   This1.A m(This0.B b)}
C={D={E=t}};
C={D={E=t}},//this means, we entered in C
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 were they are useful for direct comparison with T. for example, if the program contains `{B={} This0 m(This0.B x)}` in position `This2.A`, $p[$`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: $p_{\textbf{.push}(D)} = p'$, allowing to work with programs as if they was stacks, and $p_{\textbf{.min}(T)} = T'$, denoting the shortest type T' referring to the same nested class of T. We define $T_{\textbf{.from}(T',j)}$ and $L_{\textbf{.from}(T,j)}$; we omit all the trivial propagation cases of form $M_{\textbf{.from}(T,j)}$, $K_{\textbf{.from}(T,j)}$ and $e_{\textbf{.from}(T,j)}$.

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

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

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

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

$$\textbf{members}(\texttt{interface}?\ \{\_;\ Mz;\ \_\}) = Mz$$

$$L(m) = \textbf{members}(L)(m)$$

$$L(C) = \textbf{members}(L)(C)$$

$$dom(L) = dom(\textbf{members}(L))$$

$$mdom(L) = \{m \in dom(L)\}$$

$$(Mz, \texttt{private}?C\texttt{=}\_)[C = E] = Mz, \texttt{private}?C\texttt{=}E$$

$$LV(\emptyset) = LV$$

$$L(C.Cs) = L(C)(Cs)$$

$$L[empty = E] = E$$

$$\texttt{interface}?\ \{Tz;\ Mz;\ K?\}[C.Cs = E] = \texttt{interface}?\ \{Tz;\ Mz[C = Mz(C)[Cs = E]];\ K?\}$$

$$p_{\textbf{.min}(\texttt{This}_{n+1}.id_n.Cs)} = p_{\textbf{.min}(\texttt{This}_n.Cs)} \quad \text{where } p = id_0\texttt{=}L_0 \ldots id_n\texttt{=}L_n\_; Ds$$

$$\text{otherwise } p_{\textbf{.min}(T)} = T$$

$$\texttt{This}_n.Cs_{\textbf{.from}(T,j)} = \texttt{This}_n.Cs \quad \textit{with } n < j$$

$$\texttt{This}_{n+j}.Cs_{\textbf{.from}(\texttt{This}_m.C_1\ldots C_k,j)} = \texttt{This}_{m+j}.C_1 \ldots C_{k-n} \quad \textit{with } n \leq k$$

$$\texttt{This}_{n+j}.Cs_{\textbf{.from}(\texttt{This}_m.C_1\ldots C_k,j)} = \texttt{This}_{m+j+n-k}.C_1 \ldots C_{k-n}Cs \quad \textit{with } n > k$$

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

$$(DL_1 \ldots DL_n; \_, t\texttt{=}LV)[t] = p_{\textbf{.min}(LV_{\textbf{.from}(\texttt{This}_n,0)})}$$

$$p[T] = p_{\textbf{.min}(\texttt{interface}?\ \{Tz_{\textbf{.from}(T,0)};\ Mz_{\textbf{.from}(T,0)};\ \})} \quad \text{where } p(T) = \texttt{interface}?\ \{Tz;\ Mz;\ K?\}$$

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 $\texttt{This}_0$, and we use to the fact that refers to the same nested class as $\texttt{This}_1.id$. Rule (Top) selects the leftmost $id\texttt{=}E$ 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 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:

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

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

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:

L_1<+L_2 $=$ interface? $\{Tz_1 \cup Tz_2;\ Mz$<+$Mz', Mz_1, Mz_2;\ K?\}$
  $L_1 =$ interface? $\{Tz_1;\ Mz, Mz_1;\ K?_1\}$ $L_2 =$ interface? $\{Tz_2;\ Mz', Mz_2;\ K?_2\}$
  $\{empty, K?_1, K?_2\} = \{empty, K?\}$
  if interface? $=$ interface then $mdom(L_1) = mdom(L_2)$

$Tm(Txs)e?$<+$Tm(Txs)e = Tm(Txs)e$
$Tm(Txs)e?$<+$Tm(Txs) = Tm(Txs)e?$
$(C$=$L)$<+$(C$=$L') = C$= L<+$L,$

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<+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<+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, note how they can not be private; it is possible to sum two literals only if their private nested classes have different private names. This constraint can always be obtained by alpha-renaming them: 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 renaming is given in the appendix.

3.2 Redirect

Rule (REDIRECT) is the centre of our interest for this work. As for sum we check that the LV is in minimized 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 =$\texttt{This}_n.C.Cs = Cs_0$ =$\texttt{This}_{n+1}.C.Cs$, 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.

