Metaprogramming in 42: Declaratively generate imperative behaviour by functional reasoning

No Author Given

No Institute Given

Abstract. Quasi Quotation [**?**,**?**,**?**] is a very expressive metaprogramming technique: it allows expressing arbitrary behaviour by generating arbitrary abstract syntax trees. However it is hard to reason about Quasi Quotation statically, the process is fundamentally low-level, imperative and bug prone. In this paper we present Iterative Composition as an **effective alternative** to Quasi Quotation. Iterative Composition describes a code composition algebra over established code reuse techniques, similar to trait composition and generics. Our main contribution is that by applying functional reasoning (such as induction and folds) over those well-known operators we can generate arbitrary behaviour. This allows us to reuse the extensive body of verification research in the context of object-oriented languages to verify the properties of the code generated by Iterative Composition. Finally, we present a prototype implementation of Iterative Composition in the context of the 42 language.

1 Introducing Quasi Quotation

Lisp [**?**], MetaML [**?**], Template Haskell [**?**] and many other approaches use Quasi Quotation (QQ). This can be supported by two kinds of special parenthesis as a syntactic sugar to manipulate Abstract Syntax Trees (ASTs). Lisp uses ‘ and ,, while here we use `[|  |]` and `$( )` (as Template Haskell) for better readability.

The following example explains their meaning:

```
Int res0=x*x //normal code
Expr<x:Int⊢Int> res1=[| x*x |] //new Mul(new Var("x"),new Var("x"))
Expr<x:Int⊢Int> res2=[| x* $(12+3) |] //new Mul(new Var("x"),new Lit(15))
```

Here `res1` is initialized using a “quotation” of code. This is equivalent to generating the abstract syntax tree by hand, as shown in the comment. `res2` is initialized using a “quasi-quotation” of code: a chunk of code with a hole, that is filled by executing an expression.

There are different ways to type QQ. In an expression based language, the simplest way is to just have a primitive `Expr` type, representing every type of code. This ensures the result is syntactically well formed, but it allows for the generation of ill-typed code. Another option, for example used by MetaML, is to have a parametrized type. Here we use `Expr<`$\Gamma \vdash T$`>`; where Γ keeps track of the free variables and T is the expected type of the result. This approach is restrictive (see [**?**]) but ensures that all the resulting code is well typed.

Usually programming with QQ requires thinking about the desired method body, and often allows generating a more efficient body by generating code specialized for

some input value. A typical example is about generating a `pow` function, where the exponent is well known. The "inefficient" version would be:

```
fun power(x:Int,n:Int):Int
  = if (n=0) then 1 else x*power(x,n-1);
power7=λ x:Int. power x 7;
```

A more "efficient" version using QQ would be:

```
fun powerAux(n:Int):Expr<x:Int⊢Int>
  = if (n=0) then [|1|] else [|x * $(powerAux(n-1)) |];

fun powerGen(n:Int): Int->Int
  = compile([| λ x. $(powerAux(n)) |]);

power7=powerGen 7;
```

As you can see, by generating the abstract syntax tree, we can obtain exactly:

```
power7=λ x.x*x*x*x*x*x*x*1;
```

On most machines, `power7` runs faster than `λ x:Int. power x 7`. Metaprogramming applications include more than just speed boosts, but we start with this example because it is very popular and simple.

The code generator above is quite compact, but it is actually **hiding** (not removing) the complexity of meta-programming. A common approach to make the code more explicit is to extract logical concepts as functions. We can see that the code is proceeding in an inductive fashion: we know the code for `pow 0`, and given the code for `pow n` we can create the code for `pow (n+1)`. Thus we define `base` and `inductive` functions, and we use them inside `powerAux`:

```
fun base():Expr< x:Int⊢Int >
  = [| 1 |]
fun inductive(code:Expr< x:Int⊢Int >):Expr< x:Int⊢Int >
  = [| x * $(code) |]

fun powerAux(n:Int):Expr< x:Int⊢Int >
  = if (n=0) then base()
   else inductive ( powerAux(n-1) );
```

Then, we have to bind `x:Int` to a parameter in a function. This is an important conceptual action and thus we make it a function:

```
fun lambdaX(code:Expr< x:Int⊢Int >):Expr<⊢Int->Int >
  = [| λ x. $( code )|]

fun powerGen(n: Int):Int->Int
  = compile(lambdaX(powerAux(n) ))
```

The code we obtain is much larger, but is not logically more complex — it is just showing the logical structure better. Note how since QQ works near the code representation, a function `Int->Int` is radically different from code with a free variable `x:Int⊢Int`, while they are logically similar concepts.

