42 Programming Language
Iterative compilation

Flattening

```
A:{ 'a class with "m" returning a class with "foo"
  type method Library m() { N foo() 42N }
  'in Java this would look like
  'static Library m(){return { int foo(){return 42;}};}
  }
B: A.m() 'a class generated using library A
```

rewrites to

```
A:{ 'a class with "m" returning a class with "foo"
  type method Library m() { N foo() 42N }
  }
B:{ N foo() 42N }'a class
```

Flattening

```
N:{..}'number
S:{..}'string
A:{.. }'uses N,S
B: A.m()'uses A
D:{..E..}
C: B.m({..D..})
E: ..
```

```
N:{..}'number
S:{..}'string
A:{.. }'uses N,S
B: A.m()' A,N,S
D:{..E..}
C: B.m({..D..})
E: ..
```

```
N:{..}'number
S:{..}'string
A:{.. }'uses N,S
B: A.m()' ok
D:{..E..}
C: B.m({..D..})
E: ..
```

```
N:{..}'number
S:{..}'string
A:{.. }'uses N,S
B:{..}'uses A,N,S
D:{..E..}
C: B.m({..D..})
E: ..
```

- find the first class that requires meta execution.
- type-check all the (transitively) used classes, starting from the expression.
- type-check the yellow expression. We have enough type information now.
- execute the well typed expression. Now we get a class. By construction, this class can only refer to the types collected before. Does it means it is already guaranteed to be well-typed?

Type-check steps can end the execution with a type error, while the execution step can end the execution with an exception.

```
N:{..}'number
S:{..}'string
A:{.. }'uses N,S
B:{..}'uses A,N,S
D:{..E..}
C: B.m({..D..})
E: ..
```

```
N:{..}'number
S:{..}'string
A:{.. }'uses N,S
B:{..}'uses A,N,S
D:{..E..}'uses E
C: B.m({..D..})
E: ..'hello?
```

```
N:{..}'number
S:{..}'string
A:{.. }'uses N,S
B:{..}'uses A,N,S
D:{..E..}'uses E
C: B.m({..D..})
E: ..'hello?
```

```
N:{..}'number
S:{..}'string
A:{.. }'uses N,S
B:{..}'uses A,N,S
D:{..E..}
C:{..}'uses?
E: ..
```

- find the first class that requires meta execution.
- type-check the used classes. This time, we have a class literal that uses **D** (that uses **E**). What happens here?
- type-check the yellow expression. We have enough type information now.

Can we proceed without verifying **{..D..}**? How the decorators/composition operators are going to work over a non verified **{..D..}**?

N:{..}'number S:{..}'string A:{.. }'uses N,S B:{..}'uses A,N,S D:{..E..} C:{..}'uses E E: ..	=====> =====> =====> =====>	N:{..}'number S:{..}'string A:{.. }'uses N,S B:{..}'uses A,N,S D:{..E..} C:{..}'uses E E:{..}	N:{..}'number S:{..}'string A:{.. }'uses N,S B:{..}'uses A,N,S D:{..E..} C:{..}'uses E E: ..

- Eventually, we resolve also E.
- Finally, some class may still be not type-checked
- As a last step, we verify all those classes.

Flattening

```
I: {interface
  method N m()    }
C:{
  type method Library myReusableCode() {<: I
    method m() 42N
    }
  }
D:C.myReusableCode()
```

rewrites to

```
I: {interface
  method N m()    }
C: ...
D:{<: I 'note, there is no relation between D and C
    method m() 42N
    }
```

```
C:{
  type method Library a() {
    method N m1(N that) that+1N
    }
  type method Library b() {
    method N m1(N that)   'this is an abstract method
    method N m2(N that) this.m1(that)+1N
    }         ' "this" is of type Outer0, no relation with C
  }
D:Compose[ C.a() ] << C.b()' the result (if any) will be well typed
```

rewrites to

```
C:{
  type method Library a() {
    method N m1(N that) { return that+1N; }
    }
  type method Library b() {
    method N m1(N that)
    method N m2(N that) this.m1(that)+1N
    }         ' "this" is of type Outer0, no relation with C
  }
D:{'this code is well typed by construction
  method N m1(N that) that+1N
  method N m2(N that) this.m1(that)+1N
  }         ' "this" is of type Outer0 *and* Outer1::D
```

```
C:{     'this code is required to be completely well typed
          'before D can be generated
  type method Library a() {
    method N m1(N that) that+1N
    }
  }
D:Compose[ C.a() ] << {        'the result (if any) will be well typed
  method N m1(N that)  ' if this code is correct
  method N m2(N that) this.m1(that)+1N
  }         ' "this" is of type Outer0, is it also Outer1::D
```

