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

In this paper we show how the techniques used for method contracts in class-based OO languages, with conventional inheritance, can be applied to the more powerful and general mechanism of *iterative trait composition*. Iterative trait composition extends the notion of *trait composition* by combining it with support for meta-circular meta-programming.

We present a novel extension to the core trait composition operator that guarantees that any meta-code taking in statically verified traits, will produce a 'correct' trait. We achieve this property by simply checking the compatibility of method contracts when composing traits. Importantly, our approach does *not* require invoking a static verifier on the result of meta-programming, or during its execution.

1 Introduction

Meta-programming is a very useful technique for writing code, as it can automatically generate large chunks of code following arbitrary patterns. If we want to ensure our meta-programming produces 'correct' code, we could try and statically verify the meta-programming itself, though we are unaware of any work that actually enables this.

Just like static verification, meta-programming is often executed during compilation (or at 'meta-time'), before the program is actually run. So instead of statically verifying the program *before* we compile it, we could do so afterwards, on the result of the meta-programming. However, static verification can be slow, in addition it is often hard to predicate when it will fail to verify correct code. Instead we propose a technique by which only the *input* to the meta-programming needs to be statically verified. Our technique works by ensuring, through a simple and predictable meta-time check, that the result of primitive meta-operations is always 'correct' whenever their inputs are.

The meta-programming technique of *iterative trait composition* [9] makes this possible, as its primitive operations (in particular *trait composition* work on entire pieces of self-contained code/classes ('traits'). Though prior work does look at verifying the structure of the result of trait composition [4], such work does not consider the correctness of the *behaviour* of the result's methods. Trait composition differs significantly from other meta-programming techniques that work on incomplete code chunks, such as quasi-quotation [10] which operates directly on (partial) abstract syntax trees. We do not believe it would be simple or easy to verify the correctness of AST operations, it is not even clear to us what it should mean for an (incomplete) AST to be 'correct'.

Submitted to:
VPT 2019

An alternative way of generating code is to use higher order functions, which can also generate arbitrary behaviour. However, unlike iterative trait composition and quasi-quotation, they cannot generate classes or declarations. One can statically verify higher order functions to ensure they return code with certain contracts [12]. A potential avenue for future work would be to combine higher order functions with iterative trait composition: where one would use higher order functions to generate behaviour, and iterative trait composition to generate declarations.

In Section 2 we discuss prior work in specifying OO code, and explain the concept of iterative trait composition. In Section 3 we present and explain our contribution. In Section 4 we show how our approach works with two examples. Section 5 discusses conclusions (including limitations of our work) as well as directions for future research.

2 Background

2.1 Static Verification

The mechanism by which programs are verified to be correct is a large research area, one common approach is *automated static verification*, where a tool tries to automatically verify that code is correct during compilation, such tools typically encode the correctness of a program as mathematical theorems, and use an SMT solver to verify it []. Automated static verification is often easier to use and provides stronger guarantees compared to computer aided theorem proving [13] and runtime verification [11] (respectively).

For simplicity, we will only consider automated static verification; the details as to how this is done is also out of scope of this paper. However, our approach should be easily extendible to other verification techniques.

2.1.1 Contracts

There are many notions as to what it means for a program to be correct, once can say that a program is correct if it conforms to an arbitrary mathematical *specification*. Rather than specifying a program as a whole, a simpler and more modular approach is to specify individual components by giving them *contracts* [6]. For functions, contracts are typically just pre and post-conditions. In this paper, we will use the annotation `@requires(`*predicate*`)` to specify a precondition, and `@ensures(`*predicate*`)` to specify a postcondition: where *predicate* is a boolean expression in terms of `this`, the parameters of the method, and for the `@ensures` case, the `result` of the method call. We will allow specifying the annotations multiple times, so that `@requires(`$p1$`) @requires(`$p2$`)` is equivalent to `@requires(`$p1$ `||` $p2$`)`, and `@ensures(`$p1$`) @ensures(`$p2$`)` is equivalent to `@ensures(`$p1$ `&&` $p2$`)`. Also, if either annotation was not specified, we will assume a *predicate* of `true`.