We propose Iterative Composition (IC): while the unit of composition in QQ is the single AST node, IC enforces a higher level of abstraction and does not work directly on the AST. The unit of composition in IC is a *Library*: a class body, containing methods and possibly nested classes. Libraries are self contained in the sense that

they contain no free variables. This avoids all scope-extrusion related problems, and (as shown later) enforces local reasoning.

IC has already been presented in other work [**?**]; IC expressive power is shown by examples, but is not compared with QQ; moreover such works suggested IC power is inferior to QQ. The core idea of IQ is to *rely on operators of code composition inspired by normal code reuse*, but lifted at the expression level. As a concrete example, in Java operators `+` and `*` can be used in the expression `1+2*3`, but the operator `extends` can only be used in the specific context of class declaration, as in

```
class A extends B { /*some code*/ int m(){return 1+super.m();}}
```

In our proposed approach we lift `extends` and code literals to operator and constants that can be used in conventional expressions. We would write the former example as:

```
A = Override[m()<-superM()](
  {/*code of B*/},
  { /*some code*/ int m(){return 1+this.superM();}}
  )
```

We support the conventional super call mechanism by annotating the operator with the expected super call name: `Override[m()<-superM()](...)`.

Going back to our `pow` example, in IC, the base and inductive cases look like this:

```
Pow = {
  static method Library base()
   ={ method Num pow(Num x)= 1 } //Code literal with 1 method "pow(x)"

  static method Library inductive()
   ={ //Code literal with 2 methods: "pow(x)", "superPow(x)"
    method Num pow(Num x)= x*this.superPow(x)
    method Num superPow(Num x) //no body: it is an abstract method
    }
  static method Library inductive(Library that)
   = Override[pow(x)<-superPow(x)](that, this.inductive())

  static method Library generate(Num y)
   = if (y==0) then this.base();
     else this.inductive(generate(y-1))
  }
...
Pow7 = Pow.generate(7)
//That would reduce into the desired code as follows:
Pow7 ={method Num pow(Num x)=x.x*x*x*x*x*x*x*x*1}
```

In more detail:

`base()` is a method with no parameter with `Library` return type. This is equivalent to a non parametrised version of `Expr` in QQ. However, our approach still guarantees that all the results are well typed. `base()` returns a class with a single method `pow(x)`, returning 1.

For the inductive case, the method `pow(x)` of `inductive()` is defined in terms of another method (`superPow(x)`), representing the delegation to the former case in the inductive reasoning. Note that the declaration of `superPow(x)` is an abstract method: a method without body.

To actually perform the induction we use `inductive(that)`. Note how we use the operator `Override` inside of a normal method body. Method `generate(y)` uses recursion to **iteratively compose** the result, using induction starting from the base case. Note how this method is logically identical to `powerAux`. However, since we always work on the actual self contained code neither `lambdaX` nor `compile` are needed.

Our approach builds on top of code composition operations like multiple inheritance and generics. The literature offers [**?**,**?**,**?**] many successful efforts about proving the *semantic correctness* of code containing inheritance and generics. On the other hand, static verification of code generated with meta programming is an open research problem.

We speculate our approach may offer the opportunity to solve this problem, and by construction generate statically verified code, by reusing techniques originally developed to verify normal object oriented code.

The contributions of our work are as follows:

- Show how to apply conventional object oriented verification techniques to IC.
- Motivate the interpretation of code composition operators as declarative operators.
- Show that IC is as expressive as QQ, and that generating code using a composition algebra is a flexible and simple technique, if combined with reasonable programming patterns.

2 Compile-time verification: Pow with contracts

In this section we show how conventional verification techniques for OO can be applied on top of IC. While IC is already presented in [**?**], the idea and the techniques to use IC for verification are novel contributions.

