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 [4]. Correctness is defined only for code annotated with such contracts. We say that such code is correct if, assuming that its precondition holds before the method is called, its post-condition holds when the method returns. 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 [1]. This process can be very slow, brittle and unpredictable[1] as there may be correct code that does not pass SV on a certain theorem prover. Moreover, a contract that was verified using a certain version of a theorem prover may be unverifiable under an updated version of the same 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, if the same metaprogramming process is used to create many specialized version of code, this could be very time consuming: it would require verifying all the generated specialized code from scratch. Even worse, since SV is undecidable and brittle, there would be no guarantee that the result of a given metaprogram will be verifiable, even if its result is correct by construction.[2]

We extend the disciplined form of metaprogramming of Servetto & Zucca [7], which is based on trait composition and adaptation [6]. Here a `Trait` is a unit of code: a set of method declarations with pre/post-conditions. They are well-typed and correct. `Trait`s directly written in the source code are proven correct by SV. The composition and adaptation of `Trait`s is carefully defined to preserve correctness. Metaprogramming cannot generate code directly, code is only generated by composing and adapting traits, thus generated code 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[3] is satisfied. In this way, there is no need to re-verify the inherited code in the context of the derived class. 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.

[1] **REV1: What do you mean by "SV is unpredictable"? I would be careful here because unpredictability may mean nondeterminism. That is, you verify a program once, and get "Correct"; then you verify it the second time, and get "Buggy".**

[2] **REV2: This paragraph seems to boil down to saying that it is preferable to verify the meta-program directly rather than the code resulting from running the meta-program. This statement requires some justification rather than that the latter would be "time consuming". In some cases verification of the generated code could be much easier. The running example seems to be a case in point.**

[3] **REV2: Please explain or cite a reference**

Submitted to:
VPT 2019

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 generate code like the above, employing a technique called *compile-time execution* [8][4]:

```
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
```

[4] **REV3: But actually you use compile-time symbolic execution rather than compile-time execution, and even, speaking on technical details, program specialization method known as partial evaluation. Is it correct? But no reference to the corresponding papers is given. There is a huge literature relating to different methods of automated program specialization. Do you aware about? A number of such references should be included in the short reference list ending your extended abstract.**

There are also many works on verification by program specialization. Maybe they are useful for improving your methods. Just please pay attention to this subject.

```
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[7, 5, 3]. 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 [6, 2]. The result of + will contain all the methods from both operands.[5]

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**.

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 compile-time metaprogramming would become a compile-time error. Our approach is very similar to [7], 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, `Trait`s 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)
```

[5] **REV3: Please take into account that the + operator is associative. Consequently, program specialization and program verification methods aiming at effective analysis and transformation of the corresponding expressions with the operator have to be used and developed.**

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 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.[6] 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 SVed 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] 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] 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.

[3] Luigi Liquori & Arnaud Spiwack (2008): *FeatherTrait: A modest extension of Featherweight Java*. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 30(2), p. 11.

[4] Bertrand Meyer (1988): *Object-Oriented Software Construction*, 1st edition. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

[5] John Reppy & Aaron Turon (2007): *Metaprogramming with traits*. In: *ECOOP*, Springer, pp. 373–398.

[6] Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz & Andrew P Black (2003): *Traits: Composable units of behaviour*. In: *ECOOP*, 3, Springer, pp. 248–274.

[7] Marco Servetto & Elena Zucca (2014): *A meta-circular language for active libraries*. *Science of Computer Programming* 95, pp. 219–253.

[8] 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.

[6] **REV2: This paragraph casts doubt on the significance of what has been presented.**