# Metaprogramming Statically verified code

Hrshikesh Arora (arorahrsh@myvuw.ac.nz), Marco Servetto (marco.servetto@ecs.vuw.ac.nz)

Quasi Quotation [2, 6] 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 because the process is fundamentally low-level, imperative and bug prone. In this paper, we present Iterative Composition [5] as an **effective alternative** to Quasi Quotation. Iterative Composition describes a code composition algebra over established code reuse techniques, similar to trait composition [4] 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 L42 language.

Quasi quotation (QQ) can be supported by two kinds of special parenthesis as a syntactic sugar to manipulate Abstract Syntax Trees (ASTs).Here we use `[| |]` and `$( )` as Template Haskell [6]. 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. An '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_b=λ x.x*x*x*x*x*x*x*1;`. On most machines, `power7` runs faster than a naive recursive version. Metaprogramming applications include more than just speed boosts, but we start with this example because it is very popular and simple. However, it is unclear how to statically verify code generate with IC. We now show how to rewrite our `pow` example in our proposed approach, while annotating the code with pre and post conditions, as it happens in JML [1].

```
Pow={
  static method Library base()={
    /*@ensures exp()=0*/ method Num exp()=0
    /*@ensures pow(x)=x^{this.exp()}*/ method Num pow(Num x)= 1 }
  static method Library inductive()={
    /*@ensures @result=1+this.superExp()*/ method Num exp()=1+this.superExp()
    method Num superExp()
    /*@ensures @result=x^{this.exp()}*/ method Num pow(Num x)= x*this.superPow(x)
    /*@ensures pow(x)=x^{this.superExp()}*/ method Num superPow(Num x)}
  //@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())}
    return res } }
```

Our approach clearly is more verbose, but is exposing all the hidden complexity of the QQ code. In short, `base()` and `inductive()` are methods returning constant statically verified code. Static verification is a very computationally intensive step, and our approach verifies `base()` and `inductive()` once and for all during compilation. **`Override`** inside `generate(y)` behaves like mixin composition, and the syntax `[exp<-superExp()]` encodes the conventional super calls. Thus, we iteratively create a chain of inheritance where the `base` code is extended with `inductive` `y` times, generating the behaviour we need. Crucially, the static verifier needs to only statically verify the code of `base()` and `inductive()`; the `generate(y)` method will then just do an efficient contract matching. Thus, at every iteration of the for-loop, `res` will contain statically verified code; however, the contract of such code it is not statically know. Finally, as required by the `ensuresRV` clause, at the end of code generation, the system will check that the contract of the generate code is the expected one.

Concluding, static verification of metaprogramming is a near unexplored area and we are trying attack the problem by reusing conventional object oriented static verification by relying on composition operators similar to **`extends`** and generics to generate behaviour.

## References

[1] Bart Jacobs and Erik Poll. A Logic for the Java Modeling Language JML. In Heinrich Hussmann, editor, *Fundamental Approaches to Software Engineering*, pages 284–299, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[2] Eugenio Moggi, Walid Taha, Z El-Abidine Benaissa, and Tim Sheard. An idealized MetaML: Simpler, and more expressive. In *ESOP*, volume 1576, pages 193–207. Springer, 1999.

[3] Kent M Pitman. Special forms in Lisp. In *Proceedings of the 1980 ACM conference on LISP and functional programming*, pages 179–187. ACM, 1980.

[4] Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew P Black. Traits: Composable units of behaviour. In *ECOOP*, volume 3, pages 248–274. Springer, 2003.

[5] Marco Servetto and Elena Zucca. A meta-circular language for active libraries. *Science of Computer Programming*, 95:219–253, 2014.

[6] Tim Sheard and Simon Peyton Jones. Template meta-programming for Haskell. In *Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*, pages 1–16. ACM, 2002.