

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@myvuw.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 [3]. Correctness is defined only for code annotated with contracts. We say that such code is *correct* if before any method is called its pre-condition holds, and before a method returns, its post-condition holds. Automated SV typically works by asking an automated theorem prover to verify each method is correct individually, by assuming the correctness of every other method and the methods own pre-condition [1]. This process can be very slow, and unpredictable; that is, SV is a process that can be applied to annotated code to check if such code is correct, but there may be correct code that does not pass static verification on a certain theorem prover.

Metaprogramming is often used to programmatically generate faster specialised code when some parameters are known in advance. To use metaprogramming and SV together, we could generate code containing contracts, and such code could be checked after metaprogramming has been completed. SV could then be applied to the code resulting from the metaprogramming to ensure it is correct. However this could be very time consuming, since it would require verifying all the generated code from scratch. Even worst, since SV is unpredictable, there would be no guarantee that the result of a given metaprogram will be verifiable, even if its result is correct by construction.

We propose a disciplined form of metaprogramming based on trait composition and adaptation [5, 6]. Here a **Trait** is a class body which is well-typed and correct (according to the contracts of its methods). **Traits** directly written in the source code are proven correct by SV. The result of composing and adapting **Traits** is also correct and well-typed. Metaprogramming can not generate code directly, instead code is generated by composing and adapting traits, thus generated code is guaranteed correct. However the code resulting from metaprogramming may not be able to pass SV, since theorem provers are usually not complete.

SV handles **extends** and **implements** by verifying that every time a method is implemented/overridden, the Liskov substitution principle is preserved. In this way, there is no need to re-verify the inherited code in the context of the derived class. This idea 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 check that the contracts of a few methods match to ensure the whole result is correct. [Isaac: which methods?]

In our examples we will use the notation **@requires**(*predicate*) to specify a precondition, and **@ensures**(*predicate*) to specify a postcondition; predicates are boolean expressions in terms of the parameters of the method (including **this**), and for the **@ensures** case, the **result** of the method call. 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, `exp`, 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 generate code like the above, employing a technique called compile-time-execution [7]:

```

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 // whats up with '
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

```

[Isaac: `class Pow7 = generate(7)` looks like you implicitly converting a `Trait` to a `class`, this is why I used a `:` to distinguish class creation from deaclearing a static variable of type `class`] The three traits `base`, `even`, and `odd` are the basic building blocks we will use to compute our result. They will be compiled, typechecked and SV 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 SV, if we supply method bodies respecting these contracts, we will get *correct* code, without the need of further SV. [Isaac: Using SV everywhere, in different ways (e.g. to mean static-verification, statically-verify and statically-verified), is confusing, perhaps you could just say ‘verify’ or ‘verified’] Many works in literature allow adapting traits by renaming or hiding methods[6, 4, 2]. Hiding a method may also trigger inlining if the method body is simple enough or used only once. Since all occurrences of names are handled coherently, **renaming and hiding preserve code correctness**. [Isaac: What does ‘coherently’ mean here?] [Isaac: As I have told you a hundred times, stop bolding chunks of sentences! If you need to emphasise something you should italicise things (but even then, only italicise one or two words)]

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. The result of `+` will contain all the methods from both operands. Crucially, it is possible to sum traits where a method is declared on both sides; in this case at least one of the two competing methods needs to be abstract and the signature and contracts annotations of two competing methods need to be *compatible*. For the sake of our example, we can just require them to be syntactically identical. Relaxing this constraint is an important future work. [Isaac: Your not allowing weakening or strengthening? Interesting, I would put a sentence or two saying this can be done, but not explain how you actually check it (since none of this is implemented anyway)] Thanks to this constraint **the sum operator also preserves code correctness**.

The sum is executed when the method compose runs, if the matched contracts are not identical an exception will be raised. A leaked exception during metaprogramming would become a compile time error. [Isaac: Is the mechanism of error handling important here? Just say an ‘error’ will occur when it is executed or at compile time (meta time is a better term here, as it isn’t when the compose method is compiled, but when it is run!)] Thus, our approach does not guarantee the success of the code generation process, but it guarantees that only 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, method composition is the core of our approach: traits are first class values and can be manipulated with a set of primitive operators that preserve code correctness and well-typing.

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:

```
generate(7) == compose(generate(6),odd) == ...
compose(compose(base,even),odd),even),odd)
```

As we discussed, `base ~ pow1(x)`: it offers a method `pow(x)` computing `x**1`.

By applying compose we can conclude that [Isaac: What in the world does this sentence mean? Also what does mean?] `compose(/*pow1(x)*/,even) ~ pow2(x)`, `compose(/*pow2(x)*/,odd) ~ pow3(x)`, `compose(/*pow3(x)*/,even) ~ pow6(x)` and finally `compose(/*pow6(x)*/,odd) ~ pow7(x)`. The code of each `_pow` is only executed once for each top-level `pow` call, so the `hide` operator can simply inline it. This means that the generated code could be identical to the manually optimized version 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. As future work, 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()")
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 statically verified to produce correct code or code annotated with the right contracts; it would be sufficient to run some type of runtime verification during compile-time execution.

References

- [1] 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.
- [2] Luigi Liquori & Arnaud Spiwack (2008): *FeatherTrait: A modest extension of Featherweight Java*. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 30(2), p. 11.
- [3] Bertrand Meyer (1988): *Object-Oriented Software Construction*, 1st edition. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.
- [4] John Reppy & Aaron Turon (2007): *Metaprogramming with traits*. In: *ECOOP*, Springer, pp. 373–398.
- [5] Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz & Andrew P Black (2003): *Traits: Composable units of behaviour*. In: *ECOOP*, 3, Springer, pp. 248–274.
- [6] Marco Servetto & Elena Zucca (2014): *A meta-circular language for active libraries*. *Science of Computer Programming* 95, pp. 219–253.
- [7] 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.