rewrites to

```
C:{
  type method Library a() {
    method N m1(N that) that+1N
    }
  }
D:{
  method N m1(N that) that+1N
  method N m2(N that) this.m1(that)+1N
  }         ' "this" is of type Outer0 *and* Outer1::D
```

Sometimes, we need to delay type checking for code in class literals.

```
A:Data[] << {'the result well-typedness depends from the
            ' code in the dots
  method B playWithB(B that) .. that.fooB() ..
  ..
  }
B:Data[] << {'the result well-typedness depends from the
            ' code in the dots
  method A playWithA(A that) .. that.fooA() ..
  ..
  }
```

By using D I can refer to the type "after" the composition

```
C:{     'this code is required to be completely well typed
          'before D can be generated
  type method Library a() {
    method N m1(N that) that+1N
    }
  }
D:Compose[ C.a() ] << {'code not required to be checked at this stage.
  method N m2(N that, D other) other.m1(that)+1N
  } 'note, no declaration for m1!
```

rewrites to

```
C:{
  type method Library a() {
    method N m1(N that) that+1N
    }
  }
D:{'the code will be checked to be well typed now!
  method N m1(N that) that+1N
  method N m2(N that, D other) other.m1(that)+1N
  }
```

Is this code still ok? why or why not?

```
C:{    'this code is required to be completely well typed
         'before D can be generated
  type method Library a() {
    method N m1(N that) that+1N
    }
  }
D:Compose[ C.a() ] << {'code not required to be checked at this stage.
  method N m2(N that, D other) this.m1(that)+1N
  } 'note, no declaration for m1!
```

rewrites to

```
C:{
  type method Library a() {
    method N m1(N that) that+1N
    }
  }
D:{'this code is now available to type-check
  method N m1(N that) that+1N
  method N m2(N that, D other) this.m1(that)+1N
  }
```

What happens here?

```
D:Compose[{  method T foo()...}]<<
  Refactor::RenameSelector[ Selector"foo()" to: Selector"bar()" ] <<{
    method D foo() this
    method Any m1() this.foo().foo()
    }
```

rewrites to

```
D:Compose[{  method T foo()...}]<< {
    method D bar() this
    method Any m1() this.bar().foo()
    }
```

rewrites to

```
D:{
  method T foo() ...
  method D bar() this
  method Any m1() this.bar().foo()
  }
```

Here `this.foo()` returns `D`, and at the start we do not know what methods `D` will have.

Sometimes "typish" information is needed for the operators to work

```
D:Refactor::RenameSelector[ Selector"m()" to: Selector"k()" ]<< {
  method Outer0 m() this.m()
  method D toD() this
  method Outer0 m0() Foo().m()
  method Outer0 m1() this.m().m()
  method Outer0 m2() this.toD().m()
  Nested:{   type method Outer1 id(Outer1 that) that
             type method D toD(Outer1 that) that   }
  method Outer0 m3() Nested.id(this).m()
  method Outer0 m4() Nested.toD(this).m()
  }
```

rewrites to

```
D:{
  method Outer0 k() this.k()'renamed 2 times
  method D toD() this
  method Outer0 m0() Foo().m()
  method Outer0 m1() this.k().k()'renamed 2 times
  method Outer0 m2() this.toD().m()
  Nested:{   type method Outer1 id(Outer1 that) that
             type method D toD(Outer1 that)that      }
  method Outer0 m3() Nested.id(this).k()'renamed
  method Outer0 m4() Nested.toD(this).m()
  }
```

Final example

What happens here?

```
D:Refactor::RenameSelector[ Selector"m()" to: Selector"k()" ] << {
  method Outer0 m1() this.m().m()
  }
```

If `this.m()` returns `Outer0` I have to rename both calls, otherwise only the first one. But there is no `method Outer0 m()` in the code.
Trying to rename a method that is not declared: we decided to treat this as an exception.