Termination == True

Jan Bessai[nope][0000−0000−000−0000],
Marco Servetto[1][0000−0003−1458−2868], and Julian Mackay[1][0000−0003−3098−3901]

[1] Victoria University of Wellington, Kelburn, 6012, Wellington, New Zealand
{isaac, marco.servetto}@ecs.vuw.ac.nz
[2] The Australian National University, Canberra, 2600, ACT, Australia
alex.potanin@anu.edu.au

Abstract. How should a verification system work on a pure OO language, where objects are black boxes that can only answer to messagess and where there are no primitive types whose sementic is shared between the verification language and the object language? We argue that termination is the only observable property in such a setting, and thus termination should be the core of such a verification system.

Keywords: termination calculus· feaderweight java· static verification· well founded contracts

1 Introduction

In traditional OO verification, specifications have to be well founded. For example, consider the following postcondition: `@Post \result.size()>this.size()` Most sound traditional OO verification language would require the method size() to be total and pure: that is, it must always terminate, be deterministic and not mutate the state or do I/O. While the requirements of purity seems clearly needed to use that behaviour as specification, the totality requirement seems to be overly restrictive: We could, for example, conservatively say that such a postcondition is false if the call `\result.size()` would not terminate.

We experimented with this idea: an OO setting where non termination means false. We discovered that all of the sources of falseness we encountered in our experiments could be encoded by non termination: we could simply make a function that loops if the parameter is not of the expected value.

This pushed us to attempt a verification system based on proving termination alone, and encode all of the other properties on top. That is, we propose a specification paradigm that unifies the underlying object oriented language with the specification language, where termination is centred as the source of truth in the language.

This mindset is nicely aligned with the semantics of pure OO languages: In a pure OO setting, everything should be an object, and the only meaningful operation should be the single dispatch method call. That is, objects are black boxes and they can communicate with each other by message passing; where other black box objects will be parameters and results. In this context, where

there is no such a thing as a primitive boolean, numbers, reference equality, what does it mean to '*specify a property*'? Objects are black boxes, and we can only observe them by calling methods, but the result of such methods are other black boxes. The only observable event seems to be '*termination*'. [3]

It is tempting to allow the specification language to express properties that are not visible in the execution. This often happens in traditional oo specification languages, where the specification lifts values out of the OO world and interprets them as mathematical objects that can be specified with operations that are not part of the language semantic.

We believe this is the root of many difficulties in OO verification. [4] For example, consider a method returning an instance of an interface `Foo`. This method may return concrete instances of `Foo`, but it would not make sense for the specification to restrict what those instances should be. For example, from the perspective of the specification, a concrete foo `CFoo` should be indistinguishable from a wrapped `D(CFoo)`, if they behave the same for all possible observations.

A similar situation happens in foundational math when we want to talk about properties of natural numbers without being able to investigate if they are incarnated as von Neumann numbers or with the church encoding.

Primitive types, instanceof checks and pointer equalities break this intuition, and that is why they are often considered not pure OO operations. However, traditional verification is built around those operations; as described below, traditional OO verification specifies the abstract behaviour of an operation, by describing its concrete implementation. This requires specifiable software to abandon OO principles.

For example we could specify that a method `getOldestPerson()` returns an instance of a Person class containing a primitive `int` field, whose value can be lifted to represent a mathematical natural number that has certain relations with other numbers stored in similar objects. That is, the specification encourages us to describe the structure of the result, as a tree where nodes are records and leaves are primitive data types, or other data types whose values can be lifted to represent a well understood mathematical abstraction. This is such a pronounced pattern/encouragement by those logics that when the data types do not follow this pattern closelly, *ghost state* is used to enrich those datatypes to make them fit this specification paradigm. We call this paradigm '*Structural specification of the result*'

Our approach naturally encourages a different specification paradigm, that we call '*Behavioural description of result*'; where we just specify that objects behave correctly in certain situations, without reference to the concrete representation of the object. This paper shows how this can be achieved by simply checking termination on well crafted observations, and that verified programs are able to employ useful object oriented principles and patterns.

