Modular Termination Verification of Single-Threaded and Multithreaded Programs

BART JACOBS, KU Leuven
DRAGAN BOSNACKI, Eindhoven University of Technology
RUURD KUIPER, Eindhoven University of Technology

We propose an approach for the modular specification and verification of total correctness properties of object-oriented programs. The core of our approach is a *specification style* that prescribes a way to assign a *level expression* to each method such that each callee's level is below the caller's, even in the presence of dynamic binding. The specification style yields specifications that properly hide implementation details. The main idea is to use *multisets of method names* as levels, and to associate with each object levels that abstractly reflect the way the object is built from other objects. A method's level is then defined in terms of the method's own name and the levels associated with the objects passed as arguments.

We first present the specification style in the context of programs that do not modify object fields. We then combine it with separation logic and abstract predicate families to obtain an approach for programs with heap mutation. In a third step, we address concurrency, by incorporating an existing approach for verifying deadlock-freedom of channels and locks. Our main contribution here is to achieve information hiding by using the proposed termination levels for lock ordering as well. Also, we introduce *call permissions* to enable elegant verification of termination of programs where threads cause work in other threads, such as in thread pools or fine-grained concurrent algorithms involving compare-and-swap loops.

We explain how our approach can be used also to verify liveness of non-terminating programs.

CCS Concepts: •**Theory of computation** → **Program specifications;** *Pre- and post-conditions; Program verification;*

General Terms: Design, Algorithms, Performance

Additional Key Words and Phrases: Program termination, modular program verification, module specifications, separation logic

ACM Reference Format:
Bart Jacobs, Dragan Bosnacki, and Ruurd Kuiper, 2015. Modular termination verification of single-threaded and multithreaded programs. *ACM Trans. Program. Lang. Syst.* 0, 0, Article 0 (0), 61 pages.
DOI: 0000001.0000001

1. INTRODUCTION

Software plays a significant role in ever more areas of human activity, and in ever more applications with high reliability requirements, where failures caused by software defects could affect human safety, system security, or mission success. In many cases, software verification through testing provides insufficient assurance of the absence of defects. Formal program verification, where the program's source code is analyzed to obtain mathematical certainty that all of a program's possible executions satisfy certain formalized requirements, is in such cases a promising complementary approach.

This work was supported by EU project ADVENT and by project G.0058.13 of the Research Foundation - Flanders (FWO).
Author's addresses: B. Jacobs, Department of Computer Science, KU Leuven; D. Bosnacki and R. Kuiper, Computing Science Department, Eindhoven University of Technology.

Formal program verification approaches can be roughly divided into two categories: *whole-program* approaches and *modular* approaches. In a whole-program approach, a complete, closed program must be available before any results can be obtained. In such an approach, a method call is verified by verifying the method's implementation, taking into account the particular context of the call. If a call is dynamically bound, all potential callees are inspected. A major advantage of a whole-program approach is that typically, besides the source code itself and a formalization of the overall correctness property being verified, little or no additional user input is required. A disadvantage is that modifying any part of the program invalidates the results obtained.

In a modular approach, on the other hand, the object of verification is not whole programs, but program *modules*, coherent sets of classes, developed independently, that satisfy a well-defined *module specification*. A module need not be *closed*: it may refer to classes not defined by the module itself, but by other modules which it *imports*. A module should use only those classes and methods from an imported module that are specified as *exported* (or *public*) by that module's specification; only those elements are guaranteed to still be present in future versions of the imported module. *Verifying* a module means proving that it satisfies its specification, assuming that imported modules satisfy theirs.

In a modular approach, a method call is verified by assuming that it satisfies the called method's specification. If the call is dynamically bound, only the specification of the statically resolved method is considered. Separately, it is checked that each overriding method satisfies the specification of each method it overrides. In this approach, after modifying a method, it is sufficient to check that the method still satisfies its specification, to ensure that any properties verified previously still hold. Once stable module specifications have been established, modular verification is more scalable since it enables distributing the verification effort across multiple teams and reusing verification results.

A major issue in modular verification is the question of the *specification approach*: what should a module specification look like? The approach should be sufficiently *expressive* to be able to capture precisely the dependencies that a module's clients (i.e. those other modules that import the module) may have upon it, but it should also be sufficiently *abstract* so that proper *information hiding* is achieved: a module's specification should not unnecessarily constrain the current implementation or its future evolution. Any modification that does not break clients should be allowed.

In recent years, great progress has been made in specification approaches for *partial correctness*, the property that the program never reaches a incorrect state. However, we are not aware of any existing approach for modular specification of *total correctness* of object-oriented programs, the property that additionally the program *terminates*.

In this article, we propose such an approach.

For sequential programs, termination means absence of infinite loops and absence of infinite recursion. In the remainder of this article, we assume the program has no loops; this can be achieved by turning each loop into a recursive method.

The main difficulty in defining a specification approach for modular verification of termination of sequential object-oriented programs, is in dealing with dynamic binding. This can be understood as follows. Firstly, we assume that the module import graph is acyclic. (That is, no module directly or indirectly imports itself. If there is such a cycle, its members should be consolidated into a single module.) This assumption allows us to think of the program as consisting of *layers*, such that modules only import modules from lower layers. In the absence of dynamic binding, all method calls are either internal within a module, or descend into a lower layer. (Indeed, a module should refer only to the classes and interfaces it defines itself and the ones it imports.) Therefore, any cycle in the call graph is necessarily internal to a module, so proving

```
interface Func {
  num apply(num x)
}
class Util {
  static num deriv(Func f, num x)
  { f.apply(x + 1) − f.apply(x) }
}

class ZeroFunc imports Util implements Func {
  num apply(num x) { 0 }
  static void main()
  { Util.deriv(new ZeroFunc(), 42) }
}
class Loopy imports Util implements Func {
  num apply(num x) { Util.deriv(this, x) }
  static void main()
  { Util.deriv(new Loopy(), 42) }
}
```

Fig. 1. An example program without intra-module recursion but with infinite inter-module recursion

absence of infinite recursion is not directly a module specification issue. Indeed, the number of inter-module calls in a call stack below a given call is bounded by the *static depth* of the caller, i.e. the number of layers below it.

In contrast, in the presence of dynamic binding, the number of inter-module calls in a chain of calls is not bounded, so absence of infinite intra-module recursion does not imply absence of infinite recursion. This is the main problem addressed in this article.

For example, consider the program of Fig. 1. In the examples of this article, for simplicity, we conflate classes and modules: each module consists exactly of a single class. (We do not consider interfaces to be part of modules.) Each class declares the classes it imports in its **imports** clause. If a class mentions another class as the type of a variable, as the target of a static method call, or to create an instance of it, it must import it. In this program, then, we have three modules: Util, ZeroFunc, and Loopy, and ZeroFunc and Loopy each import Util. Util does not import any module. Notice that the import graph is acyclic, and that none of the modules contain intra-module method calls; nonetheless, whereas method ZeroFunc.main correctly computes the derivative of the constant function 0 at argument 42, method Loopy.main performs infinite inter-module recursion between methods Util.deriv and Loopy.apply.

In this article, we propose:

— a *program logic* for expressing module specifications that specify total correctness properties of methods exported by these modules, such as termination of method ZeroFunc.main;
— a corresponding *proof system* for verifying modules against their specifications that is *sound*, i.e. if a proof exists in this proof system for each module of a program, then each method satisfies its specified total correctness properties, implying that the proof system does not allow the verification of a specification that states that method Loopy.main terminates; also, the proof system is *modular*, meaning that each module's proof uses only the *specifications*, not the *implementations* of imported modules, and does not depend at all on other modules, such that the proof of Util.deriv does not depend on the existence or non-existence, or the content, of modules ZeroFunc and Loopy, and the proof of ZeroFunc.main uses only the specification of Util.deriv and not its implementation; and
— a *specification style* for writing specifications in this program logic such that they perform proper information hiding, e.g. such that method Util.deriv's specification is satisfied equally by alternative implementations that are more complex (and more accurate).

The remainder of this article is structured as follows. In Section 2, we introduce the core of our approach: an approach for assigning *level expressions* to methods that supports dynamic binding and that performs proper information hiding. This approach is initially presented in the context of programs that do not modify object fields. In Section 3, this approach is combined with separation logic to support programs that do mutate the heap. In Section 4, we show how to apply the approach to modularly verify termination of concurrent programs. In Section 5, we show how to prove liveness of non-terminating programs. In Section 6, we discuss the tool support we implemented for our approach. Finally, we discuss some ways to relax our approach in Section 7, as well as related work in Section 8, and we offer a conclusion in Section 9. A formalization of the approach is provided in the Appendix.

An earlier version of this work appeared at ECOOP 2015 [Jacobs et al. 2015a]. For the material of Sections 4 and 5, that paper referred to its accompanying technical report [Jacobs et al. 2015b]. Also, the presentation has been greatly improved througout: the various elements are introduced more incrementally and motivated better, there are many additional examples, and the material of Sections 2.2.4 and 2.2.5 is new.

2. LEVELS

To present the essence of our modular termination verification approach, in this section we consider only sequential programs. Furthermore, we adopt the standard module specification formalism where each method of a program is associated with an expression that evaluates to an element from a universe of *levels*, equipped with a *well-founded order*[1]. Such an expression, which may depend on the method's parameters and on the program state, is commonly known as a ranking function, a measure, a variant, or a decreases clause. Additionally, we adopt the standard notion of module correctness where a module is correct if for each method call, the callee's level (i.e. the level value obtained by evaluating the callee's level expression under the callee's arguments and pre-state) is below the caller's. Soundness of the approach then follows trivially: an infinite call chain would imply an infinite descending chain in the universe of levels.

The main contribution of this article, then, lies not in our choice of specification formalism or notion of module correctness, but in our proposed choice for the universe of levels and for which particular level to assign to each particular method. Indeed, for any given program that terminates, there are infinitely many possible ways to choose a universe of levels and to assign levels to the methods of the program such that each module is correct; however, only some of those assignments lead to module specifications that are *abstract*. We say a module specification S is abstract if it does not distinguish between module implementations that cannot be distinguished by clients. More specifically, S is abstract if it is closed under contextual refinement, i.e. if an implementation I satisfies S and an implementation I' contextually refines I (meaning that if a client C composed with I terminates, then C composed with I' terminates), then I' satisfies S.

To illustrate this, consider the program in Figure 2(a). Assume that each method is declared by a separate class, such that the class of method vectorSize imports the class of method sqrt, and the class of method main imports the class of method vectorSize. Notice that each module is correct: at each call, the level decreases. However, this choice of levels is not abstract. In particular, it should always be possible to refactor a method's implementation so that it reuses functionality from a library[2], without

[1]A well-founded order is an order that admits no infinite descending chains, or, equivalently, where each nonempty set S has a minimal element (an element $x \in S$ such that $\nexists y \in S.\ y < x$).
[2]We assume the library does not itself use the method being refactored for its own implementation, which would cause a cycle in the module import graph.

```
num average(num x, num y)
  level 0
{ (x + y)/2 }

num sqrt(num x)
  level 0
{ (1 + x)/2 }

num vectorSize(num x, num y)
  level 1
{ sqrt(x · x + y · y) }

void main()
  level 2
{ assert 0 ≤ vectorSize(3, 4) }
```

(a)

```
num average(num x, num y)
  level 0
{ (x + y)/2 }

num sqrt(num x)
  level 0
{ average(1, x) }

num vectorSize(num x, num y)
  level 1
{ sqrt(x · x + y · y) }

void main()
  level 2
{ assert 0 ≤ vectorSize(3, 4) }
```

(b)

Fig. 2. (a) A choice of levels that is not abstract; (b) A refactoring that it disallows. Elements shown in red violate module correctness.

breaking the method's specification. However, if we introduce a call of method average into the body of method sqrt, we cannot make this call pass the level check without changing the specifications of the modules involved.

In the remainder of this section, we introduce our proposed choice for the universe of levels, and our proposed approach for choosing which levels to assign to the methods of a program. We do so in three steps. In the first step (Section 2.1), we introduce a simplified version that works for programs that have only *upcalls*; it uses *method names* as levels. In the second step (Section 2.2), we extend this approach to support dynamic binding, using *multisets* of method names as levels. In the third step (Section 2.3), we add support for recursion to obtain the final approach, where we combine multisets of method names with *local levels*.

2.1. Upcalls Only

In this subsection, we consider only programs that perform only statically bound method calls (i.e. calls of static methods) and where the call graph has no cycles. All such programs necessarily terminate, but here we will look at how to prove this for a given program modularly, using abstract module specifications, within the modular verification framework defined above.

The example of Figure 2 illustrates that for any approach where each module's methods are associated with closed level expressions assigning levels from a fixed well-founded order, and for any two modules M_1 and M_2 that do not import each other, it has to be the case that either a refactoring of M_1 so that it uses (and therefore imports) M_2, or a refactoring of M_2 so that it uses M_1 requires changing at least one module's specification. Therefore, to allow both of these refactorings without changing module specifications, we must allow either the interpretation of level expressions or the order relation on levels to somehow depend on aspects of the way modules are implemented, without revealing such implementation aspects in the level expressions themselves.

One approach is to use *symbolic level constants* in level expressions, as illustrated in Fig 3. In this approach, a module specification can expose the existence of one or more symbolic level constants, without revealing their definition. The module's level

```
final int AVERAGE_LEVEL = 0
num average(num x, num y)
  level AVERAGE_LEVEL
{ (x + y)/2 }


final int SQRT_LEVEL = 0
num sqrt(num x)
  level SQRT_LEVEL
{ (1 + x)/2 }

final int VECTOR_SIZE_LEVEL =
  SQRT_LEVEL + 1
num vectorSize(num x, num y)
  level VECTOR_SIZE_LEVEL
{ sqrt(x · x + y · y) }

void main()
  level VECTOR_SIZE_LEVEL + 1
{ assert 0 ≤ vectorSize(3, 4) }
```

(a)

```
final int AVERAGE_LEVEL = 0
num average(num x, num y)
  level AVERAGE_LEVEL
{ (x + y)/2 }

final int SQRT_LEVEL =
  AVERAGE_LEVEL + 1
num sqrt(num x)
  level SQRT_LEVEL
{ average(1, x) }

final int VECTOR_SIZE_LEVEL =
  SQRT_LEVEL + 1
num vectorSize(num x, num y)
  level VECTOR_SIZE_LEVEL
{ sqrt(x · x + y · y) }

void main()
  level VECTOR_SIZE_LEVEL + 1
{ assert 0 ≤ vectorSize(3, 4) }
```

(b)

Fig. 3. (a) Using symbolic levels; (b) A refactoring that it allows.

expressions can mention these constants. A module's implementation must define each declared symbolic level constant in terms of the constants exposed by imported modules. Typically, a module's level constant is defined as one more than the maximum of the level constants exposed by imported modules.

An alternative approach is to use levels whose order depends on the module import graph. In particular, one could use *method names* as levels, ordered as follows: m_1 is less than m_2 if either m_1 and m_2 are defined in the same module and the definition of m_1 appears textually before the definition of m_2 in the module's program text, or there is a path in the "– imports –" graph from the module that defines m_2 to the module that defines m_1. In Figure 4, we use method names as levels to obtain abstract specifications of the modules of the example program. Notice that we simply assigned to each method m its own name m as its level.

Indeed, for programs that perform upcalls only, both specification approaches yield specifications that are trivially satisfied by current and future implementations of the program's modules[3]. For conciseness, in the rest of this article we use the second approach, where method names are used as levels. However, all of the ideas introduced in the rest of this article apply equally to the approach based on symbolic level constants.

The first approximation of the proposed specification style of this article is therefore as follows:

SPECIFICATION STYLE 1 (UPCALLS ONLY).

— *For the universe of levels, pick the set of* method names*, ordered by the module import graph and some appropriate intra-module ordering (such as textual order).*
— *Assign to each method m its own name m as its level.*

[3]Assuming that within a module, callees appear textually before callers.

```
num average(num x, num y)
  level average
{ (x + y)/2 }

num sqrt(num x)
  level sqrt
{ (1 + x)/2 }

num vectorSize(num x, num y)
  level vectorSize
{ sqrt(x · x + y · y) }

void main()
  level main
{ assert 0 ≤ vectorSize(3, 4) }
```

(a)

```
num average(num x, num y)
  level average
{ (x + y)/2 }

num sqrt(num x)
  level sqrt
{ average(1, x) }

num vectorSize(num x, num y)
  level vectorSize
{ sqrt(x · x + y · y) }

void main()
  level main
{ assert 0 ≤ vectorSize(3, 4) }
```

(b)

Fig. 4. (a) Using method names as levels; (b) A refactoring that it allows.

2.2. Dynamic Binding

2.2.1. Simple Objects. Let us now consider programs with dynamically bound method calls. Consider the example program of Figure 5. It defines the following elements:

— An interface Func for objects that represent mathematical functions on rational numbers.
— A utility class Util that provides a method deriv that computes a rough approximation of the derivative of the function given by a Func object f at a given argument x.
— A class ZeroFunc whose objects represent the constant function that equals zero everywhere.
— A main class Main that exercises classes Util and ZeroFunc.

In an initial attempt to modularly prove termination of this program, we applied the specification approach of the preceding subsection: we assigned to each method its own name as its level. Note that we will only use *class* method names, i.e. names of the form $C.m$, where C is a class name, as levels. We will not use interface method names $I.m$ or unqualified method names m as levels.[4] Note also that, as mentioned in Section 1, for modular verification of dynamically bound method calls, we will associate a specification with each interface method, verify calls of interface methods against these specifications, and verify each class method that implements an interface method against the interface method's specification. (We will not assign separate specifications to such class methods individually.) Therefore, we interpret the specification approach of the preceding subsection such that we should specify as the level of an interface method the name of the method to which calls of the interface method will be bound at run time. To do so, we will allow the use of the expression classOf(−) in **level** clauses. This way, we can specify the level for method Func.apply as classOf(this).apply.

Note that applying the specification style of the preceding subsection does *not* allow us to verify this program. Indeed, method deriv is not correct: we cannot establish

[4]This is a design decision, consistent with the decision of not considering interfaces to be part of modules, and therefore not being ordered by the module import graph. As the reader will see, this yields an elegant approach. However, we do not claim it is the only possible approach.

```
interface Func {
  num apply(num x)
    level classOf(this).apply
}
class Util {
  static num deriv(Func f, num x)
    level Util.deriv
  { f.apply(x + 1) − f.apply(x) }
}
class ZeroFunc implements Func { num apply(num x) { 0 } }
class Main imports Util, ZeroFunc {
  static void main()
    level Main.main
  { Util.deriv(new ZeroFunc(), 0) }
}
```

Fig. 5. A program with dynamically bound method calls. Method deriv's implementation is not allowed by its specification, since the calls of apply cannot be shown to satisfy the level constraint.

```
class Util {
  static num deriv(Func f, num x)
    level classOf(f).apply + 1
  { f.apply(x + 1) − f.apply(x) }
}
```

(a)

```
class Util {
  static num derivHelper(Func f, num x)
    level classOf(f).apply + 1
  { f.apply(x + 1) − f.apply(x) }
  static num deriv(Func f, num x)
    level classOf(f).apply + 1
  { Util.derivHelper(f, x) }
}
```

(b)

Fig. 6. (a) A specification that is not abstract; (b) A refactoring that is disallowed by it. The call of method derivHelper does not satisfy the level constraint.

that classOf(f).apply $<$ Util.deriv, and indeed, in the case where f is an instance of class ZeroFunc, this is not the case.

We could try to fix this by extending our universe of levels with levels of the form $m + n$, where m is a method name and n is a natural number, such that $m + n < m' + n'$ if $m < m'$ or $m = m'$ and $n < n'$. We could then assign to method deriv level classOf(f).apply $+ 1$, as shown in Figure 6(a). However, this specification would not be abstract: it would not allow us to introduce a static helper method Util.derivHelper and to call it in the body of method deriv (see Figure 6(b)).

Method deriv should be allowed to perform *both* upcalls (statically bound calls of lesser-named methods) and calls on f, and it should be able to pass f as an argument in upcalls, such that callees can call f themselves.