Looking back at the code to generate `pow` in QQ and in IC, we can note how the method `inductive(n)` in IC is equivalent to the method `inductive(n)` in QQ. However, the method `inductive()` in IC has no equivalent in QQ. `inductive()` returns the complete code of a class with an abstract method, that is then used as a handler to inject behaviour. There is no way in traditional QQ to express this same concept, and this is the crucial point that makes our approach easier to verify. QQ is based on *quasi* quotations, that is, *parametric code* that will become *complete code* when you fill in the holes. While in IC, every code literal is *complete code* (with the usual OO semantics where methods can be overridden). Unsurprisingly, proving correctness of parametric code is much harder than for complete code.

We show this in the next example, where we handle `pow` as before, but while verifying that the result encodes the right `pow` function. We will use similar notation to JML, with `@ensure` and `@result`. We include `@ensureRV` for conditions that we expect to be verified at run time, instead of statically verified. Many static verification approaches use some automatic theorem proving to figure out the proofs. Here, for completeness, we introduce a `@proof` notation, where we insert the full proof, to show what we expect a theorem proving to generate. To make the verification easier, we generate both the `pow(x)` method and a `exp()` method, keeping track of the accumulated exponent. This time we will use explicit iteration instead of recursion, just to show that our approach is not bound to any of those styles.

```
Pow:{
  static method Library base()
    ={
   //@ensures exp()=0
    method Num exp()=0

   //@ensures pow(x)=x^this.exp()
    method Num pow(Num x)= 1
    }
  ...
  }
```

We start by examining the base case of our induction: we annotated our code with trivial specifications. A code verifier could statically verify this code once and for all when `Pow` is type-checked. Then, lets look at the `inductive()` step:

```
Pow:{
  static method Library base() =... //as before
  static method Library inductive()
    ={
   //@ensures @result=1+this.superExp()
    method Num exp()=1+this.superExp()

    method Num superExp() //no body: it is an abstract method

   //@ensures @result=x^this.exp()
   //@proof:
   // @result->x*this.superPow(x)//left side
   // =x* x^this.superExp()= x^1+this.superExp()
   // x^this.superExp()->x^1+this.superExp() //right side
    method Num pow(Num x)= x*this.superPow(x)

   //@ensures pow(x)=x^this.superExp()
    method Num superPow(Num x) //no body: it is an abstract method
    }
  ...
  }
```

Here the annotations are much heavier. We show the full proof that a theorem proving may generate. Note that we need to rely on the contract of `superPow(x)` (a concept completely hidden in the QQ approach). Note how also this code, and its possibly non-trivial proof, can be verified once and for all when the code of `Pow` is compiled.

While the library literals above have contracts, neither methods `base()` nor `inductive()` have any contract: statically we only know they return a `Library`.

Here we put the pieces together:

```
Pow:{...
 //@requiresRV y>=0
 //@ensuresRV @result.exp().ensures = (@result = y)
 //and @result.pow(x).ensures = (@result = x^y)
  static method Library generate(Num y){
```

```
 var Library res=this.base()
 for(i in Range(y)){
   res=Override[exp<-superExp, pow<-superPow](res, this.inductive())
  //Override will check the composition is correct.
  //that is: renaming in res pow(x) and exp() as superPow(x) and superExp(),
  // res.superPow(x).ensures satisfyAtLeast inductive.superPow(x).ensures
   }
//here we statically know that res is statically verified;
 return res }   }//RV happens here
```

Note how when the `Override` operation runs, it will ensure the resulting class is statically verified: every `Library` value has statically verified contracts. Operations that generate `Library` values (as `Override`) will also compose those contracts, while verifying such contract composition is sound.

This may, of course fail. We think of those operations as compile time operations lifted at execution. `Override` producing an error is equivalent to getting compilation error about invalid `extends`. Note how this is not a form of run-time verification, but just the expected semantics of those operators. `Override` does not guarantee to produce correct code all the time, in the same way a parser does not guarantee to produce a correct AST for all possible strings: producing errors is part of `Override` expected behaviour.

