Using nested classes as associated types.

Authors omitted for double-bind review.
Unspecified Institution.

Abstract

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

2012 ACM Subject Classification Dummy classification

Keywords and phrases Dummy keyword

Digital Object Identifier 10.4230/LIPIcs.CVIT.2016.23

1 Introduction

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

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

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

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.
- 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

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

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

In the context of nested classes, types are paths. Syntactically, we represent them as relative paths of form `This`$n.Cs$, where the number n identify the root of our path: `This0` is the current class, `This1` is the enclosing class, `This2` is the enclosing enclosing class and so on. `This`$n.Cs$ refers to the class obtained by navigating throughout Cs starting from `This`n. Thus, `This`0 is just the type of the directly enclosing class. By using a larger then needed n, there could be multiple different types referring to the same class. Here we expect all types to be in the normalized form where the smallest possible n is used.

Code literals L serve the role of class/interface bodies; they contain the set of implemented interfaces Tz, the set of members Mz and their (optional) state. In the concrete syntax we will use `implements` in front of a non empty list of implemented interfaces and we will omit parenthesis around a non empty set of fields. 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 `This`$n.Cs.C.Cs'$, if C denotes a private nested class, then Cs is empty.

Expressions are used as body of (static) methods and for the main expression. They are variables x (including `this`) and conventional (static) method calls. Field access and `new` expressions are included but with restricted usage: well formed field accesses are of form `this`$.x$ in method bodies and $v.x$ in the main expression, while well formed `new` expressions have to be of form `new 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 difficulty. Values are of form `new` $T(vs)$.

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

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

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

2.1 Redirect

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

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

This would flatten into

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

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

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

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

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

2.2 Preferential sum

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*/
  }
```

In the ending of this paper we will show how redirect and sum allows to encode difficult code reuse patterns in a much more convenient way that Java, Scala or Rust.

$$\begin{array}{rll}
\mathcal{E}_V ::= & \square \mid \mathcal{E}_V \texttt{<+}\ E \mid LV \texttt{<+}\ \mathcal{E}_V \mid \mathcal{E}_V \texttt{<} Cs \texttt{=} T \texttt{>} & \text{context of library-evaluation}\\
LV ::= & \texttt{\{}\,\texttt{interface}\ Tz;\ amtz\texttt{\}} \mid \texttt{\{}Tz;\ MVs\ ;\ K?\texttt{\}} & \text{literal value}\\
MV ::= & C\texttt{=}LV \mid mt & \\
\mathcal{E}_v ::= & \square \mid \mathcal{E}_v.m(es) \mid v.m(vs\ \mathcal{E}_v\ es) \mid T.m(vs\ \mathcal{E}_v\ es) & \\
DL ::= & id\texttt{=}L & \text{partially-evaluated-declaration}\\
DV ::= & id\texttt{=}LV & \text{evaluated-declaration}\\
Mid ::= & C \mid m & \text{member-id}\\
p ::= & DLs;\ DVs & \text{program}
\end{array}$$

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

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

An S represents what the top-level source-code form of our language is, it's just a sequence of declarations and a main expression. The most interesting form of the grammer is a p, it is a 'program', used as the context for many reductions and typing rules, on the LHS of the ; is a stack representing which (nested) declaration is currently being processed, the bottom (rightmost) DL represents the D of the source-program that is currently being processed. Th RHS of the ; represents the top-level declarations that have allready been compiled, this is neccessary to look up top-level classes and traits.
To look up the value of a type in the program we will use the notation $p(T)$, which is defined by the following, but only if the RHS denotes an LV:

$$(;\_, C = L, \_)(\texttt{This}0.C.Cs) := L(Cs)$$
$$(id = L, p)(\texttt{This}0.Cs) := L(Cs)$$
$$(id = L, p)(\texttt{This}n+1.Cs) := p(\texttt{This}n.Cs)$$

To get the relative value of a trait, we define $p[t]$:
$$(DLs;\_, t = LV, \_)[t] := LV[\texttt{This}\#DLs]$$

To get a the value of a literal, in a way that can be understand from the current location ($\texttt{This}0$), we define:
$$p[T] := p(T)[T]$$

And a few simple auxiliary definitions:
$$Ts \in p := \forall T \in Ts \bullet p(T) \text{ is defined}$$
$$L(\emptyset) := L$$
$$L(C.Cs) := L(Cs) \text{ where } L = \texttt{interface}?\ \{\_;\ \_, C = L, \_;\ \_\}$$
$$L[C = E'] := \texttt{interface}?\ \{Tz;\ MVs\ C = E'\ Ms;\ K?\}$$
$$\text{where } \mathrm{L} = \texttt{interface}?\ \{\mathrm{Tz};\ MVs\ \mathrm{C} = \_\ Ms\ ;\ \mathrm{K}?\}$$

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

```
Define Ds e --> Ds’ e’
============================================================
DVs’ |- Ok
empty; DVs’; id | E --> E’
(compile)---------------------------------------- DVs’ subsetof DVs
DVs id = E Ds e --> DVs id = E’ Ds e

DVs |- Ok
DVs |- e : T
DVs |- e --> e’
(main)---------------------------------- for some type T
DVs e --> DVs e’
```

3 Compilation

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

```
Define p; id |- E --> E’
=====================================================
p; id |- E --> E’
(CtxV) ---------------------------------------------
p; id |- CtxV[E] --> CtxV[E’]

id = L[C = E], p; C |- E --> E’
(L) --------------------------------------------- // TODO use fresh C?
p; id |- L[C = E] ---> L[C = E’]

(trait) -----------------------------------
p; id |- t -> p[t]

LV1 <+p’ LV2 = LV3                 p’ = C’ = LV3, p
(sum) -------------------------------------- for fresh C’
p; id |- LV1 <+ LV2 --> LV3

// TODO: Inline and de-42 redirect formalism
(redirect) ------------------------------------LV’=redirect(p, LV, Cs, P)
p; id |- LV(Cs=P) -> LV’
```

4 The Sum operation

The sum operation is defined by the rule $L1 < +pL2 = L3$, it is unconventional as it assumes we allready have the result ($L3$), and simply checks that it is indead correct. We believe (but have not proved) that this rule is unambigouse, if $L1 < +pL2 = L3$ and $L1 < +pL2 = L3'$, then $L3 = L3'$ (since the order of members does not matter for Ls).

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

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

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

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

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

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

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

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

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

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
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 ..
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'
```

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

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

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

`}}][()]`

$$\dfrac{\overset{(\text{TOP})}{}\; a \underset{b}{\rightarrow} c \quad \forall i < 3 a \vdash b : \text{OK} \qquad \forall i < 3 a \vdash b : \text{OK}}{1 + 2 \rightarrow 3}\ \begin{matrix} a \\ b \\ c \end{matrix}$$

References