This means that the annotated *function* is correct if whenever the precondition holds, executing the function will cause the postcondition to hold[1], as well as the preconditions of any functions it calls. A *program* is then correct if each of its functions is correct. For example, one could write an implementation of exponentiation using repeated squaring like this:

```
@requires(x != 0 || exp != 0) // since 0**0 is undefined.
@ensures(result == x**exp)
```

[1]There is some disagreement as to whether a non-terminating function is trivially correct [7], however this does not affect the results of this paper.

```
Int pow(Int x, Nat exp) {
  if (exp == 0)           return 1;
  else if (exp == 1)      return x;
  else if (exp % 2 == 0) return pow(x*x, exp/2); // even power
  else                    return x*pow(x, exp - 1);}  // odd power
```

This says that for the *implementation* of `pow` to be 'correct', whenever we don't have `x == 0` and `exp == 0`, `pow(x, exp)` must equal `x**exp`[2]. For a *call* `pow(x, exp)` to be correct, either `x` or `y` must be nonzero.

For simplicity, we will only consider pre and post-conditions in a purely functional OO language. Contracts can be extended to other cases, such as to specify class invariants, termination, function purity (where the language is an otherwise imperative language), ownership, etc. [] However such properties are still typically encoded as pre and post-conditions.

2.2 OO and Static Verification

One of the main things you can do with OO is create an abstract specification of behaviour (an 'interface') and write code that will work with any concrete instance of this interface. Naturally, there is much research on verifying the implementations of such interfaces [2]. The common approach is to apply pre and post-conditions to methods, but only verify the post-conditions for concrete *implementations* of them. Consider for example an OO style abstract class/interface:

```
abstract class List {
  Nat length(); // abstract method
  @requires(index < this.length())
  Object get(Nat i);

  @requires(this.contains(o))
  @ensures(this.get(result) == o)
  Bool contains(Object o);

  @ensures(result == exists i : this.get(i) == o)
  Bool contains(Object o);
}
```

What this means is that any *object* claiming to be a `List` must have `get`, `contains` and `find` methods satisfying their declaration above, namely their type signatures and contracts. For static verification, this means that any *call* to a method of `List` must satisfy the precondition, and any *implementation* of such a method must ensure that the postcondition holds (whenever the precondition holds on entry to the method). Because none of the methods in `List` are implemented, they are trivially correct.

One can use *inheritance* to write implementations of these methods, for example consider the following two classes:

```
abstract class ListFinder extends List {
  @ensures(forall i : i < result ==> this.result(i) != o)
  //@requires(true) // implicit
  //@requires(this.contains(o)) // inherited
```

[2] We use the notation `x**y` to mean '`x` to the power of `y`'.

```
  //@ensures(this.get(result) == o) // inherited
  Nat find(o) {
    for (Nat i = 0; i < this.length(); i++) {
      if (this.get(i) == o) { return i; } }
    return this.length() + 1; }
}

abstract class ListContains extends List {
  //@ensures(result == exists i : this.get(i) == o) // inherited
  Bool contains(Object o) {
    for (Nat i = 0; i < this.length(); i++) {
      if (this.get(i) == o) { return true; } }
    return false; }}
```

The above classes *inherit* the methods of `List`, including their contracts, so their code can call these methods, and a static verifier can use these contracts. Importantly, the contract annotations of overridden methods will be inherited, as seen above, this ensures the Liskov Substitution Principle (LSV) [5]. Thus the precondition can be weakened, and the postcondition strengthened. An abstract method often overrides another abstract method, in order to refine contracts like this.

Finally, we can compose these two classes (using `extends`), and add the missing methods, to create a complete implementation of `List`:

```
class LinkedList extends ListContains, ListFinder {
  ... // implementations for get and length
}
```

2.3 Trait Composition

The idea behind *trait composition* is to seperate the notions of *subtyping* and *inheritence* [3]. Only *traits* can be inherited, and only *classes* can be used as types. A *trait* is like an anonymous class, in particular it can contain methods, but because it is anonymous, one cannot directly use it as a type or create instances of it. Class are then declared from traits, effectively giving them a name, but such classes will be unrelated to other classes defined from the same trait. Traits can be ‘composed‘ to produce other traits, however since traits cannot be used as types, we are free to perform operations that would break the LSV. Consider the following example:

```
Trait pair = class {
  String target();
  String hello() { return "Hello " + this.target(); }
};
// Defines a class called HelloWorld
HelloWorld: pair.extend(class {
  String target() { return "World"; }
  String hello() { this.super_print() + "!"; } // prints "Hello World!"
});
```

We use the type `Trait` as the types of traits, and a class declaration (without a name) as a *trait literal* expression. Here `trait1.extend(trait2)` contains all the methods in `trait1` and `trait2`. When a

method declaration appears in both `trait1` and `trait2`, their signatures will also be checked to ensure they are identical, this ensures that if both `trait1` and `trait2` are well-typed, the result will also be (this property is sometimes called *meta level soudnes* [8]).