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

$$\textbf{exposedTypes}(\texttt{interface}?\ \{Tz;\ Mz;\ K?\}) = Tz, \textbf{exposedTypes}(Mz)$$
$$\textbf{exposedTypes}(\texttt{static}?T_0 m(T_1 x_1 \ldots T_n x_n) e?) = T_0 \ldots T_n$$

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

The intuition behind **redirectSet** is that if the signature of a nested class mention another nested class, they must be 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 $p'_{\textbf{.bestRedirection}()}$) to be well defined, we need to check that $p_{\textbf{.redirectable}(Csz)}$ This is again a check local to the LV (the top of the program) and is also easily computable.

$$\textbf{redirectable}(p, Csz)\text{iff}$$
$$\quad empty \notin Csz$$
$$\quad \text{if } Cs \in Csz \text{ then } \texttt{This}_0.Cs \in dom(p)$$
$$\quad \text{if } Cs \in Csz \text{ and } C \in dom(p(\texttt{This}_0.Cs)) \text{ then } Cs.C \in Csz$$
$$\quad \text{if } Cs.C.\_ \in Csz \text{ then } p(\texttt{This}_0.Cs) = \texttt{interface}?\ \{\_;\ C = L\_;\ \_\}$$

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, $p_{\textbf{.bestRedirection}(R)}$, 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 Csz 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 Csz from LV, and then we apply R' as a rename, renaming all internal paths $Cs \in Csz$ to the corresponding external type $R'(Cs)$. Those two notations are formally defined as following:

$$LV_{\textbf{.remove}(Cs_1 \ldots Cs_n)} = LV_{\textbf{.remove}(Cs_1)} \cdots {}_{\textbf{.remove}(Cs_n)}$$
$$LV[Cs.C = \_]_{\textbf{.remove}(Cs.C)} = LV \text{where } Cs.C \notin dom(LV)$$

$$R(L) = R_{empty}(L)$$
$$R_{Cs}(\texttt{interface}?\ \{Tz;\ Mz;\ K?\}) = \texttt{interface}?\ \{R_{Cs}(Tz);\ R_{Cs}(Mz);\ R_{Cs}(K?)\}$$
$$R_{Cs}(C = L) = C = R_{Cs.C}(L)$$
$$R_{Cs}(M), R_{Cs}(e), R_{Cs}(K) \quad \text{simply propagate on the structure until } T \text{ is reached}$$
$$R_{C_1 \ldots C_n}(T) = \texttt{This}_{n+k+1}.Cs' \quad \text{where } T_{\textbf{.from}(\texttt{This}_0.C_1 \ldots C_n)} = \texttt{This}_0.Cs,\ R(Cs) = \texttt{This}_k.Cs'$$
$$\text{otherwise } R_{Cs}(T) = T$$

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

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

$$\textsc{(Top)}\ \frac{\begin{array}{c} DVz \subseteq DVs \\ DVz \vdash \mathbf{Ok} \\ empty; DVz; id \vdash E \Rightarrow E' \end{array}}{DVs\ id\texttt{=}E Ds \Rightarrow DVs\ id\texttt{=}E'\, Ds} \qquad \textsc{(L-enter)}\ \frac{p_{.\mathbf{push}(id\texttt{=}L[C=E])}; C \vdash E \Rightarrow E'}{p; id \vdash L[C = E] \Rightarrow L[C = E']} \qquad \textsc{(trait)}\ \frac{}{p; id \vdash t \Rightarrow p[t]}$$

$$\textsc{(sum)}\ \frac{\begin{array}{c} LV_i = p_{.\mathbf{min}(id\texttt{=}LV_i)} \\ LV_1\texttt{<+}LV_2 = LV \end{array}}{p; id \vdash LV_1\texttt{<+}LV_2 \Rightarrow LV} \qquad \textsc{(redirect)}\ \frac{\begin{array}{c} LV = p_{.\mathbf{min}(id\texttt{=}LV)} \\ p' = p_{.\mathbf{push}(id\texttt{=}LV)} \\ Csz = p'_{.\mathbf{redirectSet}(R/id)} \\ p'_{.\mathbf{redirectable}(Csz)} \\ R' = p'_{.\mathbf{bestRedirection}(R/id)} \end{array}}{p; id \vdash LV\texttt{<}R\texttt{>} \Rightarrow R'(LV_{.\mathbf{remove}(Csz)})}$$

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

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

$R' \in \mathbf{possibleRedirections}(p, R)$ if
$\quad R \subseteq R'$
$\quad dom(R') = \mathbf{redirectSet}(p, R)$
$\quad (p, R') \in \mathbf{validProblems}$

$(p, Cs_1\texttt{=}T_1 \ldots Cs_n\texttt{=}T_n) \in \mathbf{validProblems}$ iff $\forall i \in 1..n:$
$\quad p_{.\mathbf{minimize}(T_i)} = T_i$
$\quad T_i$not of form $\texttt{This}_0\texttt{.\_}$
$\quad p \vdash p[T] : \mathbf{OK}$
$\quad \mathbf{redirectable}(p, \mathbf{redirectSet}(p, R))$

