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, marco.servetto, alex, arorahrsh}@myvuw.ac.nz

Static verification relying on an automated theorem prover can be very slow and brittle: since static verification is undecidable, correct code may not pass a particular static verifier. In this work we use metaprogramming to generate code that is correct by construction. A theorem prover is used only to verify initial "traits": units of code that can be used to compose bigger programs.

In our work, meta-programming is done by trait composition, which starting from correct code, is guaranteed to produce correct code. We do this by extending conventional traits with pre- and post-conditions for the methods; we also extend the traditional trait composition (+) operator to check the compatibility of contracts. In this way, there is no need to re-verify the produced code.

We show how our approach can be applied to the standard "power" function example, where metaprogramming generates optimised, and correct, versions when the exponent is known in advance.

1 Introduction

With this short paper we contribute to research on safe metaprogramming, showing how combining pre/post conditions, trait composition and metaprogramming it is possible to create metaprograms that generate code which is correct by construction. That is: only the original source code itself needs to be verified (for example by a theorem prover), and not the code produced by metaprogramming. This is important since the original code is often orders of magnitude smaller than the generated code. We start by providing some background on those three research areas: pre/post conditions, traits, and metaprogramming.

Pre/post conditions: object-oriented (OO) 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.

Traits: originally introduced in Smalltalk by Scharli [8], traits are units of code reuse. They were created as a simpler way of performing multiple inheritance without the usual complexity. Traits are just a set of method declarations. Such methods can be abstract and mutually recursive by using the implicit `this` parameter. Traits are different from Java-style abstract classes as they are only for reuse: trait names do *not* define types. Trait composition is seen as a form of flattening: after the composition,

A.P. Lisitsa and A.P. Nemytykh (Eds.): VPT 2019
EPTCS 299, 2019, pp. 49–55, doi:10.4204/EPTCS.299.7

the resulting code contains copies of the adapted methods from the original traits, but do not reference the original traits directly. Since there is no trace of the point of origin of the code, `super` calls are not directly available and need to be somehow emulated. Traits can be combined using many different composition operators, not just *extends*. In this work we will rely on the traditional composition operators +(plus), `rename` and `hide`.

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*. In this way, + has an expressive power similar to multiple inheritance.

Rename and *hide* adapt a single trait by renaming a method or by making a method private. Many works in the 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.

Consider the following example code, where we use those 3 operators:

```
Trait a=class{Int hello(){return 1;}}
Trait b=class{
  abstract Int hello();
  String world(){return "["+this.hello()+"]";}}
Trait c=(a+b)[hide hello()][rename world()->hello()]
```

After flattening we would get the following result; where `c` now contains a single method called `world()`, with the body of the method originally called `hello()` declared in trait `b`. This body contains the inlined version of method `a.world()` that has been hidden. Note how the order of operations is important.

```
Trait a=/*as before*/
Trait b=/*as before*/
Trait c=class {String world(){return "["+1+"]";}}
```

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]. A *metaprogram* is a program, method or function that produces code. Depending on the kind of metaprogramming, such code can be directly executable or can be just an abstract representation of behaviour. Metaprogramming is called *metacircular* when the language used to write the metaprogram is the same language of the generated code. Metaprogramming can happen at run time, or at compile time. In the latter case, the produced code can be needed to typecheck and compile the rest of the code in the program. In this paper we will rely on Iterative Composition: a metacircular metaprogramming technique relying on *compile-time execution* (a form of execution also used by [10]). This disciplined form of metaprogramming introduced by Servetto and Zucca [9], is based on the trait composition operators described before, but lifted at the expression level. This means that arbitrary expressions can be used as the right hand side of trait and class declarations; 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 ‘metalanguage’.

2 Combining Metaprogramming and Static Verification

A naïve way to combine metaprogramming and static verification could be to 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 many 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.

We extend [9] by allowing methods to be annotated with pre/post-conditions. In addition to requiring that all the traits are well-typed before they are used (as in [9]) we also require that traits are correct in terms of their method 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.

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 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.