However if a method *m* is also *implemented* (i.e. non abstract) in both `trait1` and `trait2`, the operation will proceed as if the version in `trait1` was named `super_m`. Unlike conventional OO inheritance, this means that calls to *m* in `trait1` will call the implementation in `trait1`, and not the (overriding) implementation in `trait2`. In addition, the renamed `super_m` method will be 'private' to `trait2`, i.e. only code in `trait2` itself can see it, not code added in later. Their are other approaches to dealing with this, such as only renaming the declaration of *m* to `super_m`, so that calls in `trait1` will call the method in `trait2`. However in order for this to be sound in our context of producing correct code, we would have to prevent the *m* in `trait2` from having an incompatible contract to the *m* in `trait1`, thus breaking our examples in Section 4.

In this paper, we will use the *flattening* semantics of traits, in which trait operations produce a flattened result, equivalent to a trait literal, with no references to any of the input traits to the operation. Thus the above definition for `HelloWorld` is identical to:

```
HelloWorld: class {
  private String super_hello() { return "Hello " + this.target(); }
  String target() { return "World"; }
  String print() { this.super_print() + "!"; }
};
```

The main advantage of trait composition over conventional inheritance is that trait composition does *not* induce subtyping:

- Trait operations that would be unsound in a subtyping environment are allowed, such as removing methods or changing their types.
- The way code is reused in order to construct a class is not exposed to users of the class. This allows the code reuse mechanism to be changed without breaking any uses of a class.

In addition, this makes traits much simpler to reason about, making it easier to compose them in more complicated ways.

2.4 Iterative Trait Composition

Iterative trait composition [9] takes the concept of trait composition one step forward, by treating traits as first class objects, and allowing arbitrary code to execute at meta-time. This allows for fully met-circular meta-programming. In this way, classes can be declared from arbitrary expressions (of type `Trait`), and methods can take and return `Trait`s. For example, suppose we want to generate specialised code for `pow` based on the exponent[3]:

```
Trait pow_generate(Nat exp) {
  if (exp == 0)
    return static class {
      Int pow(Int x) { return 1; }
    };
  else if (exp == 1)
```

[3]We use `static class` to mean a class with no constructors or instances, and where each member is implicitly `static`.

```
    return static class {
      Int pow(Int x) { return x; }
    };
  else if (exp % 2 == 0)
    return pow_generate(exp/2).extend(static class {
      Int super_pow(Int x);
      Int pow(Int x) { return super_pow(x*x); }
    });
  else
    return pow_generate(exp - 1).extend(static class {
      Int super_pow(Int x);
      Int pow(Int x) { return x*super_pow(x); }
    });}
```

In the above code, we have a normal looking recursive `pow_generate` method: the `pow_generate(...)` calls (recursively) generate the bodies of `super_pow`. The calls to `super_pow` (in the trait literals) correspond to the recursive calls in our original `pow`. The above method can now be called to generate the body of a class:

```
Pow7: pow_generate(7);

// Equivalent to:
Pow7: static class {
  Int pow(Int x) { return x*super1_pow(x); } // x**7

  private Int super1_pow(Int x) { return super2_pow(x*x); } // x**6
  private Int super2_pow(Int x) { return x*super3_pow(x); } // x**3
  private Int super3_pow(Int x) { return super4_pow(x*x); } // x**2
  private Int super4_pow(Int x) { return x; } // x**1
}

// Which could then be automatically inlined/optimised to:
Pow7: static class {
  Int pow(Int x) {
    Int x2 = x*x; // x**2
    Int x4 = x2*x2; // x**4
    return x*x2*x4; } // Since 7 = 1 + 2 + 4
}
```

3 Combining Contracts and Trait Composition

We can handle the composition of traits with contracts in a similar way to what is done for class-based inheritance: the methods in the result of `trait1.extend(trait2)` contain the `@requires` and `@ensures` clauses of the corresponding method (if any) in `trait1` and `trait2`. This ensures that any *calls* to such methods in `trait1` and `trait2` are still correct, and any deductions made from the postconditions of such methods still hold. If a method is implemented in either `trait1` or `trait2`, but abstract in the other,

the implemented version must have at least all the `@requires` and `@ensures` clauses of the abstract one. This ensures that the implementation is still a correct implementation for the abstract method.