To achieve this, we propose to use as levels not method names, but *multisets* of method names, ordered by *multiset order* [Dershowitz and Manna 1979]. For multisets A and B, we say $A < B$ iff $A \neq B$ and there exist multisets C, A', B' such that $A = C \uplus A'$ and $B = C \uplus B'$ and $\forall x \in A'.\ \exists y \in B'.\ x < y$. In other words, starting from a given multiset, a lesser multiset is obtained by replacing one or more

```
interface Func {
   num apply(num x)
      level {[classOf(this).apply]}
}
class Util {
   static num derivHelper(Func f, num x)
      level {[Util.derivHelper, classOf(f).apply]}
   { f.apply(x + 1) − f.apply(x) }
   static num deriv(Func f, num x)
      level {[Util.deriv, classOf(f).apply]}
   { Util.derivHelper(f, x) }
}
class ZeroFunc implements Func { num apply(num x) { 0 } }
class Main imports Util, ZeroFunc {
   static void main()
      level {[Main.main]}
   { Util.deriv(new ZeroFunc(), 0) }
}
```

Fig. 7. Specifications for the example program using multisets of method names

elements by any number of lesser elements. In particular, we have $\{\!\![\text{ZeroFunc.apply}]\!\!\} < \{\!\![\text{Util.derivHelper}, \text{ZeroFunc.apply}]\!\!\} < \{\!\![\text{Util.deriv}, \text{ZeroFunc.apply}]\!\!\} < \{\!\![\text{Main.main}]\!\!\}$.[5]

This leads us to the specifications in Figure 7. These allow us to verify the program.

2.2.2. Complex Objects. However, we are not done. In particular, the specification for Func.apply is not abstract. Indeed, consider the case where the Func object has fields holding references to other objects, and the apply method performs calls on these, as in class Plus1Func in Figure 8. Method Plus1Func.apply is not correct: we cannot establish $\{\!\![\text{classOf(this.f).apply}]\!\!\} < \{\!\![\text{classOf(this).apply}]\!\!\}$. The case of Plus1Func.apply is similar to that of Util.deriv: it should be able *both* to perform upcalls and to call f.apply. And this is true recursively: the object pointed to by Plus1Func's f field may itself refer to more objects.

The solution we propose is to associate levels with *objects*, and to refer to those levels in the level clauses of the objects' methods. For example, to address the problem highlighted in Figure 8, we impose upon each class that implements interface Func the obligation to define an applyLevel, a multiset of method names, for each of its objects, and we define the level of method apply as being equal to its receiver's applyLevel. For objects of class ZeroFunc, we define applyLevel $= \{\!\![\text{ZeroFunc.apply}]\!\!\}$, and for objects of class Plus1Func, we define applyLevel $= \{\!\![\text{Plus1Func.apply}]\!\!\} \uplus \mathit{fal}$, where fal is the applyLevel of the object referred to by the Plus1Func object's f field. This allows us to verify ZeroFunc's and Plus1Func's apply methods.

For simplicity, in this section we realize the association of levels with objects technically by means of *ghost fields* (see Figure 9). This works well in the absence of heap mutation; therefore, in this section, we assume that all fields are immutable. In the next section, to show that our approach is fully compatible with heap mutation, we combine our approach with separation logic and we use abstract predicate families instead of ghost fields to associate levels with objects.

[5]The usefulness of multisets and multiset order for proving termination has long been recognized [Dershowitz and Manna 1979]. It was pointed out to us by K. Rustan M. Leino. We are not aware of more specific prior work for our particular application of the idea.

```
interface Func {
   num apply(num x)
      level ⦃classOf(this).apply⦄
}
class Plus1Func(Func f) implements Func {
   num apply(num x) { f.apply(x) + 1 }
}
class Main imports Util, ZeroFunc, Plus1Func {
   static void main()
   { Util.deriv(new Plus1Func(new ZeroFunc()), 0) }
}
```

Fig. 8. The specification of method apply disallows complex objects. We declare a class's fields in a parenthesized list after the class name. (Classes Util and ZeroFunc are unchanged from Figure 7 and not shown.)

```
interface Func {
   abstract ghost_field MethodBag applyLevel
   num apply(num x)
      level this.applyLevel
}
class Util {
   static num deriv(Func f, num x)
      level ⦃Util.deriv⦄ ⊎ f.applyLevel
   { f.apply(x + 1) − f.apply(x) }
}
class ZeroFunc implements Func {
   ghost_field MethodBag applyLevel = ⦃ZeroFunc.apply⦄
   num apply(num x) { 0 }
}
class Plus1Func(Func f) implements Func {
   ghost_field MethodBag applyLevel = ⦃Plus1Func.apply⦄ ⊎ f.applyLevel
   num apply(num x) { f.apply(x) + 1 }
}
class Main imports Util, ZeroFunc, Plus1Func {
   static void main()
      level ⦃Main.main⦄
   {
      Func f1 := new ZeroFunc();
      {f1.applyLevel = ⦃ZeroFunc.apply⦄}
      Func f2 := new Plus1Func(f1);
      {f2.applyLevel = ⦃Plus1Func.apply, ZeroFunc.apply⦄}
      Func f3 := new Plus1Func(f2);
      {f3.applyLevel = ⦃2 · Plus1Func.apply, ZeroFunc.apply⦄}
      {⦃Util.deriv, 2 · Plus1Func.apply, ZeroFunc.apply⦄ < ⦃Main.main⦄}
      Util.deriv(f3, 0)
   }
}
```

Fig. 9. Verifying termination in the presence of complex objects by associating levels with objects. In this section, for simplicity, ghost fields are used to realize this association.

Interface Func declares an abstract ghost field of type MethodBag (multiset of method names) named applyLevel. This imposes the obligation upon each class that implements interface Func to define a ghost field of the same name and type. This ghost field is used in the specification of methods Func.apply and Util.deriv and in the definition of ghost field Plus1Func.applyLevel.

The specification of method deriv shows the typical pattern for a static method that takes an object as an argument: the method's level is the multiset union of its own name and a level associated with the argument object. Since f.applyLevel $<$ $\{\!\{$Util.deriv$\}\!\} \uplus$ f.applyLevel, the method verifies.

To verify method main, after creating ZeroFunc instance f1, we look at the definition of ZeroFunc.applyLevel to conclude that we have f1.applyLevel $=$ $\{\!\{$ZeroFunc.apply$\}\!\}$. After applying similar reasoning for f2 and f3, we can conclude that the Util.deriv call is correct.

2.2.3. Abstract Object Construction. We have now successfully verified the example program of Figure 9; however, to verify method main, we looked at the definitions of ghost fields ZeroFunc.applyLevel and Plus1Func.applyLevel to derive the value of f1, f2, and f3's applyLevel from the values of these objects' fields. A more modular approach is to shield method main from the internal layout of these objects by having it call factory methods, and by having the postconditions of these factory methods describe the resulting objects abstractly using the applyLevel ghost field, as in Figure 10.

Notice that Plus1Func.create's declared level includes its argument's applyLevel: we simply applied the general specification pattern also exhibited by the specification of method Util.deriv in Figure 9. This allows future implementations of Plus1Func.create to call f.apply.

In order to allow client code to call the created object's methods, a factory method should always specify an upper bound for the created object's levels. A suitable upper bound is the level of the factory method itself: this ensures that any client that can call the factory method can also call the created object's methods.

2.2.4. Sibling objects. In the example above, method Util.deriv takes a single object as an argument. Now, consider method Util.sum in Figure 11, which takes two Func objects and returns the sum of their values at a given argument value. Since the sum method should be able to call both f1.apply and f2.apply, we assign to it the level $\{\!\{$Util.sum$\}\!\} \uplus$ (f1.applyLevel $\sqcup$ f2.applyLevel), where $\ell_1 \sqcup \ell_2$ is an abbreviation for $\mathsf{max}(\ell_1, \ell_2)$. (To ensure that this is always well-defined, we establish a total order on levels by choosing an arbitrary, fixed total order on modules consistent with the module import graph.)

Note that while conceptual economy would suggest reusing the $\uplus$ operator and assigning level $\{\!\{$Util.sum$\}\!\} \uplus$ f1.applyLevel $\uplus$ f2.applyLevel to method sum, this would overly constrain client code: it would not allow us to verify method double (also in Figure 11), whereas our use of $\sqcup$ does (since $\ell \sqcup \ell = \ell$).

We use the same approach to define the levels of objects that are built on top of multiple existing objects: see class SumFunc in Figure 11.

2.2.5. Passing Objects as Arguments to Objects. Consider interface Set in Figure 12. Its method intersects takes another object as an argument. Its level is the multiset union of the receiver's level and the argument's level.

Notice that the specification of Util.intersects differs from that of Util.sum in Figure 11: the former takes the multiset union, and the latter the maximum of the two argument objects' levels. This reflects the fact that in Util.intersects, one of the argument objects performs calls on the other one, whereas in Util.sum, the argument objects do not call each other. Indeed, in our approach, the fact that a method causes argument objects to

```
class ZeroFunc implements Func {
  ghost_field MethodBag applyLevel = {[ZeroFunc.apply]}
  num apply(num x) { 0 }
  static ZeroFunc create()
    level {[ZeroFunc.create]}
    ens result.applyLevel < {[ZeroFunc.create]}
  { new ZeroFunc() }
}
class Plus1Func(Func f) implements Func {
  ghost_field MethodBag applyLevel = {[Plus1Func.apply]} ⊎ f.applyLevel
  num apply(num x) { f.apply(x) + 1 }
  static Plus1Func create(Func f)
    level {[Plus1Func.create]} ⊎ f.applyLevel
    ens result.applyLevel < {[Plus1Func.create]} ⊎ f.applyLevel
  { new Plus1Func(f) }
}
class Main imports Util, ZeroFunc, Plus1Func {
  static void main()
    level {[Main.main]}
  {
    Func f1 := ZeroFunc.create();
    {f1.applyLevel < {[ZeroFunc.create]}}
    Func f2 := Plus1Func.create(f1);
    {f2.applyLevel < {[Plus1Func.create, ZeroFunc.create]})}
    Func f3 := Plus1Func.create(f2);
    {f3.applyLevel < {[2 · Plus1Func.create, ZeroFunc.create]})}
    {{[Util.deriv]} ⊎ f3.applyLevel < {[Main.main]}}
    Util.deriv(f3, 0)
  }
}
```

Fig. 10. Abstract object construction

call each other cannot generally be hidden from clients. We conjecture that this would be the case in any other modular termination verification approach as well.

Notice also that in this example, interface Set associates a single level level with its objects, instead of declaring separate levels containsLevel and intersectsLevel for use in the corresponding methods' level clauses. While such fine-grained specifications may sometimes enable verifying more client programs, we anticipate that most often it will be sufficient to declare a single level per object.

We summarize the specification style for programs with dynamic binding that we propose in this article as follows:

SPECIFICATION STYLE 2.

— *For the universe of levels, pick the multisets of method names, ordered by multiset order, where method names are ordered as in Specification Style 1.*
— *Declare in each interface an association between each object of the interface and a multiset of method names, called the object's* level.
— *In each class C, define the level of an object o of the class in terms of the level of the class itself (i.e. the names of the methods of the class) and the levels of the objects of which o is composed. In particular, if the class does not cause component objects*

```
class Util {
  static num sum(Func f1, Func f2, num x)
    level {[Util.sum]} ⊎ (f1.applyLevel ⊔ f2.applyLevel)
  { f1.apply(x) + f2.apply(x) }
  static num double(Func f, num x)
    level {[Util.double]} ⊎ f.applyLevel
  { Util.sum(f, f, x) }
}
class SumFunc(Func f1, Func f2) implements Func imports Util {
  ghost_field MethodBag applyLevel = {[SumFunc.apply]} ⊎ (f1.applyLevel ⊔ f2.applyLevel)
  num apply(num x) { Util.sum(f1, f2, x) }
  static Func create(Func f1, Func f2)
    level {[SumFunc.create]} ⊎ (f1.applyLevel ⊔ f2.applyLevel)
    ens result.applyLevel < {[SumFunc.create]} ⊎ (f1.applyLevel ⊔ f2.applyLevel)
  { new SumFunc(f1, f2) }
}
class Main imports Util, ZeroFunc, Plus1Func, SumFunc {
  static void main()
    level {[Main.main]}
  {
    Func f1 := ZeroFunc.create();
    Func f2 := Plus1Func.create(f1);
    Func f3 := SumFunc.create(f1, f2);
    Util.sum(f2, f3, 0)
  }
}
```

Fig. 11. Sibling objects

$o' \in O$ to call each other, define the level of o as $\{\!\![C.m]\!\!\} \uplus (\bigsqcup_{o' \in O} o'.\mathsf{level})$, where m is the greatest non-static method of C. If the class does cause component objects to call each other, define the level of o as $\{\!\![C.m]\!\!\} \uplus (\biguplus_{o' \in O} o'.\mathsf{level})$. If particular component objects call only particular other component objects, then use an appropriate combination of $\sqcup$ and $\uplus$ to combine the levels of the component objects.

— *Define each static method $C.m$'s level as $\{\!\![C.m]\!\!\} \uplus d$, where d is the appropriate combination of the levels of the objects passed into $C.m$ as arguments with $\sqcup$ and $\uplus$, depending on whether $C.m$ causes particular argument objects to call each other or not, in the same way as above.*

— *Define each non-static method $C.m$'s level as $\mathsf{this.level} \uplus d$, where d is the combined level of the argument objects, as defined above.*

2.3. Recursion

Above, we have looked at programs where method calls cross module boundaries and descend into lower layers of abstraction, and we have denoted those layers by means of multisets of method names. In this subsection, we complete our approach by adding support for programs where some method calls stay within the current layer of abstraction and are performed not to invoke an abstraction but for computational, algorithmic reasons. We assume that the author of each such algorithm involved has proven termination of the algorithm by means of a ranking function that maps each state of the algorithm to an element of some well-founded set. We call the union of these well-founded sets the set of the *local levels* of the program, and we adopt as the

```
interface Set {
  abstract ghost_field MethodBag level
  bool contains(int x)
     level this.level
  bool intersects(Set other)
     level this.level ⊎ other.level
}
class Empty() implements Set {
  ghost_field MethodBag level = {Empty.intersects}
  bool contains(int x) { false }
  bool intersects(Set other) { false }
  static Set create()
     level {Empty.create}
     ens result.level < {Empty.create}
  { new Empty() }
}
class Insert(int elem, Set set) implements Set {
   ghost_field MethodBag level = {Insert.intersects} ⊎ set.level
  bool contains(int x) { x = elem ∨ set.contains(x) }
  bool intersects(Set other) { other.contains(elem) ∨ set.intersects(other) }
  static Set create(int elem, Set set)
     level {Insert.create} ⊎ set.level
     ens result.level < {Insert.create} ⊎ set.level
  { new Insert(elem, set) }
}
class Util {
  static bool intersects(Set s1, Set s2)
    level {Util.intersects} ⊎ s1.level ⊎ s2.level
  { s1.intersects(s2) }
}
```

Fig. 12. An interface method that takes another object as an argument

final choice for the set of levels the expressions of the form d or $d + \ell$, where d is a multiset of method names, ℓ is a local level, and we have the relationships $d < d + \ell$, $d < d' \Rightarrow d + \ell < d'$, and $\ell < \ell' \Rightarrow d + \ell < d + \ell'$.

Consider for example in Figure 13 a method that computes the Ackermann function, where (m, n) denotes the lexicographically ordered pairs, with m more significant than n. Note that the specification of method ackermannIter reveals information about its algorithmic behavior. That is why, to achieve abstract module specifications, recursive methods like ackermannIter should always be private to a module, and public wrappers like ackermann should be provided whose specification conforms to the standard pattern suggested above and therefore does not reveal any algorithmic information.

SPECIFICATION STYLE 3 (RECURSION).

— *For the universe of levels, pick the expressions of the form d or $d + \ell$, where d is a multiset of method names, ordered as in Specification Style 2, and ℓ is an element of some well-founded set of* local levels. *Define the order on levels by the rules $d < d + \ell$, $d < d' \Rightarrow d + \ell < d'$, and $\ell < \ell' \Rightarrow d + \ell < d + \ell'$.*

— *If a module is implemented internally using recursion, keep all recursive methods private to the module; expose their functionality through separate public methods that*

```
class Math {
  static int ackermannIter(int m, int n)
    req 0 ≤ m ∧ 0 ≤ n
    level {Math.ackermannIter} + (m, n)
  {
    if m = 0 then n + 1 else
      if n = 0 then
        ackermannIter(m − 1, 1)
      else
        ackermannIter(m − 1, ackermannIter(m, n − 1))
  }
  static int ackermann(int m, int n)
    req 0 ≤ m ∧ 0 ≤ n
    level {Math.ackermann}
  { ackermannIter(m, n) }
}
```

Fig. 13. Recursion

are not themselves recursive, and whose specifications therefore do not reveal internal recursion termination measures.

— *For public methods, use levels of the form d, as prescribed by Specification Style 2.*

Note, however, that our approach also supports recursion involving dynamically bound calls between mutually unknown modules, as shown by the example of Figure 14.

3. SUPPORTING MUTATION: SEPARATION LOGIC

In the preceding section, for simplicity we presented our specification approach in the context of programs that do not modify object fields. However, our approach is fully compatible with heap mutation. In this section, we show how to modularly verify total correctness of object-oriented programs that modify object fields, by combining the ideas of the preceding section with separation logic.

In Section 3.1, we motivate and introduce separation logic for modular reasoning about heap-mutating programs. In Section 3.2, we combine separation logic with dynamic binding, and we discuss the issue of logical consistency. In Section 3.3, we show how to combine these ideas with the ideas of Section 2 to verify total correctness.

3.1. Separation logic for partial correctness

The example program of Figure 15 illustrates the problem addressed by separation logic when reasoning about heap-mutating object-oriented programs. The program defines three classes: Cell, CellWrapper, and Main. Objects of class Cell represent a nonnegative integer, and so do objects of class CellWrapper. The difference between these two classes is that class Cell stores its value directly, whereas CellWrapper uses a Cell object to store its value. For both classes, we define an *abstract predicate* valid. In both cases, for an object o, o.valid(v) expresses that o satisfies its invariant and that it represents value v. We use these predicates in the specification of the methods, with the goal of enabling users of these classes to reason about method calls without needing to know the classes' internal implementation.

However, the specifications of Figure 15 do not actually achieve this purpose fully. Even though the assert command in method main is satisfied when the program is executed, the specification of method CellWrapper.create does not allow the author of

```
interface IsEvenFunc {
   abstract ghost_field MethodBag level
   bool isEven(IsOddFunc f, int n)
      req 0 ≤ n
      level (this.level ⊔ f.level) + n
      ens result = (n = 0 mod 2)
}
interface IsOddFunc {
   abstract ghost_field MethodBag level
   bool isOdd(IsEvenFunc f, int n)
      req 0 ≤ n
      level (this.level ⊔ f.level) + n
      ens result = (n = 1 mod 2)
}
class Util {
   static bool isEven(IsEvenFunc e, IsOddFunc o, int n)
      req 0 ≤ n
      level {|Util.isEven|} ⊎ (e.level ⊔ o.level)
      ens result = (n = 0 mod 2)
   { e.isEven(o, n) }
}
class MyIsEvenFunc implements IsEvenFunc {
   ghost_field MethodBag level = {|MyIsEvenFunc.isEven|}
   bool isEven(IsOddFunc f, int n)
   { if n = 0 then true else f.isOdd(this, n − 1) }
}
class MyIsOddFunc implements IsOddFunc {
   ghost_field MethodBag level = {|MyIsOddFunc.isOdd|}
   bool isOdd(IsEvenFunc f, int n)
   { if n = 0 then false else f.isEven(this, n − 1) }
}
class Main imports Util, MyIsEvenFunc, MyIsOddFunc {
   static void main()
      level {|Main.main|}
   { Util.isEven(new MyIsEvenFunc(), new MyIsOddFunc(), 42) }
}
```

Fig. 14. Dynamically-bound recursion. Note: since MyIsEvenFunc.isEven does not perform any upcalls, it would be sufficient in this example to define MyIsEvenFunc.level to be the empty multiset. However, for uniformity, we applied the general approach for defining object levels defined in Specification Style 2. (A similar remark applies to MyIsOddFunc.)

method main to conclude this, without looking at the body of method create. Indeed, if the body of method create were swapped with that of method createAlt, which satisfies exactly the same specification, then the assert command would no longer be satisfied.

The problem is that the author of main cannot conclude that the assertion $\mathsf{w.valid}(0)$ is preserved by the $\mathsf{c.incr}()$ call.

One approach for addressing this problem is to modify the specification of create so that it expresses that the set of memory locations associated with object result is disjoint from the set of memory locations associated with object cell, and to modify the specification of Cell.incr so that it expresses that this method modifies only the memory locations associated with the receiver object. If, furthermore, the author of main knows

```
class Cell(int value) {
  predicate valid(int value) = (this.value = value ∧ 0 ≤ value)
  int incr()
    req this.valid(v)
    ens this.valid(v + 1) ∧ result = v ∧ 0 ≤ result
  { int value := this.value; this.value := value + 1; value }
}
class CellWrapper(Cell cell) {
  predicate valid(int value) = cell.valid(value)
  int incr()
    req this.valid(v)
    ens this.valid(v + 1) ∧ result = v ∧ 0 ≤ result
  { cell.incr() }
  static CellWrapper create(Cell cell)
    req cell.valid(v)
    ens cell.valid(v) ∧ result.valid(v)
  { new CellWrapper(new Cell(cell.value)) }
  static CellWrapper createAlt(Cell cell)
    req cell.valid(v)
    ens cell.valid(v) ∧ result.valid(v)
  { new CellWrapper(cell) }
}
class Main {
  static void main() {
    Cell c := new Cell(0);
    {c.valid(0)}
    CellWrapper w := CellWrapper.create(c);
    {c.valid(0) ∧ w.valid(0)}
    c.incr();
    {c.valid(1) ∧ w.valid(0)}
    assert w.incr() = 0
  }
}
```

Fig. 15. The problem of method effect framing in the presence of aliasing

that the truth of assertion $o.\mathsf{valid}(v)$ depends only on the values of the memory locations associated with o, then they can conclude that the assert command is satisfied.