[3] Non termination, on the other hand, can not be observed.
[4] Arguably true for verification of other paradigms too.

2 The two main observations

There are two independent observations at play here:

1. We can use termination to prove arbitrary decidable properties.
2. We argue that the only thing we should be allowed to observe in a pure OO language is termination.

2.1 We can use termination to prove arbitrary decidable properties

That is, we can express interesting properties about specific programs without the need to rely on dependent types. Usually, without dependent types we can only express general properties holding for all well typed programs. Consider for example the trivial extension of simply typed lambda calculus with conditionals natural numbers, booleans, their operations and a call by value fixpoint (intrinsically enabling a looping expression).

In context $\Gamma =$ `{ a:Num,b:Num,c:Num }` the term

```
if ( ((a+b)+c)==(a+(b+c))  )
      then true
      else  loop
```

Is typeable by Bool and will reduce to a boolean value if and only if the property ((a+b)+c)==(a+(b+c)) holds after substitution of a, b, and c with any possible well typed (closed) value of type Num. [5] Verifying reduction to a value thereby proves the desired property. This proof technique, verifying reduction to a value has the more general schema as follows:

1. Fix some deterministic language (e.g. lambda calculus) and interpretation $[|\_|]$ (e.g. logical statements).
2. In this setting, suppose you want to show some property P of some operation M:
   $P([|M|],[|X|]) = [|T|]$ for all closed well-typed values X and some value T representing truth.
3. Now first find Q, s.t. $[|Q|] = P$ (program Q encodes P). In the example above P is associativity, M is $(\_ + \_)$ and
   Q is $\lambda M, a, b, c.M(M(a, b), c) = M(a, M(b, c))$.
4. If the semantic you have is sound, we have $Q(M, X)$ reduces to T implies $P([|M|],[|X|]) = [|T|]$
5. Now construct F where $F(Y)$ reduces to some value V if Y reduces to T and $F(Y)$ diverges otherwise.
6. Show that for all closed well-typed values X there exists V, s.t. $F(Q(M, X))$ reduces to V.
7. From the properties of F conclude that for all those X, $Q(M, X)$ reduces to T and from soundness (4) conclude that $Q(M, X)$ reduces to T closes the proof.

[5] For a more functional setting,
you can consider `fix (s:Bool->Bool,a:Bool. s a) true` instead.

The use of F allows us to abstract from the concrete interpretation of truth. That is, any property that is semicomputable in the chosen language can be expressed as a termination property.

2.2 We argue that the only thing we should be allowed to observe in a pure OO language is termination; that is: observing more breaks the OO abstraction/model

For example, Hoare verification is the foundation for all/most of the verification of imperative languages, but values in the Hoare logic are predefined/primitive and their semantics are transparently lifted at the logic layer. The verification logic then states that if the input state respects certain logic predicates, then the output state also respects some logic predicates. This requires the values of the state to be exposed on the logic layer. Thus non primitive values either can not be used, or must be interpreted as a composition of values that already have a meaning in the logic layer. The core of the OO abstraction is the method call (message passing). That is, an object is not a record of fields. An object is a black box that can answer to the message '`getX()`' in a way that (informally) shows the object knows about an x coordinate.

The only thing you can do with an object is calling methods on it. In a pure OO setting, there are no primitive types, so every method just returns another black box object, that in turn can only be observed by calling methods on it.

Attempting to verify pure OO programs using traditional verification techniques violates the OO paradigm; the black boxes are open and interpreted as a composition of primitive data types. Instead, by simply observing termination, we can do proofs while preserving the pure OO abstraction: not only we can do the reasoning in the OO paradigm, but the property itself is specified as an OO expression.

3 Conclusion

We would like some feedback about the ideas expressed in this short document. Overall, we hope that levering on termination as the source of truth we could side step well foudness of contracts and many other subtile problems in OO verification, while forging a more cohesive and consistent meta theory.