

Iteratively Composing Statically Verified Traits

Isaac Oscar Gariano

Marco Servetto

Alex Potanin

Hrshikesh Arora

School of Engineering and Computer Science
Victoria University of Wellington
Wellington, New Zealand

isaac@ecs.vuw.ac.nz

marco.servetto@ecs.vuw.ac.nz

alex@ecs.vuw.ac.nz

arorahrsh@myvw.ac.nz

Object oriented languages supporting static verification (SV) usually extend the syntax for method declarations to support *contracts* in the form of pre and post-conditions [6]. Correctness is defined only for code annotated with such contracts.

We say that a method is *correct*, if whenever its precondition holds on entry, the precondition of every method it calls also holds, and the postcondition of the method holds on return. Automated SV typically works by asking an automated theorem prover to verify that each method is correct individually, by assuming the correctness of every other method [and the method's own precondition] [2]. This process can be very slow and produce unexpected results, since SV is undecidable, correct code may not necessarily pass a SV. Making matters worse, the way an SV works is not easily understood by programmers, thus it is non trivial for them to transform or construct code that will be SVed.

Metaprogramming is often used to programmatically generate faster specialised code when some parameters are known in advance, this is particularly useful where the specialisation mechanism is too complicated for a generic compiler to automatically derive [7] We could use metaprogramming to generate code together with contracts, and then once the metaprogramming has been run, SV the resulting code, thus ensuring it's correctness. However, the resulting code could be much larger than the input to the metaprogramming, and so could take a long time to SV. Even if the generated code was produce by using straightforward transformations and compositions over the input code, a SV might not verify it's correctness. The aim of our work is to ensure that result of metaprogramming is correct by construction, and thus not require SV, as well as for it to be predictable weather such constructions will succeed.

Here we use the disciplined form of metaprogramming introduced by Servetto & Zucca [10], which is based on trait composition and adaptation [9]. Here a **Trait** is a unit of code: a set of method declarations with pre/post-conditions. [Isaac: REV2 wants us to properly define what a triat is] They are well-typed and correct. **Traits** directly written in the source code, we extend prior work, by also requiring that they are *proven correct by SV*. We extend the checking performed by composition and adaptation of **Traits** to also check that contracts preserved, ensuring the correctness of the result. [Isaac: I don't think my rewording makes our contribution explicit enough..] In our style, metaprogramming does not generate code directly, rather code is only generated by composing and adapting traits, thus by induction, the result of metaprogramming is also correct. [However generated code may not be able to pass SV, since theorem provers are not complete.]

SV handles **extends** and **implements** by verifying that every time a method is implemented/overriden, the Liskov substitution principle [5] is satisfied: the implementation and any additional contracts must satisfy the contract of the method that is being implemented/overriden. In this way, there is no need to re-verify inherited code in the context of the derived class, nor verify each call to a class's method based on each possible implementation. This is easily adapted to handle trait composition, which simply provides another way to implement an **abstract** method. When traits are composed, it is sufficient to match the contracts of the few composed methods to ensure the whole result is correct.

In our examples we will use the notation `@requires(predicate)` to specify a precondition, and `@ensures(predicate)` to specify a postcondition; where *predicate* is a boolean expression in terms of the parameters of the method (including `this`), and for the `@ensures` case, the `result` of the method. Suppose we want to implement an efficient exponentiation function, we could use recursion and the common technique of ‘repeated squaring’:

```

1  @requires(exp > 0)
2  @ensures(result == x**exp) // Here x**y means x to the power of y
3  Int pow(Int x, Int exp) {
4    if (exp == 1) return x;
5    if (exp % 2 == 0) return pow(x*x, exp/2); // exp is even
6    return x*pow(x, exp-1); } // exp is odd

```

If the exponent is known at compile time, unfolding the recursion produces even more efficient code:

```

7  @ensures(result == x**7) Int pow7(Int x) {
8    Int x2 = x*x; // x**2
9    Int x4 = x2*x2; // x**4
10   return x*x2*x4; } // Since 7 = 1 + 2 + 4

