# Metaprogramming Statically verified code

Hrshikesh Arora (arorahrsh@myvuw.ac.nz),
Marco Servetto (marco.servetto@ecs.vuw.ac.nz)

Can metaprogramming generate statically verified code reusing common verification techniques?

# Example of verification (Spec#)

```
Nat pow(Nat x,Nat y) ensures result = x^y;{
  if(y==0){return 1;}
  return x*pow(x,y-1);}
```

- 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 pow3(x)

```
Nat pow3(Nat x) ensures result = x^3;{return x*x*x;}
```

- Manually declaring $pow_n$ methods: repetitive, boring
- Running the static verifier on all $pow_n$ versions: time consuming

# Metaprogramming and verification

```
Nat pow3(Nat x) ensures result = x^3;{return x*x*x;}
```

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

```
Pow3=MakePow.ofExp(3)
Pow3.pow(4) == pow3(4)
```

- There is very little literature on generating statically verified code with metaprogramming. Metaprogramming usually relies on quasi quotation, an AST based technique that do not consider contracts

# 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 Pow0{
  Nat pow0(Nat x) ensures result=x^0;{return 1;}
  }
class Pow1 : Pow0{
  Nat pow1(Nat x) ensures result=x^1;{return x*pow0(x);}
  }
class Pow2 : Pow1{
  Nat pow2(Nat x) ensures result=x^2;{return x*pow1(x);}
  }
class Pow3 : Pow2{
  Nat pow3(Nat x) ensures result=x^3;{return x*pow2(x);}
  }
```

# Metaprogramming

**class bodies** as first class values
**code reuse** operators
can be used to iteratively compose arbitrary behavior

- In the following, we rely on a language where class bodies (unnamed classes with methods) are first class values, and:
  - `code1 + code2` composes code similar to inheritance;
  - `code[rename a->b]` renames methods;
  - `code[hide a]` renames methods to private names.

# Sum: code1 + code2 = code3

```
class{
  abstract String hello();
  String helloWorld(){return hello()+" World";}
  }
+
class{String hello(){return "Hi";} }
=
class{
  String hello(){return "Hi";}
  String helloWorld(){return hello()+" World";}
  }
```

- The result of the sum contains the methods of both arguments.
- If a method is present in both arguments, they need to have the same type signature, and at least one of the two needs to be abstract.

# Sum and verification

```
class{
  abstract String hello() ensures result.size()>1;
  String helloWorld(){return hello()+" World";}
  }
+
class{String hello() ensures result.size()>1;{return "Hi";}}
=
class{
  String hello() ensures result.size()>1;{return "Hi";}
  String helloWorld(){return hello()+" World";}
  }
```

- Contracts are matched: **abstract** c1 + c1 & c2 = c1 & c2
- This example would produce an error if the implemented contract did not contained '**result**.size()>1'
- Contract matching is syntactical: '**result**.size()==2'

  would not work, it would need to be in the (equivalent) form

  '**result**.size()>1 & **result**.size()==2'

# Example revised

```
Pow0 = class{
  Nat pow(Nat x) ensures result=x^0;{return 1;}
  }
Pow1 = (Pow0[rename pow->_pow] + class{
  abstract Nat _pow(Nat x) ensures result=x^0;
  Nat pow(Nat x)ensures result=x^1;{return x*_pow(x);}
  })[hide _pow]
Pow2 = (Pow1[rename pow->_pow] + class{
  abstract Nat _pow(Nat x) ensures result=x^1;
  Nat pow(Nat x) ensures result=x^2;{return x*_pow(x);}
  })[hide _pow]
//and so on
```

- This code is more explicit then before: **abstract** _pow!
- + composes the **abstract** _pow with the concrete _pow
- The exponents are still hard coded in the contracts!!

# Example revised