A concise and elegant way to achieve this, is by using separation logic [O'Hearn et al. 2001; Parkinson and Bierman 2005]. In separation logic, assertions are interpreted under *partial heaps*, i.e. heaps that assign values to only a subset of the set of all fields of all allocated objects. To ensure that each separation logic assertion has a well-defined meaning when interpreted under any partial heap, direct field dereferences, such as $o.f = v$, are not allowed. Rather, the only way to refer to the value of a field in separation logic is using a *points-to assertion* $o.f \mapsto v$, which holds under a partial heap h if $o.f$ is in the domain of h and its value is v. Correspondingly, the meaning of a separation logic method specification is as follows: if the pre-state heap h can be written as the disjoint union of some heap h_1 that satisfies the precondition and some *frame* h_{F}, then the post-state heap h' can be written as the disjoint union of some heap h_2 that satisfies the postcondition and the same frame h_{F}. In other words, the method modifies only the memory locations asserted by the precondition. The final ingredient

```
class Cell(int value) {
  predicate valid(int value) = (this.value ↦ value ∧ 0 ≤ value)
  int incr()
    req this.valid(v)
    ens this.valid(v + 1) ∧ result = v ∧ 0 ≤ result
  { int value := this.value; this.value := value + 1; value }
}
class CellWrapper(Cell cell) {
  predicate valid(int value) = cell.valid(value)
  int incr()
    req this.valid(v)
    ens this.valid(v + 1) ∧ result = v ∧ 0 ≤ result
  { cell.incr() }
  static CellWrapper create(Cell cell)
    req cell.valid(v)
    ens cell.valid(v) ∗ result.valid(v)
  { new CellWrapper(new Cell(cell.value)) }
  static CellWrapper createAlt(Cell cell)
    req cell.valid(v)
    ens cell.valid(v) ∧ result.valid(v)
  { new CellWrapper(cell) }
}
class Main {
  static void main() {
    Cell c := new Cell(0);
    {c.valid(0)}
    CellWrapper w := CellWrapper.create(c);
    {c.valid(0) ∗ w.valid(0)}
    c.incr();
    {c.valid(1) ∗ w.valid(0)}
    assert w.incr() = 0
  }
}
```

Fig. 16. The problem of method effect framing in the presence of aliasing, addressed using separation logic

needed is the *separating conjunction*: the assertion $P * Q$ holds under a partial heap h if h can be split into two disjoint subheaps h_1 and h_2 such that h_1 satisfies P and h_2 satisfies Q. In other words, $P{*}Q$ holds only if P and Q assert distinct memory locations. It follows that for any method call $o.m(\overline{v})$ with precondition P and postcondition Q, and for any assertion R, we have $\{P * R\}\ o.m(\overline{v})\ \{Q * R\}$.

Figure 16 shows the example program, with the specifications modified as described above using separation logic. In this figure, predicate definitions and method specifications are to be interpreted in separation logic. Although the specifications have hardly changed, they have acquired a more precise meaning. In particular, asserting predicate o.valid(v) now asserts the memory locations associated with object o, in addition to expressing that the object satisfies its invariant and that its value is v. Also, the specifications of the incr methods now specify that they modify only the memory locations associated with the receiver object. Furthermore, the postcondition of method create now asserts that objects result and cell are associated with disjoint sets of memory lo-

```
interface Func {
  predicate valid((num → num) v)
  num apply(num x)
    req this.valid(v)
    ens this.valid(v) ∧ result = v(x)
}
class Util {
  static num deriv(Func f, num x)
    req f.valid(v)
    ens f.valid(v) ∧ (smooth(v, x) ⇒ result ≈ D(v)(x))
  { f.apply(x + 1) − f.apply(x) }
}
class ZeroFunc implements Func {
  predicate valid((num → num) v) = (v = λx. 0)
  num apply(num x) { 0 }
}
class Plus1Func(Func f) implements Func {
  predicate valid((num → num) v)  =
    (∃f, vf. this.f ↦ f ∗ f.valid(vf) ∧ v = λx. vf(x) + 1)
  num apply(num x) { f.apply(x) + 1 }
}
class Main {
  static void main()
    req true
    ens true
  {
    Func f := new Plus1Func(new ZeroFunc());
    {f.valid(λx. 1)}
    assert Util.deriv(f, 42) ≈ 0
  }
}
```

Fig. 17. Abstract predicate families for modular partial correctness verification of programs with dynamic binding. $D(v)$ denotes the derivative of function v. We assume appropriate definitions of smooth and $\approx$.

cations. It follows that the author of main can now conclude that the assert command is satisfied.

3.2. Abstract predicate families; recursive abstract predicates

The abstract predicates mechanism can be extended straightforwardly for modular partial correctness verification of programs with dynamic binding, as illustrated in Figure 17. The abstract predicate declared by interface Func is in fact a *family* of abstract predicates, indexed by the class of the receiver object [Parkinson and Bierman 2005]. Each class that implements the interface can provide a separate definition of the abstract predicate for its own objects.

Notice that the abstract predicate family Func.valid is defined recursively: the definition of Plus1Func.valid refers to Func.valid. In this particular case, the meaning is unambiguous, thanks to the use of separating conjunction in the definition of Plus1Func.valid: for any finite partial heap h, Plus1Func object o, and value v, o.valid(v) holds under h only if the chain of Plus1Func objects starting at o in h, obtained by following the f field, is acyclic.

In fact, in practice, in most cases, recursion within an abstract predicate family as well as mutual recursion between distinct abstract predicate families is guarded in this way. However, such guarding is not in fact necessary for consistency, and for generality, we adopt the well-formedness condition from the literature [Parkinson and Bierman 2005] that requires only that the abstract predicate family definitions of a program be *monotonic*. To understand this notion, we must first consider the notion of an *interpretation* of the abstract predicates of a program. Such an interpretation answers the question, for any partial heap h, object o, predicate name p and argument list $\overline{v}$, of whether $o.p(\overline{v})$ is true under heap h or not. We can therefore completely characterize such an interpretation by the set of tuples $(h, o, p, \overline{v})$ such that the interpretation considers $o.p(\overline{v})$ to be true under heap h.

If the abstract predicate family definitions of a program do not mention predicates in their right-hand sides, then we can obtain the intended interpretation by simply evaluating, for each object o of class C, predicate name p defined by C, argument list $\overline{v}$, and heap h, the right-hand side of the definition of p in C. If the right-hand side evaluates to true, then we include $(h, o, p, \overline{v})$ in the interpretation; otherwise, we do not. However, if the predicate definitions of a program are recursive, then in order to evaluate the right-hand sides, we need to have an interpretation already available to us! Given such an interpretation I, evaluating the right-hand sides yields a new interpretation $F(I)$. The intended interpretation, then, is one that satisfies the equation $I = F(I)$. That is, the intended interpretation is a fixpoint of F.

The Knaster-Tarski theorem tells us that such a function F has a fixpoint if it is *monotonic*, that is, if for all $I \subseteq I'$, we have $F(I) \subseteq F(I')$. In other words, a set of definitions is monotonic if interpreting additional predicate occurrences in the right-hand sides as true preserves the truth of the right-hand sides in which they occur. A sufficient condition for monotonicity is that predicates occur in definitions only in *positive positions*, such as underneath conjunctions, disjunctions, or separating conjunctions, but not underneath negation or on the left-hand side of implication.

An example of a definition that is excluded by this requirement is **predicate** $\mathsf{absurd}() = \neg\mathsf{absurd}()$. Indeed, there is no truth value for $\mathsf{absurd}()$ such that the equation $\mathsf{absurd}() = \neg\mathsf{absurd}()$ holds. However, the definition **predicate** $\mathsf{foo}() = \mathsf{foo}()$ is allowed. Not only does this equation have a solution, in fact it has multiple solutions. For the purposes of this article, we do not fix the meaning of such infinite recursions; our approach is compatible with any policy for such cases: taking the least fixpoint ($\mathsf{foo}() = \mathsf{false}$), taking the greatest fixpoint ($\mathsf{foo}() = \mathsf{true}$), or any other policy, such as a mixed policy where predicates may be marked as inductive or coinductive, and their meaning is defined through a nested fixpoint construction.

In summary, by adopting the monotonicity requirement, we obtain consistency modularly: if each module developer ensures that their predicate definitions are monotonic, then the set of predicate definitions of the program is consistent.

3.3. Verifying total correctness with separation logic

To perform modular verification of total correctness of object-oriented programs in the presence of heap mutation, we simply combine the approach of Section 2 with the separation logic approach described in this section. The only twist is that, to associate levels with objects, instead of using ghost fields, we piggyback on the use of abstract predicate families. That is, we extend the abstract predicate families used to verify partial correctness with an additional parameter of type MethodBag for each level that we wish to associate with the corresponding objects, as illustrated in Figure 18.

An example that actually performs heap mutation is shown in Figure 19. It is a dynamically bound variant of the prototypical motivating example for separation logic: in-place reversal of a linked list [Reynolds 2002].

```
interface Func {
  predicate valid((num → num) v, MethodBag applyLevel)
  num apply(num x)
    req this.valid(v, al)
    level al
    ens this.valid(v, al) ∧ result = v(x)
}
class Util {
  static num deriv(Func f, num x)
    req f.valid(v, al)
    level {[Util.deriv]} ⊎ al
    ens f.valid(v, al) ∧ (smooth(v, x) ⇒ result ≈ D(v)(x))
  { f.apply(x + 1) − f.apply(x) }
}
class ZeroFunc implements Func {
  predicate valid((num → num) v, MethodBag al)  =
      (v = (λx. 0) ∧ al = {[ZeroFunc.apply]})
  num apply(num x) { 0 }
}
class Plus1Func(Func f) implements Func {
  predicate valid((num → num) v, MethodBag al)  =
    ( ∃f, vf, alf. this.f ↦ f ∗ f.valid(vf, alf)
        ∧ v = (λx. vf(x) + 1) ∧ al = {[Plus1Func.apply]} ⊎ alf )
  num apply(num x) { f.apply(x) + 1 }
}
class Main {
  static void main()
    req true
    ens true
  {
    Func f := new Plus1Func(new ZeroFunc());
    {f.valid((λx. 1), {[ZeroFunc.apply, Plus1Func.apply]})}
    assert Util.deriv(f, 42) ≈ 0
  }
}
```

Fig. 18. Associating levels with objects by piggybacking on abstract predicate families.

4. CONCURRENCY

So far, we have considered only sequential programs. In this section, we extend our approach to achieve modular verification of total correctness of multithreaded object-oriented programs.

In Section 4.1, we show that simple concurrent programs can be verified by simply combining our sequential approach with an approach for modular verification of absence of deadlock from the literature. In Section 4.2, we extend our proposed specification style for sequential programs to obtain a specification style for concurrent programs that allows modules to introduce private locks without changing their specification. In Section 4.3, we replace level clauses by first-class call permissions, which can be transferred between threads, to improve the modularity of verifying programs where threads cause work in other threads. Finally, in Section 4.4, we show how call

```
interface List {
  predicate valid(num* elems, MethodBag level)
  List reverseAppend(List? other)
    req this.valid($\alpha$, l) $*$ ListUtil.validList(other, $\beta$, ol)
    level l
    ens result.valid($\alpha^\dagger \cdot \beta$, rl) $\land$ rl $\leq$ l $\uplus$ ol
}
class ListUtil {
  static predicate validList(List? list, num* elems, MethodBag level) =
    (list = null $\land$ elems = $\epsilon$ $\land$ level = $\mathbf{0}$ $\lor$ list $\neq$ null $\land$ list.valid(elems, level))
  static List reverseAppend(List? list, List? other)
    req validList(list, $\alpha$, l) $*$ validList(other, $\beta$, ol)
    level $\{\!\{$ListUtil.reverseAppend$\}\!\}$ $\uplus$ l
    ens validList(result, $\alpha^\dagger \cdot \beta$, rl) $\land$ rl $\leq$ l $\uplus$ ol
  { if list = null then other else list.reverseAppend(other) }
  static List reverse(List? list)
    req validList(list, $\alpha$, l)
    level $\{\!\{$ListUtil.reverse$\}\!\}$ $\uplus$ l
    ens validList(result, $\alpha^\dagger$, rl) $\land$ rl $\leq$ l
  { reverseAppend(list, null) }
}
class Cons(num value, List? tail) implements List {
  predicate valid(num* elems, MethodBag level) =
    ($\exists$v, t, $\alpha$, tl. value $\mapsto$ v $*$ tail $\mapsto$ t $*$ ListUtil.validList(t, $\alpha$, tl)
        $\land$ elems = v $\cdot$ $\alpha$ $\land$ level = $\{\!\{$Cons.reverseAppend$\}\!\}$ $\uplus$ tl )
  List reverseAppend(List? other) { List t := tail; tail := other; ListUtil.reverseAppend(t, this) }
}
```

Fig. 19. In-place reversal of a linked list of numbers, with dynamic binding. Type List? is type List extended with the special value null. **num*** denotes the type of sequences of numbers. $- \cdot -$ denotes concatenation, and $\alpha^\dagger$ denotes the reverse of α.

permissions enable the verification of termination of lock-free fine-grained concurrent algorithms, such as ones that use compare-and-swap loops.

4.1. Simple scenarios just work

Consider the simple example program of Figure 20. The **fork** c command executes command c in a new thread; the type **channel**$[\tau]$ denotes asynchronous FIFO channels (queues) for communicating messages of type τ, with unbounded buffering.

By *termination* of a concurrent program, we mean that each execution reaches a configuration where all threads have finished. (We say that the execution *finishes*, and that it reaches a *finished* configuration.) In other words, by termination we mean the absence of infinite executions and the absence of executions that reach a configuration where some threads have not finished but no thread can make a step. (In the latter case, we say that the execution *deadlocks*, and that it reaches a *deadlocked* configuration.)

In the programming language we consider in this article, an infinite execution necessarily involves an infinite number of method calls. To verify that the number of method calls in each execution is finite, it suffices to verify, just like in the sequential case, that at every call, the callee's level is below the caller's. Indeed, at each point during a concurrent execution, we can consider the multiset L obtained by replacing

```
class ProdCons {
   static void produce(channel[int] c, int n)
   { if n > 0 then { c.send(n); produce(c, n − 1) } }
   static void consume(channel[int] c, int n)
   { if n > 0 then { c.receive(); consume(c, n − 1) } }
   static void main() {
      channel[int] c := new channel[int]();
      fork consume(c, 10);
      produce(c, 10)
   }
}
```

Fig. 20. Simple producer-consumer example

each element of the multiset of all activation records of all threads by the pair (ℓ, s), where ℓ is the level of the method being executed, and s is the syntactic size of the part of the method body that remains to be executed. At each step of the execution, L decreases in the multiset order obtained by ordering the pairs (ℓ, s) lexicographically, with ℓ more significant than s. Indeed, at each call executed by a thread, the element (ℓ, s) for the thread's topmost activation record is replaced by two elements, (ℓ, s') and (ℓ', s''), with $s' < s$ and $\ell' < \ell$; at a **fork** c step, where a thread forks a new thread to execute a subcommand c, the element (ℓ, s) for the forking thread's topmost activation record is replaced by two elements of the form (ℓ, s'), where $s' < s$; when a call returns, an element is removed. At each other step, the element (ℓ, s) for the running thread's topmost activation record is replaced by (ℓ, s'), with $s' < s$. By well-foundedness of multiset order, we have that the execution is finite.

It remains, then, to verify absence of deadlock. For this, we adopt the approach of Leino et al. [2010], as later presented more elegantly by Boström and Müller [2015], for verifying absence of deadlock of programs involving locks, channels, and joinable threads, originally proposed in the context of the Chalice programming language and verification system; we will refer to it as the Chalice approach. Specifically, we consider the following concurrency constructs:

$$\begin{array}{rcl} \tau & ::= & \cdots \mid \mathbf{thread}[\tau] \mid \mathbf{lock} \mid \mathbf{channel}[\tau] \\ c & ::= & \cdots \mid \mathbf{fork}\ c \mid e.\mathbf{join}() \mid \mathbf{new\ lock}() \mid \mathbf{new\ channel}[\tau]() \\ & & \mid e.\mathbf{acquire}() \mid e.\mathbf{release}() \mid e.\mathbf{send}(e) \mid e.\mathbf{receive}() \end{array}$$

The fork c command yields a thread identifier that can be used to join the forked thread.

In the Chalice approach, performing a receive operation on a channel consumes a *credit* for this channel, which can be thought of as a *receive permission*. At any point in time, the total number of credits for a channel in the system equals the number of messages in the channel's buffer plus the number of *obligations* to send on the channel that threads have taken on. When a receive operation blocks, the receiving thread is effectively waiting for a thread holding an obligation to perform a send operation on the channel. A thread may at any time take on such an obligation. This ghost operation also creates a credit. A credit is also created by sending. A credit can be used to receive or to cancel out a send obligation.

Similarly, by successfully acquiring a lock, a thread takes on an obligation to release the lock, and a joinable thread has an obligation to eventually terminate. Each thread must discharge all of its obligations (except for its obligation to terminate) before terminating.

A blocked thread is blocked on a **join** operation, a **lock** operation, or a **receive** operation. We can think of such a thread as waiting for another thread: the thread holding the obligation to terminate, release the lock, or send on the channel[6], respectively. Therefore, in a deadlocked configuration, necessarily there is a cycle of threads, each of which is waiting on the next. This is prevented by assigning to each thread, channel, and lock (i.e. each potential target of a wait operation) a *wait level*, and by allowing a thread to perform a wait operation only if the wait level of the target is less[7] than the wait levels of the thread's obligations (i.e. its own wait level as a target for join operations, the wait levels of the channels for which it holds send obligations, and the wait levels of the locks which it holds), assuming that the less-than relation on wait levels is a strict partial order.

Indeed, consider the set B of blocked threads of the deadlocked configuration. Now consider the set of the wait levels of the waitable objects (threads, locks, and channels) they are waiting for. This finite set must have a minimal element w. So some blocked thread t is waiting for an object ω whose wait level is w. It follows that some thread t' in the configuration must have ω as an obligation. Thread t' cannot be a finished thread, since finished threads must have discharged all of their obligations. So t' must itself be a blocked thread from the set B, waiting for some object ω' whose wait level is even less than w, which is a contradiction.

To verify adherence to this system, we extend our assertion language with constructs for describing threads, locks, and channels[8]:

$$\begin{aligned} P, Q \ ::= \ & \cdots \mid t.\mathsf{thread}(Q) \mid \ell.\mathsf{lock}(\pi, I) \mid \ell.\mathsf{locked}(\pi, I, t) \\ & \mid \chi.\mathsf{channel}(P) \mid \chi.\mathsf{credit}() \mid t.\mathsf{obs}(O) \end{aligned}$$

where t, ℓ, π, χ and O range over (expressions denoting) thread identifiers, lock identifiers, fractions (rational numbers between zero, exclusive, and one, inclusive), channel identifiers, and bags of waitable object (thread, lock, or channel) identifiers, respectively. $t.\mathsf{thread}(Q)$ asserts that thread t has postcondition Q, where Q is an assertion with a free variable result denoting the thread result. $\ell.\mathsf{lock}(\pi, I)$ asserts fractional ownership [Boyland 2003; Bornat et al. 2005] with fraction π of lock ℓ with invariant I. I is an assertion that describes the resources protected by the lock. $\ell.\mathsf{locked}(\pi, I, t)$ additionally denotes that thread t currently holds the lock. $\chi.\mathsf{channel}(P)$ asserts shared ownership of channel χ with element predicate P, which is an assertion with a free variable $\mathsf{element}$ denoting an element.[9] Whenever the channel holds elements $\overline{v}$, it owns the resources described by $\circledast_{v \in \overline{v}} P[v/\mathsf{element}]$, the separating conjunction of $P[v/\mathsf{element}]$ for all $v \in \overline{v}$. $\chi.\mathsf{credit}()$ asserts ownership of one credit (receive permission) for channel χ. $t.\mathsf{obs}(O)$ asserts that some thread currently holds obligations O.