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

$R' \in \textbf{validRedirections}(p, R)$ iff
  $R' \in \textbf{possibleRedirections}(p, R)$
  $\forall Cs \in dom(R')\ \ p \vdash p[R'(Cs)] : R'(Cs) \subseteq R'(p[Cs]) : Cs$

$p \vdash P \subseteq \texttt{interface}?\ \{Tz;\ Mz;\ _\}$iff
  $Tz \subseteq \textbf{superClasses}(p, P)$
  $\forall m \in dom(Mz) :\ \ p \vdash p[P](m) \leq Mz(m)$
  if $\texttt{interface}? = \texttt{interface}$ then $\forall m \in dom(p[P])\ \ p \vdash Mz(m) \leq p[P](m)$
  if $\textbf{interface}(p[P])$ then $\texttt{static}T m(Txs)\_ \notin Mz$ else $\texttt{interface}? = empty$

$\textbf{isInterface}(L)$iff $L = \{\texttt{interface}\ \_;\_\}$

$\textbf{superClasses}(p, T) = \{T\} \cup \textbf{superClasses}(T_1) \cup \ldots \cup \textbf{superClasses}(T_n)$
  with $p[T] = \texttt{interface}?\ \{T_1 \ldots T_n;\ \_;\ \_\}$

$p \vdash \texttt{static}?\, T'_0\, m(T_1 x_1 \ldots T_n x_n)\_ \leq \texttt{static}?\, T_0\, m(T'_1 x'_1 \ldots T'_n x'_n)\_$
  with $T_0 \in \textbf{superClasses}(p, T'_0) \ldots T_n \in \textbf{superClasses}(p, T'_n)$

Note how **validRedirections**, while mathematically sound, is incredibly hard to compute: while it is easy to check if a certain $R' \in \textbf{validRedirections}(p, R)$, 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:

```
I=interface {..}
A= {method A m(I x)}
C={implements I ..}
t={B: {} T: {method T m(B x)}}
Res=t<T=A>
```

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:

$DLs; DVz\, DVz' \in \textbf{similarPrograms}(DLs; DVz)$

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.

$\textbf{bestRedirection}(p, R) = \textbf{stableMostSpecific}(p, R, \textbf{validRedirections})$

$\textbf{stableMostSpecific}(p, R, f) = R_0$iff $\forall p' \in \textbf{similarPrograms}(p)$
  $R_0 \in f(p', R)$ and $\forall R_1 \in f(p', R)\ \ \textbf{moreSpecific}(p, R_0, R_1)$

$\textbf{moreSpecific}(p, Cs_1 = T_1 \ldots Cs_n = T_n, Cs_1 = T'_1 \ldots Cs_n = T'_n)$
  $T'_1 \in \textbf{superClasses}(p, T_1) \ldots T'_n \in \textbf{superClasses}(p, T_n)$

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 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:

```
I=interface { ..}
C={implements I ..}
A={C m()=..}
t={B={ } T={method B m()} ..}
Res=t<T=A>
```

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 `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()}
  Node:{Edges connections()}
  Nodes:set<Elem=Node>//note that we do not specify equals/hash
  static Bool isConnected(Nodes nodes)=
    if(nodes.size()=0) then true
    else getConnected(nodes.asList().head()).size()==nodes.size()
  static Nodes getConnected(Node node)=getConnected(node,Nodes.empty())
  static Nodes getConnected(Node node,Nodes collected)=
    if(collected.contains(node)) then collected
    else connectEdges(node.connections(),collected.add(node))
  static Nodes connectEdges(Edges e,Nodes collected)=
    if( e.isEmpty()) then collected
    else connectEdges(e.tail(),collected.add(e.head().end()))
  }
```

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.

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

```
  City:{}
  Streets:priorityQueue<Elem=Street><+{
    Int geq(Street e1,Street e2)=e1.size()-e2.size()}
  }<+{
  Streets:{}
  City:{Streets connections, Int index}//index identify the node
  Cities:set<Elem=City><+{
    Bool eq(City e1,City e2) e1.index==e2.index
    Int hash(City e) e.index
    }
  Cities cities
  //more methods
  }
MapUtils=graphUtils<Nodes=Map.Cities>
//infers Nodes.List, Node, Edges, Edge
```

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

7.2 Loading libraries

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

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

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

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

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

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

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

```
library={//this code is fully self contained
  N={}
  C={ Bool equal(C x)}
  S={N size(), C getChar(N index)}
  S myLibFunction(S x)=....
  Map=interface{
```