In our example `Override` checks that `res.superPow(x).ensures` satisfy at least `inductive().superPow(x).ensures`. Finally, the `return` step triggers the run-time verification, and checks the contracts of `res`. Indeed before the return, we statically know that `res` is statically verified, but we have no guarantee of what its contracts are saying. Indeed we are aiming for a run-time verified `Pow.generate(y)`. We believe this strikes a fundamental balance and is analogous to what a verifying compiler [**?**] should do.

This separation of concerns in our verification proposal is a key simplification in our model: all library values are well typed and statically verified all of the time, but what those contracts entail can be only verified at run-time, and the (meta-)programmer can choose when to do it.

To summarize:

- At compile time, once and for all, we statically verify that all method bodies respect the (static) contracts on the method headers. This can be handled exactly as static verification is normally handled in OO languages.
- During code composition we match contracts (*satisfyAtLeast*). While sound and complete contract matching could be infeasible, there are simple restrictions making it trivial. For example any contract *satisfyAtLeast* the empty contract, and two syntactically equivalent contracts *satisfyAtLeast* each others. Expressiveness of matching is probably not important since the (meta-)programmer is going to write those contracts on purpose to make them matchable.
- We guarantee all library literals used by the execution are statically verified; but statically we do not know the details of their contracts.
- Run-time verification is a simple and effective way to close this last gap, verifying that the result is not only *self-consistent* but also the expected one.

3 IC is as expressive as QQ

So far we have presented how using pre-post conditions benefits the safety of our approach with respect to QQ. **Independently** from those benefits, we now aim to show that IC is as expressive as QQ, and possibly easier to use on the large scale.

To this aim, we will show how, from a very specific point of view, OO languages are declarative. We will then show that `Override` and other composition operators are declarative operators, and that they can be combined in a general purpose way to synthesize new declarative operators.

Except for the algebra of composition operators, that is already presented in [**?**], all the considerations, ideas and techniques presented in this section are novel contributions.

Object-orientation (OO) points to declarative languages

Imperative programming asks you to write down the detailed computational steps your machine should perform. This allows the programmer to reason about their programs by "emulating computer" in their mind. As we all know too well, this tempting approach to understand program behaviour does not scale. We can see OO as taking imperative programming towards a more declarative style:

Subtyping A dynamically dispatched method invocation gives us no certainty of the detailed computational steps that are going to be performed, and requires programmers to reason in terms of the abstract contract of the method: that is the desired properties of the result as function of the arguments. The programmers are still in control of the detailed behaviour, but such control is delegated to the programmer that instantiated such object. The receiver object takes up the role of the *reasoning engine* that takes care of transforming the programmer request into a result. In many OO languages, this reasoning engine amounts to just following a pointer. However, as the Visitor Pattern and other design patterns show us, this is sufficient to encode very interesting behaviour resolution. Moreover, some languages implement multiple dispatch [**?**], making this reasoning engine a lot more expressive.

Subclassing From this point of view, inheritance represents a similar loss/transfer of control: The programmer does not know the full set of methods a class offers: this depends on the methods offered by the base class, and if in a future release the heir is enriched with more methods, such methods will also be injected into the subclass.

This allows a limited declarative programming style. For example, in

```
class Rounded extends Button{ ... }
```

We declare `Rounded`s to be buttons in a way that is parametric on what `Button`s concretely are. The programmer of `Rounded` just has to specify little bits of behaviour to personalize the kind of buttons. The concrete code is then automatically derived by composing it with the existing code of `Button`. You can see `Button`s as following the double role of both a class and a class generator/decorator. You can think of this as giving some suggestions to `Button` on what code to generate when creating the class `Rounded`.

It is a very limited declarative language aimed to code composition. In Java, the *code composition reasoning engine* is part of the language. In Java is a very simple minded reasoning, but it gets much more expressive in other languages, like C++.

Since the behaviour of inheritance is set in stone, `Button` is not very active in deciding about `Rounded`. However, this point of view lets us interpret many approaches (traits [**?**], mixins [**?**], generics [**?**], family polymorphism [**?**]) as ways to enrich what `Button` can do to generate the required `Rounded` from the hints provided by the programmer: a way to enrich the declarative, domain specific language that the programmer uses to instruct base classes into generating their heirs.