These assertions satisfy the following laws:

$$\begin{array}{rrcl} \textsc{LockSplitMerge} & \ell.\mathsf{lock}(\pi_1 + \pi_2, I) & \Leftrightarrow & \ell.\mathsf{lock}(\pi_1, I) * \ell.\mathsf{lock}(\pi_2, I) \\ \textsc{LockDestroy} & \ell.\mathsf{lock}(1, I) & \Rightarrow & I \\ \textsc{ChannelDup} & \chi.\mathsf{channel}(P) & \Rightarrow & \chi.\mathsf{channel}(P) * \chi.\mathsf{channel}(P) \\ \textsc{ChannelOb} & t.\mathsf{obs}(O) * \chi.\mathsf{channel}(P) & \Leftrightarrow & t.\mathsf{obs}(O \uplus \{\!\{\chi\}\!\}) * \chi.\mathsf{channel}(P) * \chi.\mathsf{credit}() \end{array}$$

In words: lock fractions can be split and merged; full lock ownership allows the owner to destroy the lock, obtaining direct ownership of the protected resources; channel per-

[6]It does not matter whether one thinks of a thread blocked on a **receive** as waiting for *all* threads holding an obligation to send on the channel or for *some* such thread.
[7]In Chalice: *greater*. We invert Chalice's wait level order for synergy with our termination level order.
[8]The Chalice approach was formulated in the context of Chalice's *implicit dynamic frames* logic; we here present a translation of the approach into our separation logic setting.
[9]We do not consider destroying channels, so we do not use fractions to track sharing of channels.

$$\frac{\forall t'.\ t' \vdash \{t'.\mathsf{obs}(O') * P\}\ c\ \{t'.\mathsf{obs}(\mathbf{0})\}}{t \vdash \{t.\mathsf{obs}(O \uplus O') * P\}\ \textbf{fork}\ c\ \{t.\mathsf{obs}(O)\}}$$

$$\frac{\forall t'.\ \mathsf{w}(t') = w \Rightarrow\ t' \vdash \{t'.\mathsf{obs}(O' \uplus \{\!\!\{t'\}\!\!\}) * P\}\ c\ \{t'.\mathsf{obs}(\{\!\!\{t'\}\!\!\}) * Q\}}{t \vdash \{t.\mathsf{obs}(O \uplus O') * P\}\ \textbf{fork}\ c\ \{t.\mathsf{obs}(O) * \mathsf{result.thread}(Q) \wedge \mathsf{w}(\mathsf{result}) = w\}}$$

$$\begin{array}{ll} & \{t.\mathsf{obs}(O) * t'.\mathsf{thread}(Q) \wedge \mathsf{w}(t') \prec O\} \\ t \vdash & t'.\textbf{join}() \\ & \{t.\mathsf{obs}(O) * Q\} \end{array}$$

$$t \vdash \{I\}\ \textbf{new lock}()\ \{\mathsf{result.lock}(1, I) \wedge \mathsf{w}(\mathsf{result}) = w\}$$

$$\begin{array}{ll} & \{t.\mathsf{obs}(O) * \ell.\mathsf{lock}(\pi, I) \wedge \mathsf{w}(\ell) \prec O\} \\ t \vdash & \ell.\textbf{acquire}() \\ & \{t.\mathsf{obs}(O \uplus \{\!\!\{\ell\}\!\!\}) * \ell.\mathsf{locked}(\pi, I, t) * I\} \end{array} \qquad \begin{array}{ll} & \{t.\mathsf{obs}(O \uplus \{\!\!\{\ell\}\!\!\}) * \ell.\mathsf{locked}(\pi, I, t) * I\} \\ t \vdash & \ell.\textbf{release}() \\ & \{t.\mathsf{obs}(O) * \ell.\mathsf{lock}(\pi, I)\} \end{array}$$

$$t \vdash \{\mathsf{true}\}\ \textbf{new channel}[\tau]()\ \{\mathsf{result.channel}(P) \wedge \mathsf{w}(\mathsf{result}) = w\}$$

$$t \vdash \{\chi.\mathsf{channel}(P) * P[v/\mathsf{element}]\}\ \chi.\textbf{send}(v)\ \{\chi.\mathsf{credit}()\}$$

$$\begin{array}{ll} & \{t.\mathsf{obs}(O) * \chi.\mathsf{channel}(P) * \chi.\mathsf{credit}() \wedge \mathsf{w}(\chi) \prec O\} \\ t \vdash & \chi.\textbf{receive}() \\ & \{t.\mathsf{obs}(O) * P[\mathsf{result}/\mathsf{element}]\} \end{array}$$

Fig. 21. Separation logic proof rules for the Chalice approach. $w \prec O$ means $\forall o \in O.\ w < \mathsf{w}(o)$.

missions can be duplicated; channel obligations can be created and destroyed, which implies creating/destroying a credit.

Additionally, we assume a partially ordered set of wait levels and a function w that maps each waitable object identifier to a wait level. We assume that for each wait level w, there are infinitely many thread identifiers t such that $\mathsf{w}(t) = w$, and similarly for lock identifiers and channel identifiers. We will often need to express that a wait level w is below the wait levels of a bag of obligations O. We denote this by $w \prec O$; formally, it means $\forall o \in O.\ w < \mathsf{w}(o)$.

For reasoning about concurrent programs, we use correctness judgments of the form $t \vdash \{P\}\ c\ \{Q\}$, denoting that running command c in thread t, starting in a state that satisfies precondition P, does not fail or deadlock and finishes with a result value v in a state that satisfies postcondition $Q[v/\mathsf{result}]$, under appropriate assumptions on the environment. The proof rules governing the concurrency constructs are shown in Figure 21.

There are two proof rules for thread creation: one for cases where the thread will not be joined, and one for cases where it will. In both cases, the parent thread can pass some of its obligations to the child. In the latter case, the child t' is additionally charged with an obligation t', representing the obligation to terminate. w is the wait level of the new thread; it can be picked freely. The child must dispose of all of the obligations it received from its parent before it terminates.

Joining implies waiting, so the wait level of the target thread must be below the joining thread's obligations.

Creating a lock consumes the lock invariant. A wait level w for the lock can be picked freely.

```
class ProdCons {
  static void produce(channel[int] c, int n)
    req $0 \leq n \wedge obs(\{\!\!\{ n \cdot c \}\!\!\}) * c.channel(true)$
    level $\{\!\!\{ ProdCons.produce \}\!\!\} + n$
    ens $obs(\mathbf{0})$
  { if n > 0 then { c.send(n); produce(c, n − 1) } }
  static void consume(channel[int] c, int n)
    req $0 \leq n \wedge obs(\mathbf{0}) * c.channel(true) * n \cdot c.credit$
    level $\{\!\!\{ ProdCons.consume \}\!\!\} + n$
    ens $obs(\mathbf{0})$
  { if n > 0 then { c.receive(); consume(c, n − 1) } }
  static void main()
    req $obs(\mathbf{0})$
    level $\{\!\!\{ ProdCons.main \}\!\!\}$
    ens $obs(\mathbf{0})$
  {
    channel[int] c := new channel[int]();
    $\{obs(\mathbf{0}) * c.channel(true)\}$
    CHANNELOB
    $\{obs(\{\!\!\{ 10 \cdot c \}\!\!\}) * 10 \cdot c.credit * c.channel(true)\}$
    fork consume(c, 10);
    $\{obs(\{\!\!\{ 10 \cdot c \}\!\!\}) * c.channel(true)\}$
    produce(c, 10)
  }
}
```

Fig. 22. Simple producer-consumer example, verified

When a thread acquires a lock, its thread identifier is recorded as an argument of the locked assertion. This ensures that the same thread releases the lock, as required by most lock implementations.

Sending on a channel produces a credit, and receiving consumes it.

We are now ready to verify the example program of Figure 20. An annotated version is shown in Figure 22. Note: in annotations, we denote the current thread by currentThread. We use $\mathsf{obs}(O)$ as an abbreviation for $\mathsf{currentThread.obs}(O)$.

Figure 23 shows an example that illustrates locking, lock invariants, thread joining, and join permissions that specify thread postconditions, and that requires a careful choice of wait levels. The reason why this program is deadlock-free is that each thread acquires the locks in descending order. Therefore, any valid proof of this program must assign wait levels to the locks correspondingly. Indeed, if a thread waits for a lock while holding another lock, the former lock's wait level must be below the latter's. Additionally, each thread's wait level must be greater than the levels of the objects that thread waits for.

4.2. Hiding private locks by using termination levels to order locks

In the examples above, we used **req** $\mathsf{obs}(\mathbf{0})$ **ens** $\mathsf{obs}(\mathbf{0})$ as the specification for method main. Now, suppose these example programs are part of larger programs and these main methods are called while the caller is holding some obligations. The above specification for main does not allow us to verify such programs, even though such calls of main are equivalent to no-ops and perfectly safe.

```
class Counter(int n) {}
class Dining {
  static void main()
    req obs(0)
    ens obs(0)
  {
    Counter c1 := new Counter(0); lock fork1 := new lock(); {w(fork1) = 1}
    Counter c2 := new Counter(0); lock fork2 := new lock(); {w(fork2) = 2}
    Counter c3 := new Counter(0); lock fork3 := new lock(); {w(fork3) = 3}
    {obs(0) ∗ fork1.lock(1, c1.n ↦ _) ∗ fork2.lock(1, c2.n ↦ _) ∗ fork3.lock(1, c3.n ↦ _)}
    thread philo1 := fork {
      {obs({[philo1]}) ∗ fork1.lock(1/2, c1.n ↦ _) ∗ fork2.lock(1/2, c2.n ↦ _)}
      fork2.acquire(); c2.n := c2.n + 1;
      {obs({[philo1, fork2]}) ∗ fork1.lock(1/2, c1.n ↦ _)
       ∗ fork2.locked(1/2, c2.n ↦ _, philo1) ∗ c2.n ↦ _}
      fork1.acquire(); c1.n := c1.n + 1;
      {obs({[philo1, fork2, fork1]}) ∗ fork1.locked(1/2, c1.n ↦ _, philo1) ∗ c1.n ↦ _
       ∗ fork2.locked(1/2, c2.n ↦ _, philo1) ∗ c2.n ↦ _}
      fork1.release(); fork2.release()
      {obs({[philo1]}) ∗ fork1.lock(1/2, c1.n ↦ _) ∗ fork2.lock(1/2, c2.n ↦ _)}
    }; {w(philo1) = 4}
    {fork1.lock(1/2, c1.n ↦ _) ∗ fork2.lock(1/2, c2.n ↦ _) ∗ fork3.lock(1, c3.n ↦ _)
     ∗ philo1.thread(fork1.lock(1/2, c1.n ↦ _) ∗ fork2.lock(1/2, c2.n ↦ _)) ∗ obs(0)}
    thread philo2 := fork {
      fork3.acquire(); c3.n := c3.n + 1;
      fork2.acquire(); c2.n := c2.n + 1;
      fork2.release(); fork3.release()
    }; {w(philo2) = 4}
    thread philo3 := fork {
      fork3.acquire(); c3.n := c3.n + 1;
      fork1.acquire(); c1.n := c1.n + 1;
      fork1.release(); fork3.release
    }; {w(philo3) = 4}
    philo1.join(); philo2.join(); philo3.join()
    {obs(0) ∗ fork1.lock(1, c1.n ↦ _) ∗ fork2.lock(1, c2.n ↦ _) ∗ fork3.lock(1, c3.n ↦ _)}
    {obs(0) ∗ c1.n ↦ _ ∗ c2.n ↦ _ ∗ c3.n ↦ _}
  }
}
```

Fig. 23. Dining philosophers example illustrating the use of lock invariants, join permissions that specify thread postconditions, and wait levels. Here, we assume the wait levels include the natural numbers.

For programs such as the ones above, this problem can be solved easily, if we assume that the set of wait levels is such that for every finite subset, there is a wait level that is below it. We modify the specification of main to **req** obs(O) **ens** obs(O) and we pick as the wait levels for the waitable objects created internally some arbitrary wait levels that are below O.

However, consider now the module of Figure 24. We would like to devise a specification for this module that allows for maximum reuse. Notice that method sqrtCached differs from the main methods above in that it acquires a pre-existing

```
class Math {
  public static num sqrt(num x)
  { ··· expensive computation ··· }
  private static final HashMap[num, num] sqrtCache := new HashMap[num, num]()
  private static final lock sqrtCacheLock := new lock()
  public static num sqrtCached(num x) {
    sqrtCacheLock.acquire();
    if ¬sqrtCache.containsKey(x) then
      sqrtCache.put(x, sqrt(x));
    num result := sqrtCache.get(x);
    sqrtCacheLock.release();
    result
  }
}
```

Fig. 24. A module that uses a private lock

lock, with a pre-determined wait level. Therefore, it does not satisfy the specification **req** $\mathsf{obs}(O)$ **ens** $\mathsf{obs}(O)$.

To solve this problem, we propose to use our termination levels as a basis for a program's wait levels. In particular, we propose to use as the set of wait levels the pairs (ℓ, w_L) of multisets of method names ℓ and *local wait levels* w_L, lexicographically ordered, with ℓ more significant than w_L. Correspondingly, we propose the specification style where each method asserts in its precondition $\mathsf{obs}(O) \wedge \ell \prec O$, where ℓ is the method's level (as specified in its **level** clause) and $\ell \prec O$ means $\forall (\ell', w_\mathrm{L}) \in \mathsf{w}(O).\ \ell < \ell'$. That is, each method should require that the current thread's obligations are above its own termination level. If each module assigns to its internal waitable objects wait levels based on its own termination level, then this specification style allows each module to wait on its own waitable objects, as well as to call into lower modules while holding obligations on its own waitable objects.

Figure 25 shows this specification style applied to the example of Figure 24. For simplicity, we turned the static fields into instance fields. The ghost method allows the assertion $o.\mathsf{valid}$, for Math objects o, to be duplicated arbitrarily.

Since this specification style is not burdensome for callers or callees, we can, at no significant cost, apply this style to the specifications of all methods, even those whose current implementation does not involve waitable objects at all. This enables module authors to introduce private locks into modules without changing the module's specification. Indeed, notice in Figure 25 that the specification of sqrt and of sqrtCached are entirely analogous; callers cannot tell which one is using private locks.

4.3. Call permissions to support threads causing work in other threads

Notice that in the termination verification approach based on **level** clauses, a thread's level (i.e. the multiset obtained by replacing each element of the multiset of the thread's activation records by a pair (ℓ, s) where ℓ is the level of the method being executed and s is the syntactic size of the part of the method body that remains to be executed, ordered lexicographically, with ℓ more significant than s) is destined to follow a descending chain in the multiset order, regardless of communication with or interference from other threads. This can be burdensome.

Consider for example the program of Figure 26. Class ThreadPool implements a simple single-thread thread pool. When using **level** clauses, a level value bounding the number of tasks that will be submitted to the thread pool, as well as bounding the

```
class Math(final HashMap[num, num] sqrtCache, final lock sqrtCacheLock) {
  predicate valid() =
    (∃π. sqrtCacheLock.lock(π, sqrtCache.valid())
       ∧ w(sqrtCacheLock) = ({|Math.sqrtCached|}, 0))
  public static Math create()
    req obs(O) ∧ {|Math.create|} ≺ O
    level {|Math.create|}
    ens result.valid() ∗ obs(O)
  {
    HashMap[num, num] sqrtCache := new HashMap[num, num]();
    lock sqrtCacheLock := new lock();
    new Math(sqrtCache, sqrtCacheLock)
  }
  public ghost void duplicate()
    req valid()
    ens valid() ∗ valid()
  { }
  public num sqrt(num x)
    req valid() ∗ obs(O) ∧ {|Math.sqrt|} ≺ O ∧ 0 ≤ x
    level {|Math.sqrt|}
    ens obs(O) ∧ result ≈ √x
  { ··· expensive computation ··· }
  public num sqrtCached(num x)
    req valid() ∗ obs(O) ∧ {|Math.sqrtCached|} ≺ O ∧ 0 ≤ x
    level {|Math.sqrtCached|}
    ens obs(O) ∧ result ≈ √x
  {
    sqrtCacheLock.acquire();
    if ¬sqrtCache.containsKey(x) then
      sqrtCache.put(x, sqrt(x));
    num result := sqrtCache.get(x);
    sqrtCacheLock.release();
    result
  }
}
```

Fig. 25. The private locks example, verified (after eliminating static fields for simplicity)

levels of the tasks' run methods, must be determined when the thread pool is created. This could complicate the specification of modules that use the thread pool, such as method doWork.

Specifically, a specification for method ThreadPool.addTask could be:

```
public void addTask(Task task)
  req valid(level) ∗ task.valid(tl) ∧ tl < level ∧ level′ < level
  level {|ThreadPool.addTask|}
  ens valid(level′)
```

```
interface Task { void run() }
class ThreadPool(final channel[Item] channel) {
   private interface Item { void run() }
   private void work() { channel.receive().run() }
   private class TaskItem(final Task task) implements Item {
         void run() { task.run(); work() }
   }
   public void addTask(Task task) { channel.send(new TaskItem(task)) }
   private class ShutdownItem implements Item { void run() {} }
   public void shutDown() { channel.send(new ShutdownItem()) }
   public static ThreadPool create() {
      ThreadPool pool := new ThreadPool(new channel[Item]());
      fork pool.work();
      pool
   }
}
class MyTask implements Task { void run() {} }
class Main {
   static void doWork(ThreadPool pool) {
      pool.addTask(new MyTask());
      pool.addTask(new MyTask())
   }
   static void main() {
      ThreadPool pool := ThreadPool.create();
      doWork(pool);
      pool.shutDown()
   }
}
```

Fig. 26. Thread pool example

A corresponding specification for Main.doWork could be:

```
static void doWork(ThreadPool pool)
   req pool.valid({[Main.doWork]} ⊎ level)
   level {[Main.doWork]}
   ens pool.valid(level)
```

Furthermore, a non-trivial ghost state construction would be necessary in the proof of ThreadPool to maintain the link between the argument of the valid predicate and the value of the level clause of method ThreadPool.work.

Generalizing over this example, work (in the form of method calls) may in general be imposed on a thread by other threads. When using level clauses, the proof must determine a bound on the amount of such work when the thread is created, and must introduce a ghost state construction to track the level of the target thread in the originator threads.

To eliminate this burden, we propose first-class *call permissions* as a replacement for level clauses. Specifically, we define an *instrumented execution semantics* for programs where an execution state consists of a heap and a *stock of call permissions*, which is a multiset of level values. Initially, the stock of call permissions is $\{\{\![\text{Main.main}]\!\}\}$, i.e. it contains a single call permission qualified by the name of the main method, considered as a singleton multiset. In the instrumented semantics, each method call, before executing the method body, angelically picks an element from the stock of call permissions and removes it. If at the point of a method call, the stock of call permissions is empty, the instrumented execution goes wrong. Furthermore, before each execution step, the instrumented semantics replaces the stock of call permissions by an angelically picked lesser-or-equal one in multiset order. This means that each call permission is potentially replaced by an arbitrary number of lesser call permissions. By saying that the choices are angelic, we mean that if for each such choice in an execution, an option exists such that the execution does not go wrong, then for each such choice some such option is chosen.

It is easy to see that if an instrumented execution does not go wrong, then its *erasure* (i.e. the corresponding non-instrumented execution) terminates. Indeed, suppose the execution performs an infinite number of method calls. Then the sequence of the stocks of call permissions before each method call forms an infinite descending chain, which is impossible due to the well-foundedness of multiset order.

To prove termination of a program, then, it is sufficient to prove that no instrumented execution goes wrong, that is, that for each execution there exists a way to reduce the stock of call permissions (or not) at each execution step such that there exists a call permission to remove from the stock at each method call.

To prove this modularly, we use separation logic to assign *ownership* of call permissions to threads, just like we assign ownership of memory locations to threads. Just like ownership of memory locations, ownership of call permissions can be passed between threads through locks and channels.