```
    S string()
    C char()
    N integer()
    }
  }
...
Int={..}
Char={..}
String={..}
Map=interface{String string() ... }
LoadedLib=library<Map=Map>
```

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 Appendix?

PUT LATER?However, he 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

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

9 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; This0 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'
```

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:

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: $p_{.\mathbf{push}(D)} = p'$, allowing to work with programs as if they was stacks, and $p_{.\mathbf{min}(T)} = T'$, denoting the shortest type T' referring to the same nested class of T. We define $T_{.\mathbf{from}(T',j)}$ and $L_{.\mathbf{from}(T,j)}$; we omit all the trivial propagation cases of form $M_{.\mathbf{from}(T,j)}$, $K_{.\mathbf{from}(T,j)}$ and $e_{.\mathbf{from}(T,j)}$. 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 $p[T]$:

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

$$(;\_, C{=}L, \_)(\mathtt{This}_0.C.Cs) = L(Cs)$$
$$p_{.\mathbf{push}(\_=L)}(\mathtt{This}_0.Cs) = L(Cs)$$
$$p_{.\mathbf{push}(\_)}(\mathtt{This}_{n+1}.Cs) = p(\mathtt{This}_n.Cs)$$

$$\mathbf{members}(\mathtt{interface}?\ \{\_;\ Mz;\ \_\}) = Mz$$

$$L(m) = \mathbf{members}(L)(m)$$

$$L(C) = \mathbf{members}(L)(C)$$

$$dom(L) = dom(\mathbf{members}(L))$$

$$mdom(L) = \{m \in dom(L)\}$$

$$(Mz, \mathtt{private}?C{=}\_)[C = E] = Mz, \mathtt{private}?C{=}E$$

$$LV(\emptyset) = LV$$
$$L(C.Cs) = L(C)(Cs)$$

$$L[empty = E] = E$$
$$\mathtt{interface}?\ \{Tz;\ Mz;\ K?\}[C.Cs = E] =$$
$$\quad \mathtt{interface}?\ \{Tz;\ Mz[C = Mz(C)[Cs = E]];\ K?\}$$

$$p_{.\mathbf{min}(\mathtt{This}_{n+1}.id_n.Cs)} = p_{.\mathbf{min}(\mathtt{This}_n.Cs)}$$
where $p = id_0{=}L_0 \ldots id_n{=}L_n\_; Ds$

otherwise $p_{.\mathbf{min}(T)} = T$

$$\texttt{This}_n.Cs_{.\mathbf{from}(T,j)} = \texttt{This}_n.Cs \quad \textit{with } n < j$$
$$\texttt{This}_{n+j}.Cs_{.\mathbf{from}(\texttt{This}_m.C_1 \ldots C_k, j)} = \texttt{This}_{m+j}.C_1 \ldots C_{k-n} \quad \textit{with } n \leq k$$
$$\texttt{This}_{n+j}.Cs_{.\mathbf{from}(\texttt{This}_m.C_1 \ldots C_k, j)} = \texttt{This}_{m+j+n-k}.C_1 \ldots C_{k-n}Cs \quad \textit{with } n > k$$
$$\{\texttt{interface}?Tz;\ Mz;\ K\}_{.\mathbf{from}(T,j-1)} = \{\texttt{interface}?Tz_{.\mathbf{from}(T,j)};\ Mz_{.\mathbf{from}(T,j)};\ K_{.\mathbf{from}(T,j)}\}$$
$$(DL_1 \ldots DL_n; \_, t\texttt{=}LV)[t] = p_{.\mathbf{min}(LV_{.\mathbf{from}(\texttt{This}_n,0)})}$$
$$p[T] = p_{.\mathbf{min}(\texttt{interface}?\ \{Tz_{.\mathbf{from}(T,0)};\ Mz_{.\mathbf{from}(T,0)};\ \})} \quad \text{where } p(T) = \texttt{interface}?\ \{Tz;\ Mz;\ K?\}$$

sdgsd

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

where $\mathrm{L} = \mathrm{a}$

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

$$
\begin{aligned}
(DL_1 \ldots DL_n; \_, t\texttt{=}LV, \_)[t] &:= LV_{.\mathbf{from}(\texttt{This}_n)}\\
p[T] &:= p_{.\mathbf{minimize}(p(T)_{.\mathbf{from}(T)})}
\end{aligned}
$$

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

10 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

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`

$$a ::= b \quad \text{c}$$

aa`hi`**`Hi`**`class`qaq $a ::= b \quad \text{c}$

$$a ::= b \quad \text{c}$$

`}}][()]`

$$\frac{a \xrightarrow[b]{} c \quad \forall i < 3a \vdash b : \mathrm{OK} \qquad \forall i < 3a \vdash b : \mathrm{OK}}{1 + 2 \rightarrow 3}\ (\textsc{top}) \quad \begin{matrix} a \\ b \\ c \end{matrix}$$

References