In this article we wish to temporarily forget that base classes can be used as class or as types, and focus on their active role in the generation of code. In this context we will call them Class Decorators. Compile-time meta-programming [**?**] is a good way to give Decorators a mind of their own, so that they can perform arbitrary complicated steps while generating their heirs.

An algebra of composition operators

In many class based object oriented languages, a class can inherit the code from one or more parents. In order to lift this capability as a metaprogramming operation we define an algebra of code composition, where the values are `Library` literals and the operations are the composition operators.

- A `Library` is a code literal: a pair of balanced curly brackets with methods and nested classes inside. Thanks to nested classes, a code literal can contain a large portion of code with cooperating classes, possibly encapsulating a whole library.
- A *composition operator* is a functionality taking in input `Library`s and producing a `Library` result, or a *composition error*.

In the following example, class `C` contains a single static method `foo()` returning a `Library` literal. We can use class `C` and the `Override` composition operator to attempt creating classes `D1,D2,D3`.

```
C:{
  static method Library foo()={method Num m()=2  }
  }
D1:C.foo()
D2:Override[](C.foo(),{method Num n()=0 })
D3:Override[m()<-superM1(),superM2()](C.foo(),C.foo(),{
  method Num m()=3+this.superM1()+this.superM2()
  //superM1 and superM2 used for super calls
  method Num superM1()
  method Num superM2()
  })
```

By a process known as flattening [**?**], we get the following results for `D1,D2,D3`:

```
D1:{method Num m()=2  }
```

`C.foo()` is *executed at compile time* and we use the result to initialize `D1`.

```
D2:{method Num m()=2  method Num n()=0  }
```

`Override` works like inheritance; in this case since there is no conflict between the two code literals, the result is a library literal containing both

```
D3:{method Num m()=3+2+2 }  //super calls may be inlined
```

We write `Override[m()<-superM1(),superM2()](..)` to support calling the multiple conflicting versions for method `m()`. Here we try to compose the same code literal twice. This may fail, since both (identical) sides implement method m. However, `Override` treats the last library value in a special and preferential way: if there are multiple conflicting implementations of the same method (`m()` in this case) the last value can redefine such method and avert the conflict.

The `Override` operator we are proposing works as trait composition [**?**]: we can compose multiple `Library` values in an associative and commutative way; except for the last parameter, which enjoys preferential composition: its methods can override methods in the other libraries. This privilege is equivalent to the one *glue code* enjoys in the original trait model [**?**](section 3.3).

Many variations of this composition operator has been presented in the literature, and an exhaustive understanding of its behaviour is not needed to understand this paper.

In addition to `Override`, we will use many other composition operators. We believe it is fair to consider these high level operations on code as *declarative*: the programmer does not specify the details of the source code involved, but only reasons at the method call level. Concretely, in the former example this means that the implementation of method `m()` inside class `C` is not relevant: only the behaviour/contract of such method is important.

Patterns for generating arbitrary behaviour

The general idea is to define a new declarative operator using other declarative operators. Operators can generate imperative code, but such code is not directly specified, it is synthesized by inductive reasoning. As an example of a new operator, we will consider `Stringable`; it can generate an opportune `toS()`/`toString()` method inside of code literals. In our example, the resulting code is going to be the definition of class `A`. Note the declarative feel of this operation: we just specify the desired properties of the result (having a `toS()` method), and the detailed implementation is automatically derived by the shape of the library literal provided.

```
A: Stringable <>< { .. }
```

Babel fish operator `<><`

Since the main goal of our approach is to generate code, we introduce a *code generation operator* `<><`, called Babel fish. It is a binary operator, taking a Decorator and a `Library`, and producing another `Library`.

We consider a language with normal operator overriding thus `Stringable <>< {.. }` is equivalent to `Stringable.babelFish({.. })`

We choose `Stringable` since its behaviour is not obvious: it is required to examine `{ .. }` in order to propagate `toS()` to all the fields. This is also an important example: other operations like `equals(that)`, `hashCode()` and `compare(that)` also rely on propagating the operation over the fields. That is, all of these operations can be implemented by following the same programming pattern.