We introduce the assertion $\mathsf{cp}(\ell)$, where ℓ is a level value, to denote ownership of a call permission ℓ. We modify the proof rule for methods calls so that it asserts the availability of a call permission for removal:

$$\frac{o.m(\overline{v})\ \mathbf{req}\ P\ \mathbf{ens}\ Q}{\{\mathsf{cp}(\_) * P\}\ o.m(\overline{v})\ \{Q\}}$$

Furthermore, we modify the rule of consequence, to allow weakening of the locally owned stock of call permissions (in order to exchange a call permission for a number of lesser call permissions, for the sake of making call permissions available for consumption by method calls or for distribution to other threads):

$$\frac{P \sqsubseteq P' \qquad \{P'\}\ c\ \{Q\} \qquad Q \sqsubseteq Q'}{\{P\}\ c\ \{Q'\}}$$

where

$$P \sqsubseteq P' \quad \Leftrightarrow \quad \forall h, \Lambda.\ h, \Lambda \vDash P \Rightarrow \exists \Lambda' \leq \Lambda.\ h, \Lambda' \vDash P'$$

where $h, \Lambda \vDash P$ means that assertion P is satisfied by partial heap h and stock of call permissions (i.e. multiset of level values) Λ. For example, if methods $\mathsf{Foo.foo}$ and $\mathsf{Bar.bar}$ are less than $\mathsf{Main.main}$ in the order on method names, then we have $\mathsf{cp}(\{\!\{\mathsf{Main.main}\}\!\}) \sqsubseteq 2 \cdot \mathsf{cp}(\{\!\{\mathsf{Foo.foo}\}\!\}) * 3 \cdot \mathsf{cp}(\{\!\{4 \cdot \mathsf{Foo.foo}, 5 \cdot \mathsf{Bar.bar}\}\!\})$, because $\{\!\{\{\!\{\mathsf{Main.main}\}\!\}\}\!\} > \{\!\{2 \cdot \{\!\{\mathsf{Foo.foo}\}\!\}, 3 \cdot \{\!\{4 \cdot \mathsf{Foo.foo}, 5 \cdot \mathsf{Bar.bar}\}\!\}\}\!\}$.

We move from specifications of the form $\mathbf{req}\ P\ \mathbf{level}\ \ell\ \mathbf{ens}\ Q$ to specifications of the form $\mathbf{req}\ P * \mathsf{cp}(\ell)\ \mathbf{ens}\ Q$.

We illustrate this approach by applying it to the thread pool example. Figures 27 and 28 show this example with annotations proving absence of infinite recursion. (To focus on call permissions, we postpone proving absence of deadlocks; see below for a full proof.) Notice that all methods' specifications adhere to the specification style proposed in Section 2, except that the level is used to qualify a call permission asserted in the precondition instead of being used in a level clause, and except for method

```
interface Task {
  predicate valid(MethodBag level)
  void run()
    req valid(l) ∗ cp(l)
    ens true
}
class ThreadPool(final channel[Item] channel) {
  predicate valid() = channel.channel(element.validItem())
  private interface Item {
    predicate validItem()
    void run()
      req valid() ∗ validItem()
      ens true
  }
  private void work()
    req valid() ∗ cp({[ThreadPool.work]})
    ens true
  { channel.receive().run() }
  private class TaskItem(final Task task) implements Item {
    predicate validItem() =
      (∃tl. task.valid(tl) ∗ cp({[TaskItem.run]} ⊎ tl))
    void run() { task.run(); pool.work() }
  }
  public void addTask(Task task)
    req valid() ∗ task.valid(tl) ∗ cp({[ThreadPool.addTask]} ⊎ tl)
    ens valid()
  { channel.send(new TaskItem(task)) }
  private class ShutdownItem implements Item {
    predicate validItem() = true
    void run() {}
  }
  public void shutDown()
    req valid() ∗ cp({[ThreadPool.shutDown]})
    ens true
  { channel.send(new ShutdownItem()) }
  public static ThreadPool create()
    req cp({[ThreadPool.create]})
    ens result.valid()
  {
    ThreadPool pool := new ThreadPool(new channel[Item]());
    fork pool.work();
    pool
  }
}
```

Fig. 27. Annotated thread pool module (ignoring deadlocks)

```
class MyTask implements Task {
  predicate valid(MethodBag level) = (level = {[MyTask.run]})
  void run() {}
}
class Main {
  static void doWork(ThreadPool pool)
    req pool.valid() ∗ cp({[Main.doWork]})
    ens pool.valid()
  {
    {pool.valid() ∗ 4 · cp({[ThreadPool.addTask, MyTask.run]})}
    pool.addTask(new MyTask());
    {pool.valid() ∗ 2 · cp({[ThreadPool.addTask, MyTask.run]})}
    pool.addTask(new MyTask())
    {pool.valid()}
  }
  static void main()
    req cp({[Main.main]})
    ens true
  {
    {2 · cp({[ThreadPool.create]}) ∗ 2 · cp({[Main.doWork]}) ∗ 2 · cp({[ThreadPool.shutDown]})}
    ThreadPool pool := ThreadPool.create();
    {pool.valid() ∗ 2 · cp({[Main.doWork]}) ∗ 2 · cp({[ThreadPool.shutDown]})}
    doWork(pool);
    {pool.valid() ∗ 2 · cp({[ThreadPool.shutDown]})}
    pool.shutDown()
  }
}
```

Fig. 28. Annotated thread pool client (ignoring deadlocks)

Item.run, which is internal to the ThreadPool implementation. Indeed, most methods' preconditions assert a call permission qualified with the method's name (considered as a singleton multiset). This allows these methods to perform any number of calls of lesser methods. For example, consider method Main.main. Each of the three calls in its body needs two call permissions: one that is removed from the stock of call permissions at the start of the call by the instrumented execution, and one that is asserted by the callee's precondition. Therefore, at the top of the body of Main.main we use the rule of consequence shown above to reduce the incoming call permission cp({[Main.main]}) to the six call permissions needed by the body.

Similarly, to prove method Main.doWork, we use the rule of consequence to reduce the incoming call permission cp({[Main.doWork]}) to four copies of cp({[ThreadPool.addTask, MyTask.run]}), the level required by ThreadPool.addTask's specification. Two copies are consumed at the start of the two calls, and two are passed to the callee, as required by its precondition.

The added value of call permissions over level clauses is illustrated by the proof of method ThreadPool.work. It retrieves an item from the channel and calls its run method. However, ThreadPool.work's specification asserts a call permission qualified only with its own name. In the corresponding situation when using level clauses, this would be problematic. With call permissions, this problem is solved easily. Consider in particular the specification of method Item.run. Notice that it does not directly assert any call permissions! Rather, any call permissions needed by this method are asserted through the validItem predicate. See in particular the definition of validItem in class TaskItem. The resources represented by this predicate are retrieved from the channel as part of the receive operation in method work. They were transferred to the channel as part of the send operation in method addTask.

A full proof of the thread pool example, including the proof of absence of deadlock, is shown in Figures 29 and 30. The additional annotations follow the style proposed in Section 4.2: each method asserts $\mathsf{obs}(O)$ in its precondition and its postcondition, and asserts in its precondition $\ell \prec O$, where ℓ is the method's level. The novel element in this example is the abstract specification of the obligations O' associated with the thread pool, taken on by the client thread when calling ThreadPool.create, and discharged when calling ThreadPool.shutDown. In the ThreadPool implementation shown, O' is simply the thread pool's channel. However, in the specification we allow for multiple waitable objects to be associated with the thread pool. Furthermore, an implementation might want to associate distinct wait levels with its waitable objects, ordered in a particular way. At the same time, the client also has requirements about the wait levels of the obligations taken on. In particular, they should be above the levels of any methods called by the client. For these reasons, the specification of ThreadPool.create allows the client to specify a set W of available wait levels. To accomodate arbitrarily complex implementations, including ones that are internally built from arbitrarily many layers of abstraction, the specification requires this set to be infinite, and to be such that there are levels above and below any level, and between any two ordered levels. (For conciseness, we express this by requiring that W be order-isomorphic to the rational numbers.) To satisfy these requirements, method Main.main uses the rational numbers as its local wait levels, and it specifies for W the set of all of the wait levels at its own termination level.[10]

4.4. Compare-and-swap loops

In Section 4.3, we showed how call permissions enable one to elegantly prove termination of programs where threads cause work in other threads by effectively sending code for execution to other threads. Another way that a thread can cause work in another thread is by *interfering* with the other thread's execution of a *lock-free* concurrent algorithm, forcing the other thread to abandon the execution and try again.

Consider the example of a lock-free stack in Figure 31. We only show the push operation; the pop operation is similar. We use the *atomic block* notation $\langle c \rangle$ to denote the command c executed atomically. The second atomic block in the example can be implemented using a *compare-and-swap* machine instruction, available on most architectures. The code exhibits a typical compare-and-swap loop: it reads a shared variable, computes a new value, and then attempts to install the new value, under the condition that the variable was not changed by another thread in the meantime. Otherwise, it tries again.

Fine-grained concurrent data structures such as this one are used to distribute work in parallel computing. A data structure is lock-free if, at any point, there exists a number N such that if any of the threads currently accessing the data structure is sche-

[10]This approach relies on the assumption that the properties we stated as requirements for the set of wait levels available to the implementation are sufficient to verify any implementation. While we believe this is the case, we could eliminate this assumption by adopting the following alternative approach. The specification of ThreadPool would expose the existence of some partially ordered set S (e.g. by declaring it as a static final ghost field ThreadPool.waitLevels of type POSet). The specification of ThreadPool.create would then allow the client to specify an order-preserving injection h of S into the wait levels, such that $\{\mathsf{ThreadPool.create}\} \prec h(S)$. The example client would choose S as its set of local wait levels, and would define h by $h(w) = (\{\mathsf{Main.main}\}, w)$. The example implementation would define S as a singleton set, since it needs only one wait level. We thank the anonymous reviewer who suggested this approach.

```
interface Task {
  predicate valid(MethodBag level)
  void run()
    req obs($O$) ∗ valid(l) ∗ cp(l) ∧ l ≺ $O$
    ens obs($O$)
}
class ThreadPool(final channel[Item] channel) {
  predicate valid(ObligationBag $O$) =
    channel.channel(element.validItem()) ∧ $O$ = {[channel]}
  private interface Item {
    predicate validItem()
    void run()
      req obs($\mathbf{0}$) ∗ valid(_) ∗ validItem()
      ens obs($\mathbf{0}$)
  }
  private void work()
    req obs($\mathbf{0}$) ∗ valid(_) ∗ channel.credit ∗ cp({[ThreadPool.work]})
    ens obs($\mathbf{0}$)
  { channel.receive().run() }
  private class TaskItem(final Task task) implements Item {
    predicate validItem() =
      (∃tl. channel.credit ∗ task.valid(tl) ∗ cp({[TaskItem.run]} ⊎ tl))
    void run() { task.run(); pool.work() }
  }
  public void addTask(Task task)
    req obs($O$) ∗ valid($O'$) ∗ task.valid(tl) ∗ cp({[ThreadPool.addTask]} ⊎ tl)
      ∧ {[ThreadPool.addTask]} ≺ $O$
    ens obs($O$) ∗ valid($O'$)
  { channel.send(new TaskItem(task)) }
  private class ShutdownItem implements Item {
    predicate validItem() = true
    void run() {}
  }
  public void shutDown()
    req obs($O ⊎ O'$) ∗ valid($O'$) ∗ cp({[ThreadPool.shutDown]})
      ∧ {[ThreadPool.shutDown]} ≺ $O ⊎ O'$
    ens obs($O$)
  { channel.send(new ShutdownItem()) }
  public static ThreadPool create()
    req obs($O$) ∗ cp({[ThreadPool.create]}) ∧ {[ThreadPool.create]} ≺ $O$
      ∧ $W \cong_< \mathbb{Q}$ ∧ {[ThreadPool.create]} ≺ $W$
    ens ∃$O'$. obs($O ⊎ O'$) ∗ result.valid($O'$) ∧ w($O'$) ⊆ $W$
  {
    ThreadPool pool := new ThreadPool(new channel[Item]);
    fork pool.work();
    pool
  }
}
```

Fig. 29. Thread pool module: full termination proof

```
class MyTask implements Task {
  predicate valid(MethodBag level) = (level = {[MyTask.run]})
  void run() {}
}
class Main {
  static void doWork(ThreadPool pool)
    req obs($O$) $*$ pool.valid($O'$) $*$ cp({[Main.doWork]}) $\wedge$ {[Main.doWork]} $\prec O$
    ens obs($O$) $*$ pool.valid($O'$)
  {
    pool.addTask(new MyTask());
    pool.addTask(new MyTask())
  }
  static void main()
    req obs($O$) $*$ cp({[Main.main]}) $\wedge$ {[Main.main]} $\prec O$
    ens obs($O$)
  {
    ThreadPool pool := ThreadPool.create();    $W = \{$ {[Main.main]} $\} \times \mathbb{Q}$
    doWork(pool);
    pool.shutDown()
  }
}
```

Fig. 30. Thread pool client: full termination proof

```
class Node(int value, Node next) {}
class Stack(Node head) {
  void pushIter(int value) {
    Node head := ⟨this.head⟩; Node n := new Node(value, head);
    Node head1 :=
      ⟨ Node head1 := this.head; if head1 = head then this.head := n;  head1 ⟩;
    if head1 $\neq$ head then this.pushIter(value)
  }
  void push(int value) { this.pushIter(value) }
}
```

Fig. 31. A lock-free stack. (The pop operation is similar and is not shown.)

duled for N steps (potentially with steps by other threads interleaved between these N steps), then at least one of the threads accessing the data structure will make progress. Note that this is not the case for data structures that use locks, since if the thread holding the lock is not scheduled, no other thread can make progress.

Our approach for verifying absence of infinite executions can be used to verify termination of programs involving compare-and-swap loops like the one above. A proof outline for the example program is shown in Figure 32.

The proof is based on the observation that whenever an operation has to try again, then some other concurrent operation has succeeded. The idea is then that the operation that succeeds supplies a call permission to each of the concurrent operations that it causes to fail, to enable them to try again. Since it is not known beforehand how many concurrent operations will be in progress at that time, method push passes a call permission qualified with level {[Stack.pushIter]} $+ 1$ to pushIter, which can be reduced

```
class GhostBag[T] {
  predicate GhostBag(Bag[T] elems)
  predicate GhostBagHandle(T elem)
  void add(T value)
    req this.GhostBag(elems)
    ens this.GhostBag(elems ⊎ {[value]}) ∗ this.GhostBagHandle(value)
  void remove(T value)
    req this.GhostBag(elems) ∗ this.GhostBagHandle(value)
    ens this.GhostBag(elems − {[value]}) ∧ value ∈ elems
}
class Stack(Node head, final GhostBag[Node] readers) {
  static predicate nodes(Node n) =
    ∃next. n = null ∨ n.value ↦ _ ∗ n.next ↦ next ∗ nodes(next)
  predicate spaceInv() =
    ∃h, rs. head ↦ h ∗ nodes(h) ∗ readers.GhostBag(rs)
      ∗ |{[r ∈ rs | r ≠ h]}| · cp({[Stack.pushIter]})
  predicate valid() = atomic_space(spaceInv())
  ghost void duplicate()
    req valid()
    ens valid() ∗ valid()
  {}
  void pushIter(int value)
    req valid() ∗ cp({[Stack.pushIter]} + 1)
    ens valid()
  {
    Node head := ⟨
      Node head := this.head; readers.add(head);head
    ⟩;
    { atomic_space(spaceInv()) ∗ readers.GhostBagHandle(head)
      ∗ cp({[Stack.pushIter]} + 1) }
    Node n := new Node(value, head);
    Node head1 := ⟨
      readers.remove(head);
      Node head1 := this.head;
      if head1 = head then this.head := n;
      head1
    ⟩;
    { atomic_space(spaceInv())
      ∗ (head1 = head ∨ (cp({[Stack.pushIter]}) ∗ cp({[Stack.pushIter]} + 1))) }
    if head1 ≠ head then pushIter(value)
  }
  void push(int value)
    req valid() ∗ cp({[Stack.push]})
    ens valid()
  { pushIter(value) }
}
```

Fig. 32. Proof outline for the lock-free stack example

```
class OS {                   class Main {
  static void beep()           static void iter()
}                              { OS.beep(); iter() }
                               static void main()
                               { iter() }
                             }
```

Fig. 33. A program that is non-terminating but live

to an arbitrary number of $\{\!\!\{\mathsf{Stack.pushIter}\}\!\!\}$ permissions. Note: a pushIter execution does not use this call permission for its own recursive calls; rather, it uses it only when it succeeds, to supply the required call permissions to the concurrent operations. At that point, it reduces the permission to n lesser permissions, where n is the number of concurrent operations in progress at that time.

To verify the atomic blocks, the proof uses the notion of an *atomic space*: similar to a lock, an atomic space has an *atomic space invariant* that describes resources that are being shared between multiple threads and that should only be accessed through atomic blocks. The ghost operation of creating an atomic space consumes the resources described by the atomic space invariant, and produces an *atomic space handle* $\mathsf{atomic\_space}(I)$, where I is the atomic space invariant. To allow multiple threads to access an atomic space concurrently, an atomic space handle can be duplicated arbitrarily. Formally:

$$I \sqsubseteq \mathsf{atomic\_space}(I) \qquad \mathsf{atomic\_space}(I) \Rightarrow \mathsf{atomic\_space}(I) * \mathsf{atomic\_space}(I)$$

$$\frac{\{I * P\}\ c\ \{I * Q\}}{\{\mathsf{atomic\_space}(I) * P\}\ \langle c \rangle\ \{\mathsf{atomic\_space}(I) * Q\}}$$

The proof tracks the set of operations in progress (i.e. the threads that have performed the atomic read but have not yet performed the corresponding compare-and-swap) through a *ghost object* readers, an instance of *ghost class* GhostBag, whose specification is shown in Figure 32. The ghost bag is owned by the same atomic space that owns the stack's head field and the linked list of nodes (described by the nodes predicate). The GhostBagHandle predicate that a thread receives when it inserts an element into the ghost bag enables it to "remember" that it has an element in the ghost bag in the interval between the atomic operations. The ghost bag in particular contains, for each operation in progress, the value of the head field that it read. The atomic space additionally holds, for each operation in progress whose head value is out of date, a call permission. As a failed operation removes its element from the ghost bag, it can also extract a call permission from the atomic space, enabling it to try again.

5. LIVENESS

Many programs, such as servers, are not supposed to terminate. Still, they should be *responsive*: if there are pending requests, the server should eventually respond. More generally, a program should always eventually interact with its environment; we call this *liveness*.

For example, consider the program of Figure 33. This program does not terminate; however, it is live.

There is a simple way to encode this basic notion of liveness as a termination property. Suppose we wish to verify that a program always eventually performs an I/O operation. Then it is sufficient to prove that the program terminates, assuming that

```
class OS {
    static predicate IO(int n)
    static void beep()
        req IO(n)
        ens 0 < n ∧ IO(n − 1)
}
```

```
class Main {
    static void iter()
        req IO(n) ∗ cp({[Main.iter]} + n)
        ens false
    { OS.beep(); iter() }
    static void main()
        req IO(n) ∗ cp({[Main.main]})
        ens false
    { iter() }
}
```

Fig. 34. Liveness verification example

the Nth I/O operation causes the program to terminate, for some unknown but fixed N. This can be encoded into a specification of our approach as shown in Figure 34.

The above approach allows one to verify that a program performs an infinite sequence of I/O operations. We may additionally wish to verify that this sequence satisfies certain conditions. For example, we may wish to verify that a server does not starve any of its clients, i.e. that each request is eventually matched by a response. We do not propose a solution to this problem here; however, it seems that combining the approach we propose here for verifying basic liveness (as defined above) with the I/O verification approach we proposed in recent work [Penninckx et al. 2015] would serve as a good basis for addressing this problem.

6. TOOL SUPPORT

We integrated the logic into the program verification tool VeriFast [Vogels et al. 2015], so that since version 18.02 [Jacobs 2018], it now supports modular verification of termination of C and Java programs.

For Java programs, we introduced the method specification clause **terminates**, to indicate that a method should terminate. Furthermore, we introduced the predicate call_perm(ℓ), where ℓ is a list of `Class` objects. The order of the elements in the list is not significant; conceptually, it denotes a multiset. In the current preliminary implementation, we use `Class` objects as level ingredients rather than method names, for simplicity. This coarser granularity was sufficient to encode the examples of the paper, except that for encoding some examples we had to split a class up into multiple classes.

In order to reduce specification overhead for methods that do not perform callbacks, our implementation offers a ghost command that allows a method to produce out of thin air any call permission whose bag of `Class` objects is less than its own `Class` object (considered as a singleton bag). Furthermore, no call permission is consumed for upcalls. In exchange, our implementation consumes at a (non-upcall) call site not just any call permission, but only a call permission qualified by a level greater than or equal to the `Class` object of the method being called:

$$\frac{\{\!\!\{\mathsf{classOf}(o)\}\!\!\} \le \ell \qquad \textbf{instance}\ m(\overline{v})\ \textbf{req}\ P\ \textbf{ens}\ Q}{\vdash_C \{\mathsf{cp}(\ell) * P\}\ o.m(\overline{v})\ \{Q\}}$$

$$\frac{\vdash_C \{P\}\ c\ \{Q\}}{\vdash_C \textbf{instance}\ m(\overline{x})\ \textbf{req}\ P\ \textbf{ens}\ Q\ \{\ c\ \}\ \mathsf{ok}} \qquad \frac{\ell < \{\!\!\{C\}\!\!\}}{\vdash_C \{\mathsf{true}\}\ \textbf{produce\_cp}\ \{\mathsf{cp}(\ell)\}}$$

The VeriFast distribution includes, in the directory examples/java/termination, verified encodings of all of the examples of this paper.[11]

We are not aware of unsoundnesses in our current implementation, except that successful verification does not guarantee absence of deadlock in executions where exceptions occur. Indeed, in Java it is difficult to rule out deadlock completely. This is because the Java language specification allows the VM to throw a VirtualMachineError exception at any time [JLS, Java SE 8 Ed., §11.1]. Furthermore, if an exception reaches the top of a thread, the thread terminates but the program does not. Therefore, if a thread that holds an obligation to send receives a VirtualMachineError and this exception kills the thread, then any thread that waits on this obligation will deadlock. The root problem is that Java does not properly propagate failures. In earlier work, we proposed a language extension that would address this issue [Jacobs and Piessens 2009; Jacobs 2015].