Note that renaming of methods, such as the `super_` renaming described in Section 2.3, is sound provided it is also performed within contracts.[4]

Consider the following examples illustrating how this works:

```
Trait trait: class {
  @requires(R1)
  @ensures(E1)
  Void foo();
}.extend(class {
  @requires(R2)
  @ensures(E2)
  Void foo();
});
// Identical to:
Trait trait: class {
  @requires(R1) @requires(R2)
  // or @requires(R1 || R2)
  @ensures(E1) @ensures(E2)
  // or @ensures(E1 && E2)
  Void foo();
};

Trait good: class {
  @requires(R1)
  @ensures(E1)
  Void foo();
}.extend(class {
  // precondition is weaker
  @requires(R1) @requires(R2)
  // postcondition is stronger
  @ensures(E1) @ensures(E2)
  Void foo() { ... }
});
```

```
// Error!
Trait error: class {
  @requires(R1) @requires(R2)
  @ensures(E1) @ensures(E2)
  Void foo();
}.extend(class {
  // stronger precondition
  @requires(R1)
  // weaker postcondition
  @ensures(E1)
  Void foo() { ... }
});

Trait really_good: class {
  @requires(R1) @requires(R2)
  @ensures(E1) @ensures(E2)
  // will be renamed to super_foo
  Void foo() { ... }
}.extend(class {
  // stronger precondition
  @requires(R1)
  // weaker postcondition
  @ensures(E1)
  Void super_foo();

  // any contract is ok here
  @requires(R3)
  @ensures(E3)
  Void foo() { ... }
})
```

These rules guarantee that if all methods in `trait1` and `trait2` are correct, then `trait1.extend(trait2)` will fail with an error, or it will produce a correct result. This means that if all trait literals in the program are statically verified, we can be sure that any traits produced by meta-programming/composition (and hence all class declarations) will be correct by construction, without needing any further static verification.

A more general rule[5] would be that an abstract method with pre and post-conditions[6] R_1 and E_1

[4] This of course assumes that the language doesn't provide constructs that depend on method *names* (such as reflection or stack trace operations).

[5] We believe it is the most general sound rule that is independent of the method bodies in the traits, however we have not proven it.

[6] Where the pre/post-conditions are the disjunction/conjunction of all the `@requires`/`@ensures` clauses (respectively).

(respectively) can only be implemented by a method with pre and post-conditions R_2 and E_2 if $R1 \Rightarrow R2$, and $R2 \wedge E2 \Rightarrow E1$. This would require using a theorem prover to verify that the these implications always hold. However, this could be significantly faster than the alternative of verifying that the implementation of the method is correct under the contract of the abstract version.

4 Examples of Our Technique

We now show how the technique described above in Section 3 can be used with iterative composition to programmatically generate guaranteed-correct code.

4.1 Recursive Composition Example

We now extend our `pow_generate` example from Section 2.4 to generate correct code, without needing to run a static verifier each time it is called:

```
Trait pow_generate(Nat exp) {
  if (exp == 0)
    return static class {
      @ensures(result == 0)
      Nat exp() { return 0; }

      @requires(x != 0)
      @ensures(result == x**exp())
      Int pow(Int x) { return 1; }
    };
  else if (exp == 1)
    return static class {
      ... // similar to the above, but without the @requires
    };
  else if (exp % 2 == 0)
    return pow_generate(exp/2).extend(static class {
      Nat super_exp();

      @ensures(result == x**super_exp())
      Int super_pow(Int x);

      @ensures(result == 2*super_exp())
      Nat exp() { return 2*super_exp(); }

      @ensures(result == x**exp())
      Int pow(Int x) { return super_pow(x*x); }
    });
  else
    return pow_generate(exp - 1).extend(static class {
      ... // similar to the above
    });}
```

Each of the trait literals above (the `static class {...}` expressions) can be statically verified as being correct. The function `pow_generate` then combines these trait literals with our `extend` operator, guranteeing that the result (if any) is also correct. The idea is that `exp` and `pow` represent the current exponent and power function we are generating, `super_exp` and `super_pow` correspond to the exponent and power function of the recursive call. Though our trait literals are more verbose than in our non verified version, they ensure that the contract matching performed by `extend` will succeed.