```

Now we show how we can use the technique of iterative trait composition, together with our contract matching, to write a meta-programming, that given an exponent, produces code like the above. Iterative trait composition employing a technique called *compile-time execution* [11], in which a meta-program is executed at compile time, and the result (a `Trait`) is treated as if it was directly written by the programmer.

```

11 Trait base=class { //induction base case: pow(x) == x**1
12   @ensures(result>0) Int exp(){return 1;}
13   @ensures(result==x**exp()) Int pow(Int x){return x;}
14 }
15 Trait even=class { //if _pow(x)== x**_exp(), pow(x) == x**(2*_exp())
16   @ensures(result>0) Int _exp();
17   @ensures(result==2*_exp()) Int exp(){return 2*_exp();}
18   @ensures(result==x**_exp()) Int _pow(Int x);
19   @ensures(result==x**exp()) Int pow(Int x){return _pow(x*x);}
20 }
21 Trait odd=class { //if _pow(x)== x**_exp(), pow(x) == x**(1+_exp())
22   @ensures(result>0) Int _exp();
23   @ensures(result==1+_exp()) Int exp(){return 1+_exp();}
24   @ensures(result==x**_exp()) Int _pow(Int x);
25   @ensures(result==x**exp()) Int pow(Int x){return x*_pow(x);}
26 }
27 //‘compose’ performs a step of iterative composition
28 Trait compose(Trait current, Trait next){
29   current = current[rename exp->_exp, pow->_pow];
30   return (current+next)[hide _exp, _pow];}
31 @requires(exp>0) //the entry point for our metaprogramming
32 Trait generate(Int exp) {
33   if (exp==1) return base;
34   if (exp%2==0) return compose(generate(exp/2), even);

```

```

35     return compose(generate(exp-1), odd);
36 };
37 class Pow7: generate(7) //generate(7) is executed at compile time
38 //the body of class Pow7 is the result of generate(7)
39 /*example usage:*/new Pow7().pow(3)==2187//Compute 3**7

```

The traits `base`, `even`, and `odd` are the basic building blocks we will use to compute our result. They will be compiled, typechecked and SVed before the method `generate(exp)` can run. As you can see in line 37, a class body can be an expression in the language itself. At compile time such an expression will be run and the resulting `Trait` will be used as the body of the class. For example, we could write `class Pow1: base;` this would generate a class such that `new Pow1().pow(x)==x**1`. The other two traits have abstract methods; implementations for `_pow(x)` and `_exp()` must be provided. However, given the contract of `pow(x)`, and the fact that `even` and `odd` have both been SVed, if we supply method bodies respecting these contracts, we will get *correct* code, without the need for further SV. Many works in literature allow adapting traits by renaming or hiding methods[10, 8, 4]. Hiding a method may also trigger inlining if the method body is simple enough or used only once. Since all occurrences of names are consistently renamed, **renaming and hiding preserve code correctness**.

The `compose` method starts by renaming the `exp` and `pow` methods of `current` so that they satisfy the contracts in `next` (which will be `even` or `odd`). The `+` operator is the main way to compose traits [9, 3]. The result of `+` will contain all the methods from both operands.

Crucially, it is possible to sum traits where a method is declared in both operands; in this case at least one of the two competing methods needs to be abstract, and the signatures of the two competing methods need to be *compatible*. To make sure that the traditional `+` operator also handles contracts, we need to require that the contract annotations of the two competing methods are *compatible*. For the sake of our example, we can just require them to be syntactically identical. Relaxing this constraint is an important future work. Thanks to this constraint **the sum operator also preserves code correctness**. [There are many variations of the `+` operator, in particular, we could easily extend our contract matching to work with an nary operator.](#)

The sum is executed when the method `compose`¹ runs, if the matched contracts are not identical an exception will be raised. A leaked exception during compile-time metaprogramming would become a compile-time error. Our approach is very similar to [10], and does not guarantee the success of the code generation process, rather it guarantees that if it succeeds, correct code is generated.

Finally the `_pow(x)` and `_exp()` method are hidden, so that the structural shape of the result is the same as `base`'s. As you can see, `Traits` are first class values and can be manipulated with a set of primitive operators that preserve code correctness and well-typedness. In this way, by inductive reasoning, we can start from the base case and then recursively compose `even` and `odd` until we get the desired code. Note how the code of `generate(exp)` follows the same scheme of the code of `pow(x, exp)` in line 1.