7. DISCUSSION

In this section, we discuss a few ways to relax the approaches of the preceding sections.

The acyclic import graph requirement. In Sec. 2.1, we defined a less-than relation between method names, by ordering them according to textual order in case of methods declared by a single module, and according to the module import relation in case of methods declared by distinct modules. Since we need this less-than relation to be a partial order for our universe of levels to be well-founded, we assumed that the module import graph is acyclic. However, we can support cycles in the import graph by allowing a module author to mark each import as either implying an order relation (call it *strong*), or not (call it *weak*). Then, we only consider strong import edges when defining the less-than relation on method names, and we require acyclicity of the strong import graph only. This does mean that a call of a static method along a weak import edge is not an upcall and will not pass the level check if the methods have their default specifications as suggested by our specification style; instead, the caller's specification will have to reflect the fact that it effectively performs a callback.

Conditional termination. The approaches of the preceding sections verify unconditional termination. However, it is easy to relax this so that specifications can express conditional termination: simply add a top element $\top$ to the universe of levels, which is greater than all levels, including itself. (This means the less-than relation is no longer an order relation.) Then, in a method's level clause, or to qualify a call permission, use a conditional expression that evaluates to either $\top$ or some well-founded level, depending on whether termination of the method is required under the particular circumstances. Of course, care must then be taken that $\top$-qualified call permissions do not leak to methods that should terminate.

8. RELATED WORK

We are not aware of existing approaches for modular specification and verification of termination of object-oriented programs. However, work on modular verification of termination in different settings does exist.

The proof assistant Coq includes a pure functional programming language with higher-order functions. Coq checks that all functions terminate. However, Coq's type system prevents a function from being passed as an argument to itself. Our approach supports methods that call themselves through dynamic binding, and can prove their termination.

[11]Browsable online at https://github.com/verifast/verifast/tree/18.02/examples/java/termination

Koka [Leijen 2014] is a functional programming language with effect inference, including the divergence effect. However, the inference algorithm is limited: it rules out recursion through the heap, which our approach supports.

Dafny [Leino 2010] is a programming language that supports verification of termination, with powerful metrics. However, Dafny does not support dynamic binding of method calls.

Most similar to ours is the work, e.g. [Darvas and Müller 2006; Rudich et al. 2008; Leino and Middelkoop 2009], on proving well-definedness of specifications for object-oriented programs where the specifications themselves involve calls of methods of the program being specified. In most such approaches, in order to ensure that such specifications make sense and that axioms generated from such specifications are consistent, proof obligations are imposed to verify that methods called from specifications are *pure* (i.e., side-effect-free) and that they *terminate*. The levels we associate with a data structure can be seen as a refinement of the *depth of the ownership tree* (a natural number) used as a recursion measure by some of this work [Darvas and Müller 2006; Leino and Middelkoop 2009]. In these approaches, the ownership graph is frozen for the duration of the execution of a pure method, so if calls descend down an ownership tree, they terminate. In our approach, however, to support non-pure methods that create new data structures composed of lower-layer classes, we track not just the number of layers of which a data structure consists; rather, we also effectively track which modules implement each layer.

CFML. However, most related to ours in terms of what it achieves is the work by Charguéraud on CFML [Charguéraud 2011], an approach for the modular verification of total correctness of sequential higher-order imperative ML programs. In the present paper, we add features for verifying termination to a typical existing program logic for partial correctness. In contrast, CFML removes from such a typical logic the features that allow the verification of non-terminating programs in the first place. In particular, CFML's sole rule for function calls is essentially as follows:

$$\frac{\{P\}\ c[v/x]\ \{Q\}}{\{P\}\ (\lambda x.\ c)\ v\ \{Q\}}$$

Furthermore, while CFML allows the use of (essentially) Hoare triples as assertions, these cannot be self-recursive; for example, the predicate

$$P(f) = \forall f'.\ \{P(f')\}\ f(f')\ \{\mathbf{false}\}$$

is not expressible in CFML (whereas it is trivially expressible as an interface or a class in our logic). This means that there is no need for step indexing-like measures to assign a meaning to CFML's Hoare triples, which in turn means that if a command satisfies a Hoare triple, it terminates (whenever started in a state that satisfies the precondition).

Notice that CFML's proof rule for function calls means that proof trees are as deep as the depth of recursion of the program (i.e., generally infinitely deep, even though, since the program terminates, each path is finite). This is not a problem, since CFML is shallowly embedded inside the higher-order proof assistant Coq, which means Coq's facilities for (well-founded) recursive definitions can be used to express the proof trees. This yields a very powerful and very elegant logic, admittedly more elegant than the one we propose here, since there is nothing like a level clause or a level check built into the logic. Indeed, it seems that translations into ML of the examples of Sec. 2 can be verified using CFML without the need to introduce anything like our levels.

Indeed, it seems that all proofs in the logic of Sec. 3 can be translated into CFML proofs. The main complexity in such a translation is in translating the types of the program. Since interface and class definitions within a module are generally recursive (since the interface and class names being defined can be used as parameter or return types of the methods), this is not entirely trivial. One approach is to translate the interfaces and classes of a module into a set of mutually recursively defined CFML predicates, each indexed by a level. For example, the interface

$$\textbf{interface}\ \mathsf{Set}\ \{\ \textbf{bool}\ \mathsf{intersects}(\mathsf{Set\ other})\ \textbf{req}\ P_X^{\mathsf{int}}\ \textbf{level}\ \ell_X^{\mathsf{int}}\ \textbf{ens}\ Q_X^{\mathsf{int}}\ \}$$

(where X represents the free variables of the method specification linking the precondition, the level, and the postcondition) can be translated into a CFML predicate roughly like this:

$$\mathsf{Set}_\ell(f) = \forall \mathsf{other}, X.\ \ell = \ell_X^{\mathsf{int}} \Rightarrow \mathsf{Set}_{<\ell}(\mathsf{other}) \Rightarrow \{P_X^{\mathsf{int}}\}\ f(\mathsf{other})\ \{Q_X^{\mathsf{int}}\}$$

where $\mathsf{Set}_{<\ell}(f)$ means $\forall \ell' < \ell.\ \mathsf{Set}_{\ell'}(f)$. This works because the fact that the program was verified using our logic means that nested calls will be at lower levels. Notice, however, that this does not work for return types; one way to work around this is to first translate the program and the proof into continuation-passing style (CPS).

The fact that a translation of our proofs into CFML seems to generally require a CPS transformation, suggests an avenue for finding a proof that would be more elegant in our approach than in CFML. However, so far we have failed to find such a proof.

Concurrency. Leino et al. [2010] proposed the approach for verifying absence of deadlock in programs with channels and locks which our approach incorporates. Boström and Müller [2015] proposed an approach for verifying *finite blocking* of nonterminating concurrent programs. da Rocha Pinto et al. [2016] proposed an approach for modular termination verification of lock-free fine-grained concurrent algorithms. None of these works address the issue of modular verification of absence of infinite recursion in the presence of dynamic binding, and none address the issue of allowing transparent introduction of private locks.

Call permissions. Our call permissions are similar to the way Atkey [2011] extends separation logic for amortized resource (e.g., heap space or execution time) analysis. Since he is interested in enforcing particular resource bounds, he does not qualify his permissions by arbitrary ordinals. Atkey's resource analysis has been implemented in CFML [Charguéraud and Pottier 2011].

Our call permissions can be seen as a straightforward extension of Atkey's resources. Our contribution, then, lies in the combination of this well-known idea with our levels scheme to enable each module to create arbitrary numbers of call permissions.

Hoffmann et al. [2013] propose a concurrent separation logic with *tokens*, similar to Atkey's resources, to prove lock-freedom of concurrent data structures. In their approach, each thread starts with a particular number of tokens, given as a natural number in terms of the number of threads in the system, and consumes one token per loop iteration. When a thread needs to retry an operation because of interference from another thread, a token is transfered from the interfering thread to allow it to do so. This way, they prove that the total number of retries in the system is bounded, which implies lock-freedom.

In Sec. 4.4, we prove termination of programs that involve compare-and-swap loops in a similar way. Again, by qualifying our call permissions with ordinals chosen according to our levels scheme, we achieve modular token accounting.

9. CONCLUSION

We propose an approach for the modular specification and verification of total correctness properties of single-threaded and multithreaded object-oriented programs involving dynamically bound method calls. As far as we know, it is the first such approach.

We propose a specification style that does not constrain implementations unnecessarily. The style enables any module to acquire private locks. The approach supports compare-and-swap loops. We sketch an encoding of liveness properties.

We have implemented tool support for our approach in a verification tool and validated it on a handful of small but challenging example programs. Further experimentation is needed, however, to see if our approach conveniently handles all program patterns.

APPENDIX

A. FORMALIZATION AND SOUNDNESS PROOF

In this appendix we formalize the key concepts and prove key properties, including soundness of the program logics assumed in the paper. We formalize the material of Sec. 2 in Sec. A.1 and the material of Sec. 3 and 4 in Sec. A.2.

A.1. Levels (Sec. 2)

In this section, we formalize the notions of module, module specification, and module correctness used in Sec. 2, and we prove that if a well-typed program's modules are correct, then its executions terminate. Furthermore, we formalize the concepts used in the specification styles of Sec. 2.

Because Sec. 2 considers only programs that do not update the heap, for the formalization of this section we take a Featherweight Java [Igarashi et al. 2001]-like approach, where values of class or interface type are of the form $\mathbf{new}\ C(\overline{v}) \triangleleft I(\overline{v}')$, i.e. they are tuples of field values $\overline{v}$ and ghost field values $\overline{v}'$ rather than references pointing into a heap. (This means the formal language of this section does not support an object identity comparison operator; however, the formal language of the next section does.)

A machine-checked encoding of the definitions and proofs of this section into the logic of the Coq proof assistant is available [Jacobs 2018].

A.1.1. Programs. We assume disjoint infinite sets $\mathcal{I}$, $\mathcal{C}$, $\mathcal{M}$, $\mathcal{F}$, $\mathcal{G}$, and $\mathcal{X}$ of interface names, class names, unqualified method names, field names, ghost field names, and variable names, ranged over by symbols I, C, m, f, g, and x, respectively.

We define the set of *method names* as $\mathit{MethodNames} = \{C.m \mid C \in \mathcal{C}, m \in \mathcal{M}\}$. A *method precedence order* $\prec \subseteq \mathit{MethodNames} \times \mathit{MethodNames}$ (or *method precedence* for short) is a well-founded partial order on method names that is consistent with a partial order on class names; specifically, if it relates any two method names qualified by different classes C and C', then it relates all method names qualified by those classes in the same way: $(\exists m, m'.\ C.m \prec C'.m') \Rightarrow (\forall m, m'.\ C.m \prec C'.m')$. We call the induced partial order on class names the *induced import order*. We write $C \prec C'$ to mean $\exists m, m'.\ C.m \prec C'.m'$.

We assume a set of types $\mathcal{T}$, ranged over by τ, including at least the interface names, the class names, the type $\mathbf{bool}$ of booleans, and the unit type $\mathbf{void}$: $\mathcal{I}, \mathcal{C} \subseteq \mathcal{T}, \mathbf{bool}, \mathbf{void} \in \mathcal{T}$.

We assume a set $\mathit{Operators}$ of operators, ranged over by op, and a function $\mathsf{opType} : \mathit{Operators} \to (\mathcal{T}^* \times \mathcal{T})$ that specifies the parameter types and the result type of each operator.

We assume a set $\mathbb{V}$ of values, ranged over by v, including at least the booleans ($\mathbb{B} \subseteq \mathbb{V}$) and the unit value $() \in \mathbb{V}$ and closed under object creation: if $\overline{v}, \overline{v}' \in \mathbb{V}$ then $\mathbf{new}\ C(\overline{v}) \triangleleft I(\overline{v}') \in \mathbb{V}$. (Not all of these objects are well-typed in the context of a given program; see below.)

We assume an interpretation function $[\![-]\!] : \mathcal{T} \to \mathcal{P}(\mathbb{V})$ for types, that maps a type to the set of its values. We assume $[\![\mathbf{bool}]\!] = \mathbb{B}$, $[\![\mathbf{void}]\!] = \{()\}$, $\forall C \in \mathcal{C}.\ [\![C]\!] = \{\mathbf{new}\ C(\overline{v}) \triangleleft$

$$
\begin{array}{rcl}
e & ::= & x \mid op(\overline{e}) \mid e.f \mid e.g \mid v \\
c & ::= & e \mid C.m(\overline{e}) \mid e.m(\overline{e}) \mid \textbf{new}\ C(\overline{e}) \\
 & & \mid \tau\ x := c; c \mid \textbf{if}\ e\ \textbf{then}\ e\ \textbf{else}\ e \mid \textbf{assert}\ e \\
\mathit{gfspec} & ::= & \textbf{ghost\_field}\ \tau\ g \\
\beta & ::= & \textbf{static} \mid \textbf{instance} \\
\mathit{mspec} & ::= & \beta\ \tau\ m(\overline{\tau\ x})\ \textbf{req}\ e\ \textbf{level}\ e\ \textbf{ens}\ e \\
\mathit{itf} & ::= & \textbf{interface}\ I\ \{\ \overline{\mathit{gfspec}}\ \overline{\mathit{mspec}}\ \} \\
\mathit{cspec} & ::= & \textbf{class}\ C\ \{\ \overline{\mathit{mspec}}\ \} \\
\mathit{gf} & ::= & \mathit{gfspec} = e \\
\mathit{meth} & ::= & \mathit{mspec}\ \{\ c\ \} \\
\mathit{class} & ::= & \textbf{class}\ C(\overline{\tau\ f})\ \textbf{imports}\ \overline{\mathit{itf}}\ \overline{\mathit{cspec}}\ \textbf{implements}\ I\ \{\ \overline{\mathit{gf}}\ \overline{\mathit{meth}}\ \} \\
\mathit{prog} & ::= & \overline{\mathit{itf}}\ \overline{\mathit{class}}
\end{array}
$$

Fig. 35. Syntax of the formal language

$I(\overline{v}') \mid \exists \overline{v}, I, \overline{v}'\}$, and $\forall I \in \mathcal{I}.\ [\![I]\!] = \{\textbf{new}\ C(\overline{v}) \triangleleft I(\overline{v}') \mid \exists C, \overline{v}, \overline{v}'\}$. (We define a more precise interpretation function that interprets class and interface types more precisely below.)

We assume an interpretation function $[\![\cdot]\!]_{\prec} : \mathit{Operators} \to \mathbb{V}^* \to \mathbb{V}$ for operators that is consistent with opType. The interpretation function may depend on a *program method precedence order* $\prec$.

We assume operators do not create objects. More specifically, consider any well-typed *term* t built from some set of free variables $\overline{x}$ of class and interface types $\overline{\tau}$ and operator applications. Then, consider any values $\overline{v} \in [\![\overline{\tau}]\!]$ for $\overline{x}$. If the type of t is a class or interface type, then the value of $t[\overline{v}/\overline{x}]$ is in $\overline{v}$.

For simplicity, we also assume that non-object values do not "contain" objects. Specifically, consider any well-typed term t built from some set of free variables $\overline{x}$ of types $\overline{\tau}$ and operator applications. Then, consider any values $\overline{v} \in [\![\overline{\tau}]\!]$ for $\overline{x}$. If the value v of $t[\overline{v}/\overline{x}]$ is an object, then v is the value of a variable $x \in \overline{x}$ whose type is a class or interface name.

We assume a type τ_{levels} of levels. We define the set of levels $\mathbb{L} = [\![\tau_{\text{levels}}]\!]$; we assume it is equipped with a well-founded order $<_{\prec}$, which may depend on the method precedence.

The syntax of programs is defined in Fig. 35. The expressions are the variables x, the operator expressions $op(\overline{e})$, the field dereferences, the ghost field dereferences, and the literal values. The latter appear only during execution (but some literal values, excluding in particular the objects, can be encoded as nullary operators). The commands are the expressions e, the static method calls $C.m(\overline{e})$, the instance method calls $e.m(\overline{e})$, the object creation commands, the let commands $\tau\ x := c; c'$, the conditional commands, and the assert command.

We define a *context* $\Gamma = \overline{\mathit{itf}}\ \overline{\mathit{cspec}}\ \overline{\mathit{class}}\ \overline{x : \tau}$ as a sequence of interfaces, class specifications, classes, and variable typings.

Given a context Γ, we define the *context method precedence relation* R_Γ as follows: $C\ R_\Gamma\ C'$ iff a class specification or class **class** $C \cdots$ appears before class (not class specification) **class** $C'(\cdots)\ \cdots$ in Γ, and, for each class C and method names m and m', $C.m\ R_\Gamma\ C.m'$ iff **class** $C(\cdots)\ \cdots\ \{\ \cdots\ \beta\ \tau\ m(\cdots)\ \cdots\ \beta'\ \tau'\ m'(\cdots)\ \cdots\ \} \in \Gamma$. If R_Γ is antireflexive, we define the *context method precedence order* $\prec_\Gamma$ to be equal to R_Γ; otherwise, we leave $\prec_\Gamma$ undefined.

We define the *class context* of a class class as the context $\Gamma = \overline{\mathit{itf}}\ \overline{\mathit{cspec}}\ \mathit{class}$ consisting of the class' imported interfaces and class specifications and the class itself. We define the *class method precedence order* $\prec^{\mathsf{c}}_{\mathit{class}}$ as the context precedence order $\prec_\Gamma$ for the

$$\frac{\tau \notin \mathcal{I} \cup \mathcal{C}}{\Gamma \vdash \tau} \qquad \frac{\textbf{interface}\ I\ \cdots \in \Gamma}{\Gamma \vdash I} \qquad \frac{\textbf{class}\ C(\cdots)\ \cdots\ \in \Gamma}{\Gamma \vdash C} \qquad \frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}$$

$$\frac{\mathsf{opType}(op) = (\overline{\tau}, \tau) \qquad \Gamma \vdash \overline{e} : \overline{\tau}}{\Gamma \vdash op(\overline{e}) : \tau} \qquad \frac{\Gamma \vdash e : C \qquad \textbf{class}\ C(\ldots, \tau\ f, \ldots)\ \cdots \in \Gamma}{\Gamma \vdash e.f : \tau}$$

$$\frac{\Gamma \vdash e : I \qquad \textbf{interface}\ I\ \{\ \cdots\ \textbf{ghost\_field}\ \tau\ g\ \cdots\ \} \in \Gamma}{\Gamma \vdash e.g : \tau}$$

$$\frac{\textbf{class}\ C \cdots\ \{\ \cdots\ \textbf{static}\ \tau\ m(\overline{\tau\ x})\ \cdots\ \} \in \Gamma \qquad \Gamma \vdash \overline{e} : \overline{\tau}}{\Gamma \vdash C.m(\overline{e}) : \tau}$$

$$\frac{\Gamma \vdash e : I \qquad \textbf{interface}\ I\ \{\ \cdots\ \tau\ m(\overline{\tau\ x})\ \cdots\ \} \in \Gamma \qquad \Gamma \vdash \overline{e} : \overline{\tau}}{\Gamma \vdash e.m(\overline{e}) : \tau}$$

$$\frac{\textbf{class}\ C(\overline{\tau\ f})\ \cdots \in \Gamma \qquad \Gamma \vdash \overline{e} : \overline{\tau}}{\Gamma \vdash \textbf{new}\ C(\overline{e}) : C} \qquad \frac{\textbf{class}\ C \cdots\ \textbf{implements}\ I\ \cdots \in \Gamma \qquad \Gamma \vdash e : C}{\Gamma \vdash e : I}$$

$$\frac{\Gamma \vdash c : \tau \qquad x \notin \mathrm{dom}(\Gamma) \qquad \Gamma \vdash \tau \qquad \Gamma, x : \tau \vdash c' : \tau'}{\Gamma \vdash \tau\ x := c; c' : \tau'} \qquad \frac{\Gamma \vdash e : \textbf{bool} \qquad \Gamma \vdash c_1, c_2 : \tau}{\Gamma \vdash \textbf{if}\ e\ \textbf{then}\ c_1\ \textbf{else}\ c_2 : \tau}$$

$$\frac{\Gamma \vdash e : \textbf{bool}}{\Gamma \vdash \textbf{assert}\ e : \textbf{void}}$$

Fig. 36. Well-typedness of types and expressions

class context. It relates the methods of the class by their textual order. Furthermore, it relates all imported classes' methods to all of the class' methods.