We can also see `Override` as a class decorator, and write `Override(a,b,c)<><d` to highlight that the last parameter enjoys preferential composition.

Objective code

In order to have a clear goal, let's imagine a tentative code we would like to obtain:

```
Stringable<><{S name Num age method S sayHi() ="Hi, i'm "++this.name()}
```

should evaluate into

```
{S name Num age
 method S toS()=
   "["++this.name().toS()++", "++this.age().toS()++"]"
 method S sayHi()=
   "Hi, i'm "++this.name()
}
```

How to obtain this kind of result by using IC instead of QQ? The main idea is to try to not think about the actual code but about its behaviour, and how to decompose it into functions, and how to put these functions together.

Top level operation

We are going to show the implementation of `Stringable` in a Java like language where we are going to repeat parameter names in the call site when this can improve readability. The following is our entry point: we define a babelFish static method returning a `Library`.

```
Stringable:{
  static method Library <><(Library that) = (
    libs=this.baseCases(that) //1
    acc=this.fold(libs,acc:{method S toS()= ""})//2
    res=this.close(acc)//3
    Override[](res)<><that//4
    )
...
```

The method uses inductive reasoning and is divided into 4 meaningful steps:

1. Using the input, we generate the base cases; in this case computing `toS()` for a class with a **single specific field**.
2. Then we **fold** our base cases into a single solution; in this case we specify how to merge two `toS()` implementations.
3. Finally we **close/wrap** our implementation (adding `[]`).
4. Our result `res` would now expose **only** the `toS()` method. We then use `Override` to compose our `toS()` with the original input.

Phase 1: base cases

To define our base case, we first declare a method returning a constant `Library` value: an approximation for a class whose `toS()` method delegates to a field.

```
static method Library baseTrait() = {
  T f
  T:{method S toS()} //nested class
  method S toS()= this.f().toS()
  }
```

It has a field called `f` of type `T`. `T` is a type with an abstract `toS()` method returning a string `S`. The `toS()` method of the top level class, just propagates out the behaviour of `T.toS()`.

Note how we call such method a "trait": inspired by trait composition, this method represents a reusable piece of code, where all the dependencies are explicit. That is, this code is *general* (`T` offers only the required `toS()` signature) but is not *generic* in the Java/C# sense.

Now we can define our `baseCases(that)` method.

```
static method Libs baseCases(Library that) = Libs[
    RedirectType(path:"T" into:fi.type())<><
    Rename(selector:"f" into:fi.selector())<><
    this.baseTrait()
    | with fi in Fields(that)]
```

We extract the fields by observing with introspection our input class. As available in both Haskell and Python, we assume we can use the common syntax for list comprehension: `CollectionType[e| x in list]` Where the expression `e` denotes the entries in the newly generated collection. In this case we generate our entries with the following interesting code:

```
    RedirectType(path:"T" into:fi.type())<><
    Rename(selector:"f" into:fi.selector())<><
    this.baseTrait()
```

Here we apply two decorators to our `baseTrait`. First we rename our field name from `f` to the current field name, then we redirect `T` into the current field type. This redirect is not an obvious operation, and is the kind of code manipulation where IC shines with respect to QQ, since it allows one to express logic problems that would end up hidden in QQ. The operator `RedirectType` is more general than `Redirect` as shown in [**?**]. `Redirect` is a powerful operator: it deletes the nested class `T` and replaces all the occurrences of `T` with occurrences of such external type. `Redirect` is indeed a powerful form of generics, where generic types can be expressed as nested classes with abstract members. However, logically `Redirect(path:"T"into:fi.type())` would work only in case of `fi.type()` being a type defined **outside** of the decorated class. In case of `fi.type()` denoting a **nested class** inside of the decorated class, we would need to perform `Rename(path:"T"into:fi.type())` instead. `RedirectType` just switches between the two options and calls the right operation internally.

`Rename` and `Redirect` leverage on the conventional nominal type system to avoid errors: `fi.type()` returns a general `Type` type, and offers operations to extract either the internal path (typed `Path`, that would be a well typed argument for `Rename`) or the external class (typed `Class`, that would be a well typed argument for `Redirect`).

