Iteratively Composing Statically Verified Traits

Can metaprogramming generate statically verified code reusing common verification techniques?

Isaac Oscar Gariano[1]
Marco Servetto [1]
Alex Potanin [1]
Hrshikesh Arora [1]

[1]Victoria University of Wellington

Pow

```
@requires(exp > 0)
@ensures(result = x**exp)
Int pow(Int x, Int exp) {
  if (exp == 1) return x;
  if (exp %2 == 0) return pow(x*x, exp/2);//even
  return x*pow(x, exp-1); }  //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
```

Pow

- OO languages supporting static verification usually extends method declaration with syntax supporting pre/post conditions. During compilation, directly after typing, an automatic theorem prover checks the constraints.
- Very slow even on fast hardware!
- `pox(x,y)` execution is slower than an hand written `pow7(x)`
- Manually declaring $\texttt{pow}_n$ methods: repetitive, boring.
- Running the static verifier on all $\texttt{pow}_n$ versions: time consuming

Pow

What if we could programmatically generate code at compile time? For example, we could write `generate(y)` as a method generating a class body with a method `pow(x)` computing `pow(x,y)`

```
class Pow7: generate(7)
...
Pow7.pow(4) == pow7(4)
```

Iteratively building up pow

- Verification supports code reuse / class inheritance.
- What if metaprogramming was based on code reuse?
- For example, here we use inheritance to encode statically verified pows while reusing part of the already verified contract
- A code optimizer could inline nested calls, producing the same efficient code of the hand written versions of before

```
class Pow1{
  @ensures(result=x**1)
  Int pow1(Int x){return x;}  }

class Pow2 extends Pow1{
  @ensures(result=x**2)
  Int pow2(Int x){return x*pow1(x);}  }

class Pow3 extends Pow2{
  @ensures(result=x**3)
  Int pow3(Int x){return x*pow2(x);}  }
```

Pow and Exp

```
class Pow1{
  Int exp1(){return 1;}
  @ensures(result=x**exp1())
  Int pow1(Int x){return x;}  }

class Pow2 extends Pow1{
  Int exp2(){return exp1()+1;}
  @ensures(result=x**exp2())
  Int pow2(Int x){return x*pow1(x);}  }

class Pow3 extends Pow2{
  Int exp3(){return exp2()+1;}
  @ensures(result=x**exp3())
  Int pow3(Int x){return x*pow2(x);}  }
```

Traits

```
//induction base case: pow(x)=x**1
Trait base=class {
  @ensures(result>0)@ensures(result=1)Int exp(){return 1;}
  @ensures(result=x**exp())Int pow(Int x){return x;}
  }
//if _pow(x)=x**_exp(), pow(x)=x**(1+_exp())
Trait odd=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);}
}
//if _pow(x)=x**_exp(), pow(x)=x**(2*_exp())
Trait even=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);}
}
```

Trait composition

```
//'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);
  }

class Pow7: generate(7)
//the body of class Pow7 is the result of generate(7)


/*example usage:*/new Pow7().pow(3)==2187//Compute 3**7
```

Comparing code

```
//conventional code
@requires(exp > 0)
@ensures(result = x**exp)
Int pow(Int x, Int exp) {
  if (exp == 1) return x;
  if (exp%2 == 0) return pow(x*x, exp/2);//even
  return x*pow(x, exp-1); }  //odd

//metaprogramming
@requires(exp>0)
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);
  }

class Pow7:{//generated code
@ensures(result>0) exp(){return 1+1+1+1+1+1+1;}
@ensures(result = x**exp()) Int pow(Int x) {??}
```

Concluding

- Idea: merge conventional verification and trait composition.
- Iterative composition (metaprogramming) allows to generate code that is correct by construction.
- Theorem prover/manual verification only for the basic building blocks.
- Contracts are syntactically matched during the sum operation.
- What is the contract of the result?
- We are searching for relevant related work!

Thanks

Questions?