Given a program $prog$, we define the *program method precedence relation* R^{p}_{prog} as the transitive closure of the union of the class method precedence orders of the classes of $prog$. If this relation is antireflexive, we define the *program method precedence order* $\prec^{\mathsf{p}}_{prog}$ to be equal to R^{p}_{prog}; otherwise, we leave it undefined.

We define well-typedness of programs and program elements under a context Γ in Figs. 36 and 37. Notice the following: literal value expressions are never well-typed; a class type C may appear only in the body of a method of class C (since the typing rule for class types requires the class (not just the class specification) to be present in the context); class ghost fields or instance methods that do not implement an interface member are not supported; the only free variables allowed in ghost field initializers are the field names (we assume an injection $\mathsf{fieldVar} : \mathcal{F} \to \mathcal{X}$ for this purpose).

Notice furthermore that the final premise of the rule for well-typedness of a program states that the transitive closure of the union of the class method precedence orders of the classes of the program is antireflexive. (If the classes are well-typed, this is equivalent to saying that the class import graph is acyclic.) It follows that the program method precedence order of a well-typed program is well-defined.

In Fig. 38 we define three concepts: well-typedness of a context $\Gamma = \overline{itf}\ \overline{cspec}\ \overline{class}$; the value $[\![e]\!]_{\Gamma,\prec} \in \mathbb{V}$ of a closed expression e under a well-typed context Γ and method precedence $\prec$; and the interpretation $[\![\tau]\!]_{\Gamma,\prec}$ of a type τ under a well-typed context Γ and method precedence $\prec$.

$$\frac{\begin{array}{c}\overline{x}, \mathsf{result\ distinct}\\ \Gamma \vdash \tau, \overline{\tau} \qquad \Gamma, \overline{x} : \overline{\tau} \vdash P : \mathbf{bool} \qquad \Gamma, \overline{x} : \overline{\tau} \vdash e : \tau_{\mathrm{levels}} \qquad \Gamma, \overline{x} : \overline{\tau}, \mathsf{result} : \tau \vdash Q : \mathbf{bool}\end{array}}{\Gamma \vdash_{\tau_{\mathsf{this}}} \mathbf{static}\ \tau\ m(\overline{\tau\ x})\ \mathbf{req}\ P\ \mathbf{level}\ e\ \mathbf{ens}\ Q\ \mathsf{wt}}$$

$$\frac{\begin{array}{c}\mathsf{this}, \overline{x}, \mathsf{result\ distinct} \qquad \Gamma \vdash \tau, \overline{\tau} \qquad \Gamma, \mathsf{this} : \tau_{\mathsf{this}}, \overline{x} : \overline{\tau} \vdash P : \mathbf{bool}\\ \Gamma, \mathsf{this} : \tau_{\mathsf{this}}, \overline{x} : \overline{\tau} \vdash e : \tau_{\mathrm{levels}} \qquad \Gamma, \mathsf{this} : \tau_{\mathsf{this}}, \overline{x} : \overline{\tau}, \mathsf{result} : \tau \vdash Q : \mathbf{bool}\end{array}}{\Gamma \vdash_{\tau_{\mathsf{this}}} \mathbf{instance}\ \tau\ m(\overline{\tau\ x})\ \mathbf{req}\ P\ \mathbf{level}\ e\ \mathbf{ens}\ Q\ \mathsf{wt}}$$

$$\frac{\overline{g}\ \mathsf{distinct} \qquad \Gamma \vdash \overline{\tau} \qquad \overline{mspec}.m\ \mathsf{distinct} \qquad \overline{mspec}.\beta \subseteq \{\mathbf{instance}\} \qquad \Gamma \vdash_I \overline{mspec}\ \mathsf{wt}}{\Gamma \vdash \mathbf{interface}\ I\ \{\ \overline{\mathbf{ghost\_field}\ \tau\ g}\ \overline{mspec}\ \}\ \mathsf{wt}}$$

$$\frac{\begin{array}{c}cspec = \mathbf{class}\ C\ \{\ \overline{mspec}\ \}\\ \overline{mspec}.m\ \mathsf{distinct} \qquad \overline{mspec}.\beta \subseteq \{\mathbf{static}\} \qquad \Gamma \vdash_C \overline{mspec}\ \mathsf{wt}\end{array}}{\Gamma \vdash cspec\ \mathsf{wt}}$$

$$\frac{\Gamma \vdash_C mspec\ \mathsf{wt} \qquad mspec = \mathbf{static}\ \tau\ m(\overline{\tau\ m})\ \cdots\ \{\ c\ \} \qquad \Gamma, \overline{x} : \overline{\tau} \vdash c : \tau}{\Gamma \vdash_C mspec\ \{\ c\ \}\ \mathsf{wt}}$$

$$\frac{\Gamma \vdash_C mspec\ \mathsf{wt} \qquad mspec = \mathbf{instance}\ \tau\ m(\overline{\tau\ m})\ \cdots\ \{\ c\ \} \qquad \Gamma, \mathsf{this} : C, \overline{x} : \overline{\tau} \vdash c : \tau}{\Gamma \vdash_C mspec\ \{\ c\ \}\ \mathsf{wt}}$$

$$\frac{\begin{array}{c}class = \mathbf{class}\ C(\overline{\tau\ f})\ \mathbf{imports}\ \overline{itf}\ \overline{cspec}\ \mathbf{implements}\ I\ \{\ \overline{\mathbf{ghost\_field}\ \tau'\ g = e}\ \overline{meth}\ \}\\ \overline{itf}.I\ \mathsf{distinct} \qquad \overline{itf} \vdash \overline{itf}\ \mathsf{wt} \qquad \overline{cspec}.C, C\ \mathsf{distinct}\\ \overline{itf} \vdash \overline{cspec}\ \mathsf{wt} \qquad \overline{f}\ \mathsf{distinct} \qquad \overline{itf} \vdash \overline{\tau} \qquad \mathbf{interface}\ I\ \{\ \overline{\mathbf{ghost\_field}\ \tau'\ g}\ \overline{mspec}\ \} \in \overline{itf}\\ \overline{itf}, \overline{f} : \overline{\tau} \vdash \overline{e} : \overline{\tau'} \qquad \{mspec \mid mspec\ \{\ c\ \} \in \overline{meth}, mspec.\beta = \mathbf{instance}\} = \overline{mspec}\\ \overline{itf}, \overline{cspec}, class \vdash_C \overline{meth}\ \mathsf{wt}\end{array}}{\vdash class\ \mathsf{wt}}$$

$$\begin{array}{l}\mathsf{spec}(\mathbf{class}\ C(\cdots)\ \cdots\ \{\ \overline{gf}\ \overline{meth}\ \}) = \mathbf{class}\ C\ \{\ \overline{mspec}\ \}\\ \quad \mathbf{where}\ \overline{mspec} = \{mspec \mid mspec\ \{\ c\ \} \in \overline{meth}, mspec.\beta = \mathbf{static}\}\end{array}$$

$$\frac{\begin{array}{c}\overline{itf}.I\ \mathsf{distinct} \qquad \overline{class}.C\ \mathsf{distinct} \qquad \overline{itf} \vdash \overline{itf}\ \mathsf{wt} \qquad \vdash \overline{class}\ \mathsf{wt}\\ \forall \mathbf{class}\ C(\cdots)\ \mathbf{imports}\ \overline{itf}'\ \overline{cspec}\ \cdots \in \overline{class}.\ \overline{itf}' \subseteq \overline{itf} \land \overline{cspec} \subseteq \mathsf{spec}(\overline{class})\\ (\cup_{class \in \overline{class}} \prec^{\mathsf{c}}_{class})^{+}\ \mathsf{antireflexive}\end{array}}{\vdash \overline{itf}\ \overline{class}\ \mathsf{wt}}$$

Fig. 37. Well-typedness of method specifications, interfaces, class specifications, classes, and programs

The examples of Sec. 2 can be turned into programs of the formal language by introducing temporary variables for subexpressions, making field dereferences on this (outside of ghost field initializers) explicit, introducing dummy empty interfaces for classes that do not explicitly implement one, adding interface imports, and expanding each class import to include the specifications of the static methods. (Note, however, that examples that do not perform abstract object construction through a factory method cannot be translated since the formal language supports the creation of objects of class C only within class C.)

$$\frac{\overline{\mathit{itf}}.I \text{ distinct} \qquad \overline{\mathit{cspec}}.C, \overline{\mathit{class}}.C \text{ distinct} \qquad \overline{\mathit{itf}} \vdash \overline{\mathit{itf}} \text{ wt} \qquad \overline{\mathit{itf}} \vdash \overline{\mathit{cspec}} \text{ wt} \qquad \vdash \overline{\mathit{class}} \text{ wt} \\ \forall \mathbf{class}\ C(\cdots)\ \mathbf{imports}\ \overline{\mathit{itf}}'\ \overline{\mathit{cspec}}'\ \cdots \in \overline{\mathit{class}}.\ \overline{\mathit{itf}}' \subseteq \overline{\mathit{itf}} \land \overline{\mathit{cspec}}' \subseteq \overline{\mathit{cspec}} \cup \mathsf{spec}(\overline{\mathit{class}})}{\vdash \overline{\mathit{itf}}\ \overline{\mathit{cspec}}\ \overline{\mathit{class}} \text{ wt}}$$

$$[\![v]\!]_{\Gamma,\prec} = v \qquad\qquad [\![op(\overline{e})]\!]_{\Gamma,\prec} = [\![op]\!]_{\prec}([\![\overline{e}]\!]_{\Gamma,\prec})$$

$$\frac{[\![e]\!]_{\Gamma,\prec} = \mathbf{new}\ C(\overline{v})\ \cdots \qquad \mathbf{class}\ C(\overline{\tau\ f})\ \cdots \in \Gamma}{[\![e.f_i]\!]_{\Gamma,\prec} = v_i}$$

$$\frac{[\![e]\!]_{\Gamma,\prec} = \mathbf{new}\ C(\overline{v}) \triangleleft I(\overline{v}') \qquad \mathbf{interface}\ I\ \{\ \overline{\mathbf{ghost\_field}\ \tau\ g}\ \cdots\ \}}{[\![e.g_i]\!]_{\Gamma,\prec} = v_i'}$$

$$\frac{\mathbf{class}\ C(\overline{\tau\ f})\ \cdots\ \mathbf{implements}\ I\ \{\ \overline{\mathbf{ghost\_field}\ \tau'\ g = e}\ \cdots\ \} \in \Gamma \\ \overline{v} \in [\![\overline{\tau}]\!]_{\Gamma,\prec} \qquad \overline{v}' = [\![\overline{e}[\overline{v}/\overline{f}]]\!]_{\Gamma,\prec}}{\mathbf{new}\ C(\overline{v}) \triangleleft I(\overline{v}') \in [\![C]\!]_{\Gamma,\prec}}$$

$$\frac{\mathbf{class}\ C(\cdots)\ \cdots \notin \Gamma \qquad v \in [\![C]\!]}{v \in [\![C]\!]_{\Gamma,\prec}}$$

$$\frac{\mathbf{class}\ C(\overline{\tau\ f})\ \cdots\ \mathbf{implements}\ I\ \{\ \overline{\mathbf{ghost\_field}\ \tau'\ g = e}\ \cdots\ \} \in \Gamma \\ \overline{v} \in [\![\overline{\tau}]\!]_{\Gamma,\prec} \qquad \overline{v}' = [\![\overline{e}[\overline{v}/\overline{f}]]\!]_{\Gamma,\prec}}{\mathbf{new}\ C(\overline{v}) \triangleleft I(\overline{v}') \in [\![I]\!]_{\Gamma,\prec}}$$

$$\frac{\mathbf{class}\ C(\cdots)\ \cdots \notin \Gamma \qquad \mathbf{interface}\ I\ \{\ \overline{\mathbf{ghost\_field}\ \tau\ g}\ \cdots\ \} \in \Gamma \qquad \overline{v}' \in [\![\overline{\tau}]\!]_{\Gamma,\prec}}{\mathbf{new}\ C(\overline{v}) \triangleleft I(\overline{v}') \in [\![I]\!]_{\Gamma,\prec}}$$

$$\frac{\mathbf{class}\ C(\cdots)\ \cdots \notin \Gamma \qquad \mathbf{interface}\ I\ \cdots \notin \Gamma}{\mathbf{new}\ C(\overline{v}) \triangleleft I(\overline{v}') \in [\![I]\!]_{\Gamma,\prec}} \qquad\qquad \frac{\tau \notin \mathcal{C} \cup \mathcal{I} \qquad v \in [\![\tau]\!]}{v \in [\![\tau]\!]_{\Gamma,\prec}}$$

Fig. 38. Well-typed context; interpretation of expressions and types under a context and method precedence

A.1.2. Operational semantics. We define a big-step relation $c \Downarrow_{prog} O$, where c is a closed command and $O \in \mathbb{V} \cup \{\top\}$ is an *outcome*. The outcome v denotes successful termination with result value v; the outcome $\top$ denotes divergence. We define the relation by interpreting the inference rules in Fig. 39 coinductively [Nakata and Uustalu 2009]. (Read each $\Downarrow$ symbol as $\Downarrow_{prog}$. We identify closed expressions with their value under $prog$ and the program method precedence order. Also, for $b \in \mathbb{B}$, we identify **if** b **then** c **else** c' with its reduction to c or c'. Also, a class $class$ mentioned as a premise denotes $class \in prog$.) Notice that for a divergent command c, we have not just $c \Downarrow \top$, but $c \Downarrow v$ as well, for every value v. This imprecision is harmless since we will be proving termination.

A.1.3. Proof rules. We define *run-time well-typedness* of an expression e, denoted $\Gamma \vdash^{\mathsf{rt}}_{\prec} e : \tau$, by the same inference rules used to define well-typedness, plus an additional rule for values:

$$\frac{v \in [\![\tau]\!]_{\Gamma,\prec}}{\Gamma \vdash^{\mathsf{rt}}_{\prec} v : \tau}$$

$$v \Downarrow v \qquad \begin{array}{c} \textbf{class}\ C\ \cdots\ \{\ \cdots\ \textbf{static}\ \tau\ m(\overline{\tau\ x})\ \cdots\ \{\ c\ \}\ \cdots\ \} \qquad c[\overline{v}/\overline{x}] \Downarrow O \\ \hline\hline C.m(\overline{v}) \Downarrow O \end{array}$$

$$\begin{array}{c} o \in [\![C]\!] \qquad \textbf{class}\ C\ \cdots\ \{\ \cdots\ \textbf{instance}\ \tau\ m(\overline{\tau\ x})\ \cdots\ \{\ c\ \}\ \cdots\ \} \qquad c[o/\mathsf{this}, \overline{v}/\overline{x}] \Downarrow O \\ \hline\hline o.m(\overline{v}) \Downarrow O \end{array}$$

$$\begin{array}{c} \textbf{class}\ C(\overline{\tau\ f})\ \cdots\ \textbf{implements}\ I\ \{\ \overline{\textbf{ghost\_field}\ \tau'\ g = e}\ \cdots\ \} \\ \hline\hline \textbf{new}\ C(\overline{v}) \Downarrow \textbf{new}\ C(\overline{v}) \triangleleft I(\overline{e}[\overline{v}/\overline{f}]) \end{array} \qquad \begin{array}{c} c \Downarrow v \qquad c'[v/x] \Downarrow O \\ \hline\hline \tau\ x := c; c' \Downarrow O \end{array}$$

$$\begin{array}{c} c \Downarrow \top \\ \hline\hline \tau\ x := c; c' \Downarrow \top \end{array} \qquad \textbf{assert}\ \mathsf{true} \Downarrow () \qquad \textbf{assert}\ \mathsf{false} \Downarrow \top$$

Fig. 39. Coinductive big-step operational semantics

Building on this, we define *run-time well-typedness* of a command by replacing well-typedness of expressions by run-time well-typedness in the rules defining well-typedness of commands.

We define the command correctness judgment $\Gamma, \ell \vdash_{\prec} c\ \{Q\}$, where Γ is a well-typed context, $\ell \in \mathbb{L}$ is the current level, $\prec$ is a method precedence to be used to evaluate expressions and interpret types, c is a closed command, and Q is a well-typed postcondition (a boolean expression whose only free variable is result). Our command correctness judgments do not mention a precondition; it is implicitly true. We also define correctness of a method, a class, and a program. We define the judgments inductively by the inference rules in Fig. 40. Notice that a class' methods are verified under an arbitrary method precedence relation that extends the class method precedence relation, and that a well-typed program is correct if each class is correct and the program is well-typed, which implies that the class import relation is acyclic.

A.1.4. Soundness. We fix a program prog and we assume $\vdash \mathit{prog}\ \mathsf{correct}$. We prove that if a well-typed command is correct, then it does not diverge and its result value satisfies its postcondition:

THEOREM A.1. *For all ℓ, c, τ, Q, O, if $\mathit{prog} \vdash c : \tau$ and $\mathit{prog}, \ell \vdash_{\prec_{\mathit{prog}}} c\ \{Q\}$ and $c \Downarrow_{\mathit{prog}} O$, then $\exists v.\ O = v \wedge Q[v/\mathsf{result}]$.*

PROOF. By well-founded induction on ℓ and nested induction on the derivation of $\mathit{prog}, \ell \vdash_{\prec_{\mathit{prog}}} c\ \{Q\}$. □

A.1.5. Specification Style. The development above is parameterized over the set of types and operators, and their interpretations. Operator interpretations may depend on the program method precedence order $\prec$. In this subsection, we define the types and operators required to apply the specification style of Sec. 2, and their interpretations for a given value of $\prec$.

