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 usually extend the syntax for method declarations to support *contracts* in the form of pre and post-conditions [5]. 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 directly invoked method holds, and the postcondition of the method holds when the method returns. Automated static verification typically works by asking an automated theorem prover to verify that each method is correct individually, by assuming the correctness of every other method [1]. This process can be very slow and can produce unexpected results: since static verification is undecidable correct code may not pass a particular static verifier. Many static verification approaches are not resilient to standard refactoring techniques like method inlining. Sometimes static verification even times out, making the behaviour even more sensitive to such refactoring techniques.

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 [6] We could use metaprogramming to generate code together with contracts, and then once the metaprogramming has been run, statically verify the resulting code. However, the resulting code could be much larger than the input to the metaprogramming, and so it could take a long time to statically verify. Moreover, one of the many goals of metaprogramming is to make it easier to generate specialised versions of the same code. The aim of our work is to statically verify only the original source code itself, and not the code produced by metaprogramming; instead, we ensure that the result of metaprogramming is correct by construction.

Here we use the disciplined form of metaprogramming introduced by Servetto & Zucca [9], which is based on trait composition and adaptation [8]. Here a `Trait` is a unit of code: a set of method declarations. Such methods can be abstract and be mutually recursive by using the implicit parameter `this`.

As in [9] we require that all the traits are well-typed before they are used. we extend this by allowing methods to be annotated with pre/post-conditions, and ensuring that traits are correct in terms of such contracts. `Trait`s directly written in the source code are statically verified, while traits resulting from metaprogramming are ensured correct by only providing trait operations that preserve correctness. In particular, we **only** need to extend the checking performed by the traditional trait composition (+) operator, to also check the compatibility of contracts.

Our metaprogramming approach does not allow generating code from scratch (such as by directly generating ASTs), rather the language provides a specific set of primitive composition and adaptation operators which preserve correctness. Thus the result of metaprogramming is guaranteed to be well typed and correct. Note that generated code may not be able to pass a particular static verifier.

Static verification usually handles `extends` and `implements` by verifying that every time a method is implemented/overriden, the Liskov substitution principle [4] is satisfied by checking that the contracts

Submitted to:
VPT 2019

of the method in the derived class implies the contract of any corresponding methods in its base classes. In this way, there is no need to re-verify inherited code in the context of the derived class. This concept 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':

```
@requires(exp > 0)
@ensures(result == x**exp) // Here x**y means x to the power of y
Int pow(Int x, Int exp) {
  if (exp == 1) return x;
  if (exp %2 == 0) return pow(x*x, exp/2); // exp is even
  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:

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

Now we show how the technique of *Iterative Composition* (introduced in [9] and enriched by the contract compatibility check we propose performing in trait composition) can be used to write a metaprogram that given an exponent, produces code like the above. Iterative Composition is a metacircular metaprogramming technique relying on *compile-time execution* (a form of execution also used by [10]), in our context this means that arbitrary expressions can be used as the right hand side of a class declaration; during compilation such expressions will be evaluated to produce a `Trait`, which provides the body of the class. In this way metaprograms can be represented as otherwise normal functions/methods that return a `Trait`, without requiring the use of any additional 'meta language'.

```
Trait base=class {//induction base case: pow(x) == x**1
  @ensures(result>0) Int exp(){return 1;}
  @ensures(result==x**exp()) Int pow(Int x){return x;}
  }
Trait even=class {//if _pow(x)== x**_exp(), pow(x) == x**(2*_exp())
  @ensures(result>0) Int _exp();
  @ensures(result==2*_exp()) Int exp(){return 2*_exp();}
  @ensures(result==x**_exp()) Int _pow(Int x);
  @ensures(result==x**exp()) Int pow(Int x){return _pow(x*x);}
}
Trait odd=class {//if _pow(x)== x**_exp(), pow(x) == x**(1+_exp())
  @ensures(result>0) Int _exp();
  @ensures(result==1+_exp()) Int exp(){return 1+_exp();}
  @ensures(result==x**_exp()) Int _pow(Int x);
  @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 statically verified 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 statically verified, if we supply method bodies respecting these contracts, we will get *correct* code, without the need for further static verification. Many works in literature allow adapting traits by renaming or hiding methods[9, 7, 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 [8, 2]. 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**.

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 [9], 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)
```

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.

Our approach, as presented in this extended abstract, only guarantees that code resulting from metaprogramming follows its own contracts, it does not statically ensure what those contracts may be. As future work, we are investigating how the resulting contracts can be ensured to have a particular meaning or form. To do so, we need to allow assertions on the contracts of `Trait`s to be used within pre/post conditions. For example we could allow post conditions like
`@ensures(result.`*methName*`.ensures ==` *predicate*`)`
to mean that the resulting `Trait` has a method called *methName*, whose `@ensures` clause is syntactically identical to *predicate*; whilst
`@ensures(result.`*methName*`.ensures ==>` *predicate*`)`
would use a static verifier to ensure that *methName*'s `@ensures` clause logically implies *predicate*. With these two features we could annotate the method `generate` in line 32 above as:

```
42 @requires(exp>0)
43 @ensures(result.exp().ensures ==> (result==exp))
44 @ensures(result.pow(x).ensures == (result==x**exp()))
45 Trait generate(Int exp) {...}
```

In this way, we could statically verify the `generate` method, however we fear such verification will be too complex or impractical. We could instead automatically check the above postconditions after each call to `generate`. If `generate` is used to define a class (such as `Pow7` above), we will guarantee that such class has the correct contracts, before it is used. Thus there is no need to ensure the correctness of the metaprogram itself: such runtime checks are sufficient to ensure that after compilation, the code produced by metaprogramming has its expected behaviour.

In conclusion, by leveraging over conventional OO static verification techniques, we have extended the Iterative Composition form of metaprogramming with a simple contract compatibility check, to statically ensure the correctness of code produced by such metaprogramming. In particular, our approach does not require static verification of the result of metaprogramming, but only requires verification of code present directly in source code.

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] 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`.

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

[6] 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`.

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

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

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

[10] 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.