To understand our example better, imagine executing the code of `generate(7)` while keeping `compose` in symbolic form. We would get the following (where `c` is short for `compose`):

```

generate(7) == c(generate(6), odd) == ...
            == c(c(c(c(base, even), odd), even), odd)

```

As `base` represents `pow1(x)`; `c(base, even)` represents `pow2(x)`. Then `c(/*pow2(x)*/, odd)` represents `pow3(x)`, `c(/*pow3(x)*/, even)` represents `pow6(x)`, and finally, `c(/*pow6(x)*/, odd)` represents `pow7(x)`. The code of each `_pow` method is only executed once for each top-level `pow` call, so

¹a generic implementation of this method that renames and hides conflicting methods has been implemented L42 [1]

the `hide` operator can inline them. Thus, the result could be identical to the manually optimized code in line 7.

Note that while our approach guarantees that the resulting code follows its own contracts, it does not statically ensure what contracts it would have.² We are investigating how to perform an additional verification check on the result of metaprogramming. For example, the following code:

```
@ensures(new Pow7().exp()==7&&Pow7.pow.ensures=="result==x**exp()")
class Pow7: generate(7)
```

may require the static verifier to check that the execution of `new Pow7().exp()` will deterministically reduce to 7, and that the `ensures` clause of `Pow7.pow` is syntactically equivalent to `result==x**exp()`. Note how this final step of static verification does not need to re-verify the body of `Pow7.pow` and only needs to do a coarse grained determinism check on the implementation of `Pow7.exp()`, before symbolically executing it.

In conclusion, static verification of metaprogramming is an exciting new area of research; we are attacking the problem by reusing conventional object oriented static verification techniques coupled with trait composition, extended to also check contract compatibility. A crucial design decision is that code performing metaprogramming does not need to be SVD to produce code annotated with the desired contracts; it would be sufficient to apply some type of runtime verification during compile-time execution.

References

- [1] 42 - *The definitive answer to design, code and everything*. <http://142.is/>.
- [2] Mike Barnett, K Rustan M Leino & Wolfram Schulte (2004): *The Spec# programming system: An overview*. In: *International Workshop on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, Springer, pp. 49–69.
- [3] Giovanni Lagorio, Marco Servetto & Elena Zucca (2009): *Featherweight Jigsaw: A Minimal Core Calculus for Modular Composition of Classes*. In Sophia Drossopoulou, editor: *ECOOP 2009 - Object-Oriented Programming, 23rd European Conference, Genoa, Italy, July 6-10, 2009. Proceedings, Lecture Notes in Computer Science* 5653, Springer, pp. 244–268, doi:10.1007/978-3-642-03013-0_12.
- [4] Luigi Liquori & Arnaud Spiwack (2008): *FeatherTrait: A modest extension of Featherweight Java*. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 30(2), p. 11.
- [5] Barbara H. Liskov & Jeannette M. Wing (1994): *A Behavioral Notion of Subtyping*. *ACM Trans. Program. Lang. Syst.* 16(6), pp. 1811–1841, doi:10.1145/197320.197383. Available at <http://doi.acm.org/10.1145/197320.197383>.
- [6] Bertrand Meyer (1988): *Object-Oriented Software Construction*, 1st edition. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.
- [7] Georg Ofenbeck, Tiark Rompf & Markus Püschel (2017): *Staging for Generic Programming in Space and Time*. In: *Proceedings of the 16th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, GPCE 2017, ACM, New York, NY, USA*, pp. 15–28, doi:10.1145/3136040.3136060. Available at <http://doi.acm.org/10.1145/3136040.3136060>.
- [8] John Reppy & Aaron Turon (2007): *Metaprogramming with traits*. In: *ECOOP*, Springer, pp. 373–398.
- [9] Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz & Andrew P Black (2003): *Traits: Composable units of behaviour*. In: *ECOOP*, 3, Springer, pp. 248–274.
- [10] Marco Servetto & Elena Zucca (2014): *A meta-circular language for active libraries*. *Science of Computer Programming* 95, pp. 219–253.

²REV2: This paragraph casts doubt on the significance of what has been presented.

- [11] Tim Sheard & Simon Peyton Jones (2002): *Template meta-programming for Haskell*. In: *Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*, Haskell '02, ACM, pp. 1–16, doi:10.1145/581690.581691.