The idea is that `pow_generate(`e`)` (where $e > 0$) will return a literal of the following form:

```
static class {
  @ensures(...)
  Nat exp() { ... } // Will return e when called

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

  ... // and perhaps some private super_ methods
}
```

Then `pow_generate(`e'`)` (where $e' > e$) will `extend` this trait with one of the form:

```
static class {
  Nat super_exp();

  @ensures(result == x**super_exp())
  Int super_pow(Int x);

  @ensures(...)
  Nat exp() { ... } // Will return e' when called

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

Because `exp` and `pow` are implemented in both `pow_generate(`e`)` and `pow_generate(`e'`)`, `extend` will add a `super_` prefix to the ones in `pow_generate(`e`)`. Thus `pow_generate(`e'`)` will return:

```
static class {
  @ensures(...)
  Nat super_exp() { ... } // Will return e when called

  @ensures(result == x**super_exp())
  Int super_exp(Int x) { ... }

  ... // And some private methods (which will not clash with the above)
}.extend(static class {
  Nat super_exp();

  @ensures(result == x**super_exp())
  Int super_pow(Int x);
```

```
  @ensures(...)
  Nat exp() { ... } // Will return e' when called

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

The contracts of both operands of the `extend` are compatible, and so the above will succeed and produce something of the form:

```
static class {
  @ensures(...)
  Nat exp() { ... } // Will return e' when called

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

  @ensures(...)
  private Nat super_exp() { ... } // Will return e when called

  @ensures(result == x**super_exp())
  private Int super_exp(Int x) { ... }

  ... // maybe some more private methods
}
```

Note that `pow_generate` never recursively calls `pow_generate(0)`, so the `@requires` clause in the result of `pow_generate(0)` will not cause any problems with our contract matching.

Though our system guarantees that the result of `pow_generate` is 'correct' (i.e. the methods in the return trait satisfy their contracts), it does not say what methods will be in the result or what their contracts will be. We could statically verify `pow_generate` itself, however an easier option would be to check at runtime that the result of `pow_generate(`e`)` is of the form:

```
static class {
  @requires(true)
  Nat exp() { ... }

  @requires(x != 0) // if e == 0
  @requires(true) // otherwise
  @ensures(result == x**exp())
  Int pow(Int x) { ... }
}
```

This could be done with introspection on the AST of the resulting trait. We also need to check that calling the above `exp()` method produces e. However, static methods in traits cannot be directly called, instead we could dynamically create a class from the trait and call `exp()` on the result. Alternatively, we could manually 'interpret' the method's AST.

4.2 Iterative Composition Example

We can use introspection to automatically generate code for an input trait. Consider for example the following interface[7]):

```
Account: interface {
  Nat income();

  @ensures(result <= this.income())
  Nat expenses();
};
```

The idea being that an account cant spend more money than they it has.

Now Suppose we want to create a new type of account, a combined one with multiple sub accounts:

```
BuisnessAccount: class implements Account {
  Account salary;
  Account sales;
  Account property;

  Nat income() {
    return this.salary.income()
      + this.sales.income() + this.property.income(); }

  @ensures(result <= this.income())
  Nat expenses() {
    return this.salary.expenses()
      + this.sales.expenses() + this.property.expenses();}

  ...
};
```

Now we can statically verify the above code, but what if we want to combine accounts like this frequently? This could take lots of time for both programmers and automated static verifiers, instead we can create a function `combine_accounts` that does this for us:

```
// Equivalent to the above
BuisnessAcount: combine_accounts(class {
  Account salary;
  Account sales;
  Account property;
  ... });
```

Were we define `combine_accounts` like this:

```
Trait combine_accounts(Trait input) {
  Trait res = input.extend(class implements Account {
    Nat income() { return 0; }
```

[7] An abstract class with only abstract methods.

```
    @ensures(result <= this.income())
    Nat expenses() { return 0; }
  });
  for (FieldDeclaration fd : input.fields()) {
    res = res.extend(class implements Account {
      Nat super_income();

      @ensures(result <= this.super_income())
      Nat super_expenses();

      Account account; // Will be renamed to fd.name
      Nat income() {
        return this.super_income() + this.account.income(); }

      @ensures(result <= this.income())
      Nat expenses() {
        return this.super_expenses() + this.account.expenses(); }
    }.rename("account", fd.name)); }
  return res;
}
```

Where `trait.fields` returns AST structures for each field in the given trait, and `trait.rename(`a`,`b`)` returns `trait` but with the declaration named a renamed to b. The code above is straightforward: it adds an implementation of `Account` (suitable if `input` has no fields) to the `input` trait, it then iterates over every field in `input` and adds its corresponding `income()` and `expenses()` to the implementations in `res`. We use contract matching on the `super_` methods in the same was as in `pow_generate`. As with `pow`, the above two trait literals can be statically verified, and so we guarantee that the result of `combine_accounts` is correct.