3 Concrete Example

In our example below we will use the notation `@requires(`*predicate*`)` for specifying 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
```

We now show how *Iterative Composition* (enriched by the contract compatibility check we proposed) can be used to write a metaprogram that given an exponent, produces code like the above.

First we will define tree traits: base, even and odd.

```
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);}
}
```

They are the basic building blocks we will use to compute our result. They will be compiled, type-checked and statically verified before being used in any way. Note that we could use base directly: 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. Since all occurrences of names are consistently renamed, **renaming and hiding preserve code correctness**. Method names starting with _, like _pow(x) and _exp(), are not special and are not treated in any special way by trait composition. We use the _ naming convention to emulate super when using trait composition: a call to _exp() is used like a super.exp() call in a language with conventional class based inheritance like Java.

The compose(current,next) method starts by renaming the exp() and pow(x) methods of current so that they satisfy the contracts in next (which will be even or odd).

```
//‘compose’ performs a step of iterative composition
Trait compose(Trait current, Trait next){
  current = current[rename exp()->_exp(), pow(x)->_pow(x)];
  return (current+next)[hide _exp(), _pow(x)];}
@requires(exp>0)//the entry point for our metaprogramming
Trait generate(Int exp) {
  if (exp==1) return base;
  if (exp%2==0) return compose(generate(exp/2),even);
  return compose(generate(exp-1),odd);
};
```

Then, the operator + is used to compose the code of the parameters. Here we show how we ensure that the traditional + operator also handles contracts: we require that the contract annotations of the two competing methods are *compatible*. In this paper, we 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.

Executing `compose(base,even)` or `compose(base,odd)` will pass this test: since the contract of `base.pow()` is the same of `even._pow()` and `even._pow()`, and the same for `exp()`.

Finally the `_pow(x)` and `_exp()` method are hidden, so that the structural shape of the result is the same as `base`'s. Note that this structural equality includes the contracts of methods.

Note that `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(x)` method is only executed once for each top-level `pow(x)` call, so the `hide` operator can inline them. Thus, the result could be identical to the manually optimized code in line 7. We can use our `generate(7)` as follows:

```
47 class Pow7: generate(7)//generate is executed at compile time
48 //the body of class Pow7 is the result of generate(7)
49 /*example usage:*/
50 new Pow7().pow(3)==2187//Compute 3**7
```

4 Future Work

Our approach, as presented in this short paper, 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 `Traits` 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(exp)` in line 32 above as:

```
51 @requires(exp>0)
52 @ensures(result.exp().ensures ==> (result==exp))
```

```
@ensures(result.pow(x).ensures == (result==x**exp()))
Trait generate(Int exp) {...}
```

In this way, we could statically verify the `generate(exp)` 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(exp)`. If `generate(exp)` is used to define a class (such as `Pow7` above), we will guarantee that such class has the expected 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.

5 Conclusion

By exploiting 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. Following general terminology in software verficiation, we say that a trait is *correct* when its methods respect their contracts. Thus our result is that starting from a set of well typed and correct traits, any code resulting from arbitrary many steps of trait composition will also be correct and well typed. In this way, the programmer need only provide correct bulding blocks using traits; code generated by metaprogramming can be integrated with a correct program without needing to use expensive theorem provers or manual verification. Our example is applied to code specialization of a mathematical function, but our experience suggests that Iterative composition can be used to synthesize arbitrary behaviour.

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, doi:10.1007/978-3-540-30569-9_3.

[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, doi:10.1145/1330017.1330022.

[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.

[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.

[7] John Reppy & Aaron Turon (2007): *Metaprogramming with traits*. In: *ECOOP*, Springer, pp. 373–398, doi:10.1007/978-3-540-73589-2_18.

[8] Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz & Andrew P Black (2003): *Traits: Composable units of behaviour*. In: *ECOOP*, 3, Springer, pp. 248–274, doi:10.1007/978-3-540-45070-2_12.

[9] Marco Servetto & Elena Zucca (2014): *A meta-circular language for active libraries*. *Science of Computer Programming* 95, pp. 219–253, doi:10.1016/j.scico.2014.05.003.

[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.