Iteratively Composing Statically Verified Traits

Isaac Oscar Gariano   Marco Servetto

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 Potanin   Hrshikesh Arora

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

alex@ecs.vuw.ac.nz   arorahrsh@myvuw.ac.nz

Object oriented languages supporting static verification usually extends method declarations with syntax supporting pre-post conditions *contracts* [**?**]. We say that contract annotated code is *correct* if all possible execution of such methods would respect their contract. During compilation, directly after typing, an automatic theorem prover checks the constraints. This process can be very slow even on fast hardware, and is usually not complete: static verification is a process that can be applied to contract 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 generate faster code when some parameters can be known in advance. If we wanted to use metaprogramming and static verification together, we could generate code containing also contracts, and those contracts could be checked after metaprogramming has been completed. However, this could be very time consuming, since it would require to verify all the generated code from scratch. Moreover, it could be very hard to follow the generation process of the contracts to be sure that they represents the intentions of the programmer. We propose a disciplined form of metaprogramming based on trait composition and adaptation, where correct and well-typed units of code can be composed and adapted, while guaranteeing that the result is still correct and well-typed. In our system, all the code literals manually written in the application are proven correct by applying static verification, while all the code generated by metaprogramming is correct since it is obtained only by composing and adapting correct code. Note that there is no guarantee that the code resulting from metaprogramming would still be able to pass static verification (since theorem provers are usually not complete). To provide a succinct explanation of our approach, we will show how we can statically verify specialized version of the iconic pow function, using the well known optimization technique ‘repeated squaring’. We will use the annotation `@requires(`*predicate*`)` to specify a precondition, and `@ensures(`*predicate*`)` to specify a postcondition; predicates are boolean expression in terms of `this`, the parameters of the method, and for the `@ensures` case, the `result` of the method call.

In this setting, the following is a correct implementation of exponentiation using repeated squaring:

```
@requires(exp>0) // to avoid the tricky 0**0 undefined case
@ensures(result == x**exp) //notation 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); // even power
  return x*pow(x, exp - 1);}  // odd power
```

Submitted to:
VPT 2019

However, if the exponent was well known, we could write a more efficient version of pow: for example pow7 would look like:

```
@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
}
```

In the following, we will explain the following code, that generates statically verified versions of powi using our disciplined metaprogramming technique: (WE SHOULD ADD LINE NUMBERS)

```
Trait compose(Trait current, Trait next){
  current = current[rename exp->_exp,pow->_pow];
  return (current+next)[hide _exp,_pow];
  }
@requires(exp>0)
Trait generate(Int exp) {
  if (exp==1) return
    class {
      @ensures(result>0) Int exp(){return 1;}

      @ensures(result==x**exp()) Int pow(Int x){return x;}
    };
  if (exp%2==0) return compose(generate(exp/2),
    class {
      @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);}
    });
  return compose(generate(exp-1),
    class {
      @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);}
    });
};
Pow7=generate(7)
...
```

```
new Pow7().pow(3)==?
```

The method compose shows the core of our approach: Traits are first class values and their methods can be renamed, or hidden. Hiding a method may also trigger inlining if the method body is simple enough or used only once. Moreover, two traits can be summed, and the sum will... The compose method expects current to provide pow and exp methods, and next to have pow and exp, and to declare abstract _pow and _next.

In this way, by inductive reasoning, we can start from a base case in line X and then recursively compose the...