5 Conclusion and Future Work

As we have shown with our examples, iterative composition is quite powerful, we believe we can use this power together with our approach to perform complex code transformations over statically verified code, without needing further verification.

For a trait literal to be statically verified, the verifier needs to see the contracts of any methods called within it, this precludes such literals from referencing code that has yet to be meta-programmed. This hampers the common technique of using meta-programming to add methods to a trait: the input trait cannot call such additional methods, since they have not been generated yet. A workaround to this is to provide the declaration and contracts (but not implementation) of any method a trait literal needs to call. This however lessens some of the advantages of meta-programming.

We would like to prove that our contract matching `extend` function is 'sound', as well as investigating the soundness of other composition operators (such as ones that make traits reference different classes, with possibly different method contracts).

We leave the complicated task of actually implementing static verification for an iterative composition language, such as '42' [1], to future work. In particular, it would be worthwhile investigating

how contracts of meta-operations (like our `pow_generate` and `combine_accounts` functions) could be specified and verified.

References

[1] *42 - The definitive answer to design, code and everything*. `http://l42.is/`.

[2] Reza Ahmadi, K. Rustan M. Leino & Jyrki Nummenmaa (2015): *Automatic Verification of Dafny Programs with Traits*. In: *Proceedings of the 17th Workshop on Formal Techniques for Java-like Programs*, FTfJP '15, ACM, New York, NY, USA, pp. 4:1–4:5, doi:10.1145/2786536.2786542. Available at `http://doi.acm.org/10.1145/2786536.2786542`.

[3] Hrshikesh Arora, Marco Servetto & Bruno C. d. S. Oliveira (2019): *Separating Use and Reuse to Improve Both (forthcoming)*. *The Art, Science, and Engineering of Programming*.

[4] Ferruccio Damiani, Johan Dovland, Einar Broch Johnsen & Ina Schaefer (2011): *Verifying Traits: A Proof System for Fine-grained Reuse*. In: *Proceedings of the 13th Workshop on Formal Techniques for Java-Like Programs*, FTfJP '11, ACM, New York, NY, USA, pp. 8:1–8:6, doi:10.1145/2076674.2076682. Available at `http://doi.acm.org/10.1145/2076674.2076682`.

[5] 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. Available at `http://doi.acm.org/10.1145/197320.197383`.

[6] Bertrand Meyer (1988): *Object-Oriented Software Construction*, 1st edition. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

[7] Michael J. O'Donnell (1982): *A Critique of the Foundations of Hoare Style Programming Logics*. *Commun. ACM* 25(12), pp. 927–935, doi:10.1145/358728.358748. Available at `http://doi.acm.org/10.1145/358728.358748`.

[8] Marco Servetto & Elena Zucca (2010): *MetaFJig: A Meta-circular Composition Language for Java-like Classes*. In: *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, ACM, New York, NY, USA, pp. 464–483, doi:10.1145/1869459.1869498. Available at `http://doi.acm.org/10.1145/1869459.1869498`.

[9] Marco Servetto & Elena Zucca (2013): *A Meta-circular Language for Active Libraries*. In: *Proceedings of the ACM SIGPLAN 2013 Workshop on Partial Evaluation and Program Manipulation*, PEPM '13, ACM, New York, NY, USA, pp. 117–126, doi:10.1145/2426890.2426913. Available at `http://doi.acm.org/10.1145/2426890.2426913`.

[10] Tim Sheard & Simon Peyton Jones (2002): *Template meta-programming for Haskell*. In: *Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*, ACM, pp. 1–16.

[11] T. Stephen Strickland, Sam Tobin-Hochstadt, Robert Bruce Findler & Matthew Flatt (2012): *Chaperones and Impersonators: Run-time Support for Reasonable Interposition*. In: *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, ACM, New York, NY, USA, pp. 943–962, doi:10.1145/2384616.2384685. Available at `http://doi.acm.org/10.1145/2384616.2384685`.

[12] Dana N. Xu, Simon Peyton Jones & Koen Claessen (2009): *Static Contract Checking for Haskell*. In: *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '09, ACM, New York, NY, USA, pp. 41–52, doi:10.1145/1480881.1480889. Available at `http://doi.acm.org/10.1145/1480881.1480889`.

[13] Karen Zee, Viktor Kuncak & Martin C. Rinard (2009): *An Integrated Proof Language for Imperative Programs*. In: *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, ACM, New York, NY, USA, pp. 338–351, doi:10.1145/1542476.1542514. Available at `http://doi.acm.org/10.1145/1542476.1542514`.