```
Pow0 = class{
  Nat exp() ensures result=0;{return 0;}
  Nat pow(Nat x) ensures result=x^exp();{return 1;}
  }
Pow1 = (Pow0[rename pow->_pow, exp->_exp] + class{
  abstract Nat _exp()/*empty contract */;
  abstract Nat _pow(Nat x)ensures result=x^_exp();
  Nat exp() ensures result=1+_exp();{return 1+_exp();}
  Nat pow(Nat x)ensures result=x^exp();{return x*_pow(x);}
  })[hide _pow, _exp]
Pow2 = (Pow1[rename pow->_pow, exp->_exp] + class{
  abstract Nat _exp()/*empty contract */;
  abstract Nat _pow(Nat x)ensures result=x^_exp();
  Nat exp() ensures result=1+_exp();{return 1+_exp();}
  Nat pow(Nat x)ensures result=x^exp();{return x*_pow(x);}
  })[hide _pow, _exp]
```

- Note the only difference between Pow2 and Pow1
- We can use a loop to get metaprogramming!

# Example Metaprogramming

```
MakePow = class{
  static Code ofExp(Nat y) {
    ClassLit base=class{
      Nat exp()ensures result=0;{return 0;}
      Nat pow(Nat x)ensures result=x^exp();{return 1;}};
    ClassLit inductive=class{
      abstract Nat _exp()/*empty contract */;
      abstract Nat _pow(Nat x)ensures result=x^_exp();
      Nat exp() ensures result=1+_exp();{return 1+_exp();}
      Nat pow(Nat x) ensures result=x^exp();{return x*_pow(x);}};
    ClassLit res=base;
    for(Nat i in 0..y){
      res=res[rename pow->_pow, exp->_exp];
      res=res+inductive;
      res=res[hide _pow, _exp];
      }
    return res;
    }  }
...
Pow7 = MakePow.ofExp(7);//example usage
```

# Example Metaprogramming

- How does static verification flow? what do we know and when do we know it?

```
static Code ofExp(Nat y) {
  Code base={..};//SV during compilation of MakePow.ofExp
  Code inductive={..};//SV during compilation of MakePow.ofExp
  Code res=base;//SV since base is SV
  for (Nat i in 1..y) {
    res=res[rename pow->_pow, exp->_exp];
    //res is still SV since contracts are resilient to renaming
    res=res+inductive;
    //+ sums contracts in  a way that preserves SV
    res=res[hide _pow, _exp];
    //res is still SV since contracts are resilient to hiding
    }
  return res;//res is still SV
  //However, statically, we do not know what its contracts are
  }
```

# Example Metaprogramming

```
MakePow = class{
    static Code ofExp(Nat y)
    ensuresRV result.exp.ensures equivTo (result = y)
      & result.pow.ensures = (result = x^exp());
    { .. }  }
...
Pow7 = MakePow.ofExp(7);//mini verification here for equivTo
```

- Code composition ensure contract matching, but the resulting contract is unknown
- Thus, the contract for MakePow.ofExp(y) can only be checked at run-time. The contract of pow(x) is checked to be identical to `x^exp()` ; but checking exp() is harder: `@result.exp().ensures equivTo (@result = y)`
- EquivTo checks that the contract `1+_exp$342()` is equivalent to y, this needs the contract of private method `_exp$342()`

# Concluding

- Iterative composition is a disciplined metaprogramming technique that reasons at class level
- It allows holes in the behavior using abstract methods
- The static verifier runs only once on each piece of source code
- Metaprogramming reuses already SV code, which does not need to be SV again
- Only few and cheap checks are performed when code is composed

# Example of metaprogramming: QQ

```
fun powerAux(n : Nat) : Expr< x:Nat |- Nat > =
  if (n = 0) then [| 1 |]  else [| x * $(  powerAux (n -1) ) |];

fun powerGen(n : Nat) : Nat -> Nat =
  compile (  [| λ x . $( powerAux (n)  ) |]  );

power7 = powerGen 7;
power7 = λ x . x*x*x*x*x*x*x*1 //equivalent!
```

- Quasi Quotations are expressions not first class values.
- The Value of type **Expr** have no holes any more.
- Holes (denote by **$(** e **)**) do not have a name.
- There is no obvious way to map verification on QQ.