Note how the convenient and expressive operator `RedirectType` is a **derived operator**, exactly as `Stringable`, and is expressed in the language itself. Once and for all the programmer can understand the complexity of a specific problem and encapsulate it in a solution.

On the other hand, while coding this kind of program manipulation with QQ, most programmers will just forget to handle the case where the type of a field is a nested class of the input. Depending on the target language, this could be a big problem later on.

To give an example of the result of `baseCases(that)`, if we were to call

```
baseCases({
  S name Num age
  method S sayHi() = "Hi, i'm "++this.name()
  })
```

we would obtain

```
Libs[
  {S name   method S toS() = this.name().toS()};
  {Num age  method S toS() = this.age().toS()};
  ]
```

Note how there is no trace of the `sayHi()` method, and the nested classes named `T` have been removed to point at the external types `S` and `Num`

Phase 2: folding

Next we fold all our base cases into a single `toS()` method. Traditionally, to fold a list of values into a single value, a binary operation is needed. However, here we fold code representing methods, so we need a lifted version of fold: we supply a `Library` value where a method uses two alternative versions of itself.

```
    {
    method S toS()= this.superToS1()++", "++this.superToS2()
    method S superToS1()
    method S superToS2()
    }
```

This corresponds both to multiple inheritance, where a new version of a method is obtained by composing the two super implemenations, but also to a binary fold operation from `S superToS1()` and `S superToS2()` into a `S toS()` result. Thus, to apply this folding, we use `Override` and we leverage on the multiple inheritance interpretation:

```
  static method Library fold(Libs that,Library acc) = {
    if that.isEmpty() (return acc)
    newAcc=Override[toS()<-superToS1(),superToS2()](that.left(), acc)<><{
      method S toS()= this.superToS1()++", "++this.superToS2()
      method S superToS1()
      method S superToS2()
      }
    return this.fold(that.withoutLeft(),acc:newAcc)
    }
```

The base case of this recursive code is the empty list, where `acc` is returned, otherwise `Override` provides us with multiple inheritance where `superToS1()` and `superToS2()` allow us to call super from the first/second parameter. Of course we can not just compose `acc` with the top of our list: they both offer a `toS()` method. We need to provide extra code to override the conflicting implementation and provide new behaviour, in this case, the two strings separated by a comma.

Continuing with the example from before, starting from the same code literal we would now obtain:

```
{S name Num age
 method S toS() = this.name().toS()++", "++this.age().toS()
}
```

Phase 3: wrapping

We could be satisfied with such result, but often we wish to wrap our final result to present it better.

```
  static method Library close(Library that) =
    Override[toS()<-superToS()](that)<><{
      method S toS() = "["++this.superToS()++"]"
      method S superToS()
      }
  }
```

In the `close(that)` method here, `Override` adds `[]` around the string.

Starting from

```
{S name Num age method S sayHi() "Hi, I'm "++this.name()}
```

we would now obtain

```
{S name Num age
 method S toS() = "["++this.name().toS()++", "+this.age().toS()++"]"
}
```

Phase 4: composition

The last operation in the top level method, is composing our generated `toS()` behaviour with the original code (containing also `sayHi`) in order to obtain the final result.

We wish to stress how this example code could easily be adapted for `equals(that)` and a plethora of other field dependent operations. With slightly more adaptation it could generate any pattern based on method shape/names in any code source. This is clearly supporting all the expression power of MorphJ [**?**].

It could be possible to "abstract" over this code, using the template method pattern so that to write generators for, let's say, `equals(that)` and `toS()` the programmer may reuse the logic of `fold(that,acc)` and the top level method. Here we preferred to show the concrete code of the Decorator `Stringable` for the sake of a more direct example.

To conclude, in this section we have a shown how a combination of trait multiple inheritance, redirect/generics and rename can be used as a starting point to synthesize arbitrary behaviour for a code literal using an inductive mindset. Trait multiple inheritance, redirectand rename are high level declarative class composition and adaptation operators, and the obtained decorator also has this declarative 'feel'.

4 Case study: evaluation in the 42 language

42 is an ambitious language, aiming to allow thousands of libraries to work together in a safe and maintainable manner. To this aim the actual 42 syntax is a little different from the one presented here, that is focused on making IC easier to grasp.

For example, operations are properly packaged under libraries: indeed most operators are accessed under `Refactor`, as in `Refactor.rename(...)`. The super call mechanism is different: instead of user specified names, override uses conventional names: for `toS()` they would be `#1toS()` and `#2toS()`.

Another very important difference is that 42 sightly relaxes the requirement that all library literals are well typed whenever they are involved in execution. This carefully designed relaxation does not weaken the formal properties of the language, and provides two advantages: it facilitates the use of mutually recursive type declarations, as very common in OO languages, and it allows to omit some abstract method declarations.

The core primitive operators include symmetric sum of code, redirect and rename. Programmers rarely use such operators directly, favouring derived and more high level operators, similar to `Override` as presented in this paper.

42 is **designed around meta-programming**, where IC is the **only** way for code reuse and code adaptation.

Large practical experiments give us confidence that IC can be successfully and conveniently used to replace both QQ and conventional core reuse features, as `extends` or generics. An (anonymized) 42 tutorial can be found at l42.is/tutorial.xhtml, while the GitHub project (not anonymized) can be reached at github.com/ElvisResearchGroup/L42. At URL github.com/ElvisResearchGroup/L42/tree/master/Tests/src you can find about 10k lines of 42 code.

To show that we can replace conventional code reuse, we have implemented a minimal 'collections' library (adamsTowel01/libProject/Collections/)[1] and a large 'introspection' library (adamsTowel01/libProject/Location), allowing to examine library literals. The collection library uses IC instead of generics, while the introspection library has many classes reusing common code, and this reuse is obtained using IC instead of `extends`. To show that we can replace QQ, we developed a (quite compact thanks to IC) 'units of measure' library (adamsTowel02/libProject/Units) , and a sophisticated `Data` decorator (adamsTowel02/libProject/Data) that adds equality, `toS()`, run time invariant checking and other features to library literals.

Finally, to show that IC can scale to work with large units of code, we have implemented a non-trivial 'library loading' library (adamsTowel02/libProject/Load and adamsTowel02/libProject/DeployLibrary), that automatically tweaks libraries to use different implementations for their dependencies; this allows, for example, to change what kind of numbers and strings are internally used by a 42 third party library. This is obtained by smart usage of the `Redirect` and `Rename` operators.

5 Conclusion

Quasi Quotation offers the maximum possible expressive power, since it can generate any possible AST. We argue that our approach can generate any behaviour, but not any AST. For example, we have no direct control on the way local variables, private methods and in-lining are handled. We believe this is a good thing: fine control of the

[1] For this and other links the full url looks like github.com/ElvisResearchGroup/L42/tree/master/Tests/src/adamsTowel01/libProject/Collections

structure of expressions requires the (meta-)programmer to understand and handle scope, scope-extrusion, variable hiding and similar representation related issues.

Practical experience shows that by using nested classes our approach allows generation of large chunks of code, by generating many interconnected classes at the same time. While, QQ is mostly used to generate single method bodies/functions.

We show that IC can be used to design highly abstract code generation, with a clear declarative feel, while QQ requires (by design) to keep in mind the concrete shape of the generated code.

We also speculate how conventional OO verification techniques can be used to verify code generated with IC. To the best of our knowledge, there is no equivalent verification for QQ.

Due to space limitations we cannot include an extensive related work section, but we tried to discuss other approaches during the exposition. The interested reader may refer to the good survey by Smaragdakis [**?**].

Others proposed to use traits and code composition to perform meta-programming [**?**].

The very popular Scala library LMS (scala-lms.github.io), in addition to conventional QQ offers more abstract techniques for AST rewriting and manipulation/simplification, to go towards the kind of abstraction offered by IC; thus recognizing QQ is, at least sometimes, too low level.

We wonder if Ur [**?**] could be extended to represent our `inductive()` concept: Ur is a QQ system focused on guaranteeing generation of well-typed records, and it may be possible to extend it to records with abstract members.