For any set X, we define the set $\mathit{Multisets}(X) \subseteq X \to \mathbb{N}$ of (finite) multisets of elements of X (also called multisets over X) inductively as follows:

$$\mathbf{0} \in \mathit{Multisets}(X) \qquad \frac{M \in \mathit{Multisets}(X)}{M \uplus \{\!\!\{x\}\!\!\} \in \mathit{Multisets}(X)}$$

where $\mathbf{0}$ is defined as $\lambda\_.\ 0$, $\{\!\!\{x\}\!\!\}$ as $\mathbf{0}[x := 1]$, and $M \uplus M'$ as $\lambda x.\ M(x) + M'(x)$. We define $\{\!\!\{a, b, c\}\!\!\}$ as $\mathbf{0} \uplus \{\!\!\{a\}\!\!\} \uplus \{\!\!\{b\}\!\!\} \uplus \{\!\!\{c\}\!\!\}$. We say $x \in M$ iff $M(x) > 0$.

$$\Gamma, \ell \vdash_{\prec} v \, \{\mathsf{result} = v\} \qquad \frac{\begin{array}{c}\textbf{class}\ C \cdots \ \{\ \cdots\ \textbf{static}\ \tau\ m(\overline{\tau\ x})\ \textbf{req}\ P\ \textbf{level}\ e\ \textbf{ens}\ Q\ \cdots\ \} \in \Gamma \\ P[\overline{v}/\overline{x}] \qquad e[\overline{v}/\overline{x}] <_{\prec} \ell\end{array}}{\Gamma, \ell \vdash_{\prec} C.m(\overline{v})\ \{Q[\overline{v}/\overline{x}]\}}$$

$$\frac{\begin{array}{c}o \in [\![I]\!] \qquad \textbf{interface}\ I\ \{\ \cdots\ \tau\ m(\overline{\tau\ x})\ \textbf{req}\ P\ \textbf{level}\ e\ \textbf{ens}\ Q\ \cdots\ \} \in \Gamma \\ P[o/\mathsf{this}, \overline{v}/\overline{x}] \qquad e[o/\mathsf{this}, \overline{v}/\overline{x}] <_{\prec} \ell\end{array}}{\Gamma, \ell \vdash_{\prec} o.m(\overline{v})\ \{Q[o/\mathsf{this}, \overline{v}/\overline{x}]\}}$$

$$\frac{\textbf{class}\ C(\overline{\tau\ f})\ \cdots\ \{\ \overline{\textbf{ghost\_field}\ \tau'\ g = e}\ \cdots\ \} \in \Gamma}{\Gamma, \ell \vdash_{\prec} \textbf{new}\ C(\overline{v})\ \{\mathsf{result} = \textbf{new}\ C(\overline{v}) \triangleleft I(\overline{e}[\overline{v}/\overline{f}])\}}$$

$$\frac{\Gamma, \ell \vdash_{\prec} c\ \{Q\} \qquad \forall v \in [\![\tau]\!]_{\Gamma,\prec}.\ Q[v/\mathsf{result}] \Rightarrow \Gamma, \ell \vdash_{\prec} c'[v/x]\ \{R\}}{\Gamma, \ell \vdash_{\prec} \tau\ x := c; c'\ \{R\}}$$

$$\Gamma, \ell \vdash_{\prec} \textbf{assert}\ \mathsf{true}\ \{\mathsf{true}\}$$

$$\frac{\Gamma, \ell \vdash_{\prec} c\ \{Q'\} \qquad \Gamma \vdash^{\mathsf{rt}}_{\prec} c : \tau \qquad \forall v \in [\![\tau]\!]_{\Gamma,\prec}.\ Q'[v/\mathsf{result}] \Rightarrow Q[v/\mathsf{result}]}{\Gamma, \ell \vdash_{\prec} c\ \{Q\}}$$

$$\frac{\forall o \in [\![C]\!]_{\Gamma,\prec}, \overline{v} \in [\![\overline{\tau}]\!]_{\Gamma,\prec}.\ P[o/\mathsf{this}, \overline{v}/\overline{x}] \Rightarrow \Gamma, e[o/\mathsf{this}, \overline{v}/\overline{x}] \vdash_{\prec} c[o/\mathsf{this}, \overline{v}/\overline{x}]\ \{Q[o/\mathsf{this}, \overline{v}/\overline{x}]\}}{\Gamma \vdash_{C,\prec} \textbf{instance}\ \tau\ m(\overline{\tau\ x})\ \textbf{req}\ P\ \textbf{level}\ e\ \textbf{ens}\ Q\ \{\ c\ \}\ \mathsf{correct}}$$

$$\frac{\forall \overline{v} \in [\![\overline{\tau}]\!]_{\Gamma,\prec}.\ P[\overline{v}/\overline{x}] \Rightarrow \Gamma, e[\overline{v}/\overline{x}] \vdash c[\overline{v}/\overline{x}]\ \{Q[\overline{v}/\overline{x}]\}}{\Gamma \vdash_{C,\prec} \textbf{static}\ \tau\ m(\overline{\tau\ x})\ \textbf{req}\ P\ \textbf{level}\ e\ \textbf{ens}\ Q\ \{\ c\ \}\ \mathsf{correct}}$$

$$\frac{\mathit{class} = \textbf{class}\ C(\cdots)\ \textbf{imports}\ \overline{\mathit{itf}}\ \overline{\mathit{cspec}}\ \cdots}{\prec^{\mathsf{c}}_{\mathit{class}} = \prec_{\overline{\mathit{itf}}\ \overline{\mathit{cspec}}\ \mathit{class}}}$$

$$\frac{\begin{array}{c}\mathit{class} = \textbf{class}\ C(\cdots)\ \textbf{imports}\ \overline{\mathit{itf}}\ \overline{\mathit{cspec}}\ \cdots\ \{\ \cdots\ \overline{\mathit{meth}}\ \} \\ \vdash \mathit{class}\ \mathsf{wt} \qquad \forall \prec.\ \prec^{\mathsf{c}}_{\mathit{class}} \subseteq \prec\ \Rightarrow \overline{\mathit{itf}}\ \overline{\mathit{cspec}}\ \mathit{class} \vdash_{C,\prec} \overline{\mathit{meth}}\ \mathsf{correct}\end{array}}{\vdash \mathit{class}\ \mathsf{correct}}$$

$$\frac{\vdash \mathit{prog}\ \mathsf{wt} \qquad \mathit{prog} = \overline{\mathit{itf}}\ \overline{\mathit{class}} \qquad \vdash \overline{\mathit{class}}\ \mathsf{correct}}{\vdash \mathit{prog}\ \mathsf{correct}}$$

Fig. 40. The proof rules for the correctness judgments. An expression e used as a value should be read as $[\![e]\!]_{\Gamma,\prec}$.

As mentioned in Sec. 2, if a set X is equipped with a well-founded order, this induces a well-founded order on $\mathit{Multisets}(X)$, where $A < B$ iff $A \neq B$ and

$$\exists A', B', C.\ A = A' \uplus C \wedge B = B' \uplus C \wedge \forall x \in A'.\ \exists y \in B'.\ x < y$$

We define the set of multisets of method names as $\mathit{MethodBags} = \mathit{Multisets}(\mathit{MethodNames})$.

A program method precedence order induces a well-founded order on $\mathit{MethodBags}$. However, this order is not total because the program method precedence order is not

$$\begin{array}{rcl}
 & & t \in \mathit{ThreadIds}, \ell \in \mathit{LockIds}, \chi \in \mathit{ChannelIds}, r \in \mathbb{R}, n \in \mathbb{N}, \iota \in \mathit{ObjLabels} \\
v & ::= & () \mid \mathsf{true} \mid \mathsf{false} \mid t \mid \ell \mid \chi \mid r \mid n \mid \textbf{new}\ C(\iota, \overline{v}) \mid \cdots \\
\tau & ::= & \textbf{void} \mid \textbf{bool} \mid \textbf{thread}[\tau] \mid \textbf{lock} \mid \textbf{channel}[\tau] \mid \textbf{waitobj} \mid \textbf{real} \mid \textbf{nat} \mid \textbf{bag}[\tau] \\
 & & \mid C \mid I \mid \cdots \\
e & ::= & x \mid op(\overline{e}) \mid e.\mathit{ff} \mid v \\
a & ::= & e \mid a * a \mid a \wedge a \mid a \vee a \mid \exists x : \tau.\ a \mid e.p(\overline{e}) \mid e.f \overset{e}{\mapsto} e \mid \mathsf{cp}(e) \mid e \cdot a \\
 & & \mid e.\mathsf{thread}(a) \mid e.\mathsf{lock}(e, a) \mid e.\mathsf{locked}(e, a, e) \mid e.\mathsf{channel}(a) \mid e.\mathsf{credit}() \mid e.\mathsf{obs}(e) \\
c & ::= & e \mid C.m(\overline{e}) \mid e.m(\overline{e}) \mid \textbf{new}\ C(\overline{e}) \mid \tau\ x := c; c \mid \textbf{if}\ e\ \textbf{then}\ e\ \textbf{else}\ e \mid \textbf{assert}\ e \\
 & & \mid e.f \mid e.f := e \mid \textbf{new lock}() \mid e.\textbf{acquire}() \mid e.\textbf{release}() \\
 & & \mid \textbf{fork}\ c \mid e.\textbf{join}() \mid \textbf{new channel}[\tau]() \mid e.\textbf{send}(e) \mid e.\textbf{receive}() \\
\mathit{pspec} & ::= & \textbf{predicate}\ p(\overline{\tau\ x}) \\
\beta & ::= & \textbf{static} \mid \textbf{instance} \\
\mathit{mspec} & ::= & \beta\ \tau\ m(\overline{\tau\ x})\ \textbf{forall}\ \overline{\tau\ x}\ \textbf{req}\ a\ \textbf{ens}\ a \\
\mathit{itf} & ::= & \textbf{interface}\ I\ \{\ \overline{\mathit{pspec}}\ \overline{\mathit{mspec}}\ \} \\
\mathit{cspec} & ::= & \textbf{class}\ C\ \{\ \overline{\mathit{mspec}}\ \} \\
\mathit{pred} & ::= & \mathit{pspec} = a \\
\mathit{meth} & ::= & \mathit{mspec}\ \{\ c\ \} \\
\mathit{class} & ::= & \textbf{class}\ C(\overline{\textbf{final}\ \tau\ \mathit{ff}}, \overline{\tau\ f})\ \textbf{imports}\ \overline{\mathit{itf}}\ \overline{\mathit{cspec}}\ \textbf{implements}\ I\ \{\ \overline{\mathit{pred}}\ \overline{\mathit{meth}}\ \} \\
\mathit{prog} & ::= & \overline{\mathit{itf}}\ \overline{\mathit{class}}
\end{array}$$

Fig. 41. Mutation and concurrency: syntax of the formal language

total.[12] Therefore, the $\sqcup$ (max) operator introduced in Sec. 2.2.4 to deal flexibly with sibling objects is not well-defined under this order. Therefore, for every program method precedence order $\prec$, we fix a *linearized program method precedence order* $\mathsf{tot}(\prec)$, chosen arbitrarily among the well-orderings (total well-founded orders) that extend $\prec$. Such a well-ordering exists, by a simple extension of the Well-Ordering Theorem. We then define the order on $\mathit{MethodBags}$ as the order induced by $\mathsf{tot}(\prec)$.

As mentioned above, a machine-checked encoding of this section into the Coq proof assistant is available [Jacobs 2018].

A.2. Mutation and Concurrency (Sec. 3–4)

In this section, we sketch a formalization of the programming language, specification formalism, and notion of module correctness for programs with mutation and concurrency, based on separation logic, the Chalice approach for verifying absence of deadlock, and call permissions, that were gradually introduced in Sec. 3–4, as well as a soundness proof.

A.2.1. Programs. The programming language of this section largely extends that of the preceding one, except that level clauses are replaced by call permissions, and ghost fields are replaced by predicates. The syntax is defined in Fig. 41.

In addition to the sets assumed in the previous section, we assume infinite sets $\mathit{PredNames}$ of predicate names, ranged over by p, and $\mathit{FinalFieldNames}$ of final field names, ranged over by ff. We use the field names (elements $f \in \mathcal{F}$) to denote mutable fields.

[12]If we define $\mathit{MethodNames}_{prog}$ as the names of the methods of program $prog$, then the restriction of $\prec^{\mathsf{p}}_{prog}$ to $\mathit{MethodNames}_{prog}$ is total iff the class import order is total (which it generally is not). However, $\prec^{\mathsf{p}}_{prog}$, which is a relation on $\mathit{MethodNames}$, is never total because it relates two method names $C.m$ and $C.m'$ qualified by the same class name C only if methods with these names appear in the program.

In this section, objects, ranged over by o, are of the form $\mathbf{new}\ C(\iota; \overline{v}) \triangleleft I$, where $\overline{v}$ are the final field values and ι is an element from an infinite set $\mathit{ObjLabels}$ of *object labels* used to distinguish distinct objects that have the same final field values.

We assume disjoint infinite sets $\mathit{ThreadIds}$, $\mathit{LockIds}$, and $\mathit{ChannelIds}$ of thread, lock, and channel identifiers, ranged over by symbols t, ℓ, and χ, respectively. We define the set $\mathit{WaitObjs}$ of waitable objects, ranged over by ω, as $\mathit{WaitObjs} = \mathit{ThreadIds} \cup \mathit{LockIds} \cup \mathit{ChannelIds}$.

The set $\mathbb{V}$ of values, ranged over by v, includes the unit value $()$, the booleans, the reals (used for fractional permissions), the natural numbers (used as the coefficient for a multiplied assertion $n \cdot a$), the objects, and the waitable objects, and it is closed under multiset construction. (That is, $\mathbf{0} \in \mathbb{V}$ and if $v \in \mathbb{V}$, then $\{\!\{v\}\!\} \in \mathbb{V}$, and if multisets $M, M' \in \mathbb{V}$, then $M \uplus M' \in \mathbb{V}$.)

Notice that method specifications feature a **forall** clause. The variables declared in this clause are in scope in the precondition and the postcondition and can be instantiated arbitrarily at each call site. They correspond to logical variables in Hoare logic, and they serve to connect the precondition and the postcondition. In the examples, the **forall** clauses are left implicit: read a method specification **req** P **ens** Q as **forall** $\overline{\tau\ x}$ **req** P **ens** Q, where $\overline{x}$ are the free variables of P and Q (after binding the method parameters and the special variables this and result).

We define well-typedness of assertions by the rules shown in Fig. 42.

A.2.2. Operational Semantics. We define the set $\mathit{PhysRes}$ of *physical resources*, ranged over by α, as follows:

$$\begin{array}{rcl} \alpha & ::= & \mathsf{alloc}(o) \mid o.f \mapsto v \mid \ell.\mathsf{lock}(v) \mid t.\mathsf{thread}(\tilde{c}) \mid \chi.\mathsf{channel}(\overline{v}) \\ \tilde{c} & ::= & c \mid \bot \end{array}$$

We define the set $\mathit{PhysHeaps}$ of *physical heaps*, ranged over by h, as $\mathit{PhysHeaps} = \mathcal{P}_{\text{fin}}(\mathit{PhysRes})$; i.e., a physical heap is a finite set of physical resources. We use physical heaps as machine configurations, and we define a machine step relation $\rightarrow\ \subseteq \mathit{PhysHeaps}^2$ using the rules shown in Fig. 43.

As can be inferred from the step rules, the meaning of the presence of the various resources in a machine configuration is as follows: $\mathsf{alloc}(o)$ means that object o has been allocated; $o.f \mapsto v$ means that field f of object o currently holds value v; $\ell.\mathsf{lock}(v)$ means lock ℓ has been allocated and its current value is v (where value 0 means the lock is currently available and value 1 means it is currently held by some thread); $t.\mathsf{thread}(\tilde{c})$ means that thread t has been allocated, where $\tilde{c} = c$ means that when the thread is scheduled next, it will execute command c, and $\tilde{c} = \bot$, which does not occur in a machine configuration but only temporarily during a machine step, means that thread t has been scheduled and is currently taking a step; and $\chi.\mathsf{channel}(\overline{v})$ means that channel χ has been allocated and the values $\overline{v}$ have been sent on it, in that order, but not yet received.

An *execution* is a finite or infinite sequence of machine configurations, where each subsequent configuration is related to the previous one by a machine step. The purpose of the proof system of this section is to prove absence of two kinds of executions: those that deadlock (i.e., finite executions where the final configuration is not finished, i.e., where some thread's command is not a value), and those that are infinite. We say a machine configuration h is *bad*, denoted $\mathsf{bad}(h)$, if such an execution exists that starts in h. We define bad coinductively by the rules shown in Fig. 43.

A.2.3. Logical resources. To reason modularly in the presence of aliasing and concurrency, separation logic assigns *ownership* of resources to threads and commands. However, using physical resources as the unit of ownership would lead to an ownership

$$\frac{\Gamma \vdash e : \mathbf{bool}}{\Gamma \vdash e : \mathsf{asn}} \qquad \frac{\odot \in \{*, \wedge, \vee\} \qquad \Gamma \vdash a, a' : \mathsf{asn}}{\Gamma \vdash a \odot a' : \mathsf{asn}} \qquad \frac{\Gamma, x : \tau \vdash a : \mathsf{asn}}{\Gamma \vdash \exists x : \tau.\ a : \mathsf{asn}}$$

$$\frac{\Gamma \vdash e : I \qquad \mathbf{interface}\ I\ \{\ \cdots\ \mathbf{predicate}\ p(\overline{\tau\ x})\ \cdots\ \} \in \Gamma \qquad \Gamma \vdash \overline{e} : \overline{\tau}}{\Gamma \vdash e.p(\overline{e}) : \mathsf{asn}}$$

$$\frac{\Gamma \vdash e : C \qquad \mathbf{class}\ C(\cdots\ \tau\ f\ \cdots)\ \cdots \in \Gamma \qquad \Gamma \vdash e' : \mathbf{real} \qquad \Gamma \vdash e'' : \tau}{\Gamma \vdash e.f \overset{e'}{\mapsto} e'' : \mathsf{asn}}$$

$$\frac{\Gamma \vdash e : \tau_{\text{levels}}}{\Gamma \vdash \mathsf{cp}(e) : \mathsf{asn}} \qquad \frac{\Gamma \vdash e : \mathbf{nat} \qquad \Gamma \vdash a : \mathsf{asn}}{\Gamma \vdash e \cdot a : \mathsf{asn}}$$

$$\frac{\Gamma \vdash e : \mathbf{thread}[\tau] \qquad \Gamma, \mathsf{result} : \tau \vdash a : \mathsf{asn}}{\Gamma \vdash e.\mathsf{thread}(a) : \mathsf{asn}} \qquad \frac{\Gamma \vdash e : \mathbf{lock} \qquad \Gamma \vdash e' : \mathbf{real} \qquad \Gamma \vdash a : \mathsf{asn}}{\Gamma e.\mathsf{lock}(e', a) : \mathsf{asn}}$$

$$\frac{\Gamma \vdash e : \mathbf{lock} \qquad \Gamma \vdash e' : \mathbf{real} \qquad \Gamma \vdash a : \mathsf{asn} \qquad \Gamma \vdash e'' : \mathbf{thread}[\tau]}{\Gamma e.\mathsf{locked}(e', a, e'') : \mathsf{asn}}$$

$$\frac{\Gamma \vdash e : \mathbf{channel}[\tau] \qquad \Gamma, \mathsf{element} : \tau \vdash a : \mathsf{asn}}{\Gamma \vdash e.\mathsf{channel}(a) : \mathsf{asn}} \qquad \frac{\Gamma \vdash e : \mathbf{channel}[\tau]}{\Gamma \vdash e.\mathsf{credit}() : \mathsf{asn}}$$

$$\frac{\Gamma \vdash e : \mathbf{thread}[\tau] \qquad \Gamma \vdash e' : \mathbf{bag}[\mathbf{waitobj}]}{\Gamma \vdash e.\mathsf{obs}(e') : \mathsf{asn}}$$

$$\frac{\begin{array}{c}\overline{x}, \overline{x}', \mathsf{result\ distinct} \\ \Gamma \vdash \tau, \overline{\tau}, \overline{\tau'} \qquad \Gamma, \overline{x} : \overline{\tau}, \overline{x'} : \overline{\tau'} \vdash P : \mathsf{asn} \qquad \Gamma, \overline{x} : \overline{\tau}, \overline{x'} : \overline{\tau'}, \mathsf{result} : \tau \vdash Q : \mathsf{asn}\end{array}}{\Gamma \vdash_{\tau_{\mathsf{this}}} \mathbf{static}\ \tau\ m(\overline{\tau\ x})\ \mathbf{forall}\ \overline{\tau'\ x'}\ \mathbf{req}\ P\ \mathbf{ens}\ Q\ \mathsf{wt}}$$

$$\frac{\Gamma \vdash \overline{\tau} \qquad \mathsf{this}, \overline{x}\ \mathsf{distinct} \qquad \Gamma, \mathsf{this} : \tau_{\mathsf{this}}, \overline{x} : \overline{\tau} \vdash a : \mathsf{asn}}{\Gamma \vdash_{\tau_{\mathsf{this}}} \mathbf{predicate}\ p(\overline{\tau\ x}) = a\ \mathsf{wt}}$$

Fig. 42. Well-typedness of assertions

regime that is too coarse. Therefore, the ownership regime used in this section uses *logical resources*, and *fractions* thereof, as the unit of ownership.

We define the set $LogRes$ of logical resources, ranged over by $\hat{\alpha}$, as follows:

$$\begin{array}{rl}\hat{\alpha} ::= & o.f \mapsto v \mid t.\mathsf{thread}(Q) \mid \ell.\mathsf{lock}(I) \mid \ell.\mathsf{locked}(\pi, I, t) \mid \chi.\mathsf{channel}(P) \mid \chi.\mathsf{credit}() \\ & \mid t.\mathsf{obs}(O) \mid \mathsf{cp}(\ell)\end{array}$$

We define the set $LogHeaps$ of logical heaps, ranged over by H, as the fractional multisets of logical resources: $LogHeaps = LogRes \to \mathbb{R}^+$ (where $\mathbb{R}^+ = \{r \in \mathbb{R} \mid 0 \leq r\}$). We define $H \uplus H' = H + H' = \lambda\hat{\alpha}.\ H(\hat{\alpha}) + H'(\hat{\alpha})$.

We use symbol Λ to range over multisets of levels: $\Lambda \in Multisets(\mathbb{L})$. We will identify multisets of levels Λ with *stocks of call permissions*, logical heaps $H = \{\!\{\mathsf{cp}(\ell) \mid \ell \in \Lambda\}\!\}$ containing only call permissions.

$$\frac{\textbf{class}\ C \cdots \{ \cdots\ \textbf{static}\ \tau\ m(\overline{\tau\ x})\ \{\ c\ \}\ \cdots \}}{h, C.m(\overline{v}) \to h, c[\overline{v}/\overline{x}]}$$

$$\frac{o \in [\![C]\!] \qquad \textbf{class}\ C \cdots \{ \cdots\ \textbf{instance}\ \tau\ m(\overline{\tau\ x})\ \{\ c\ \}\ \cdots \}}{h, o.m(\overline{v}) \to h, c[o/\mathsf{this}, \overline{v}/\overline{x}]}$$

$$\frac{\textbf{class}\ C(\overline{\textbf{final}\ \tau\ \mathit{ff}}, \overline{\tau'\ f})\ \cdots\ \textbf{implements}\ I\ \cdots \qquad o = \textbf{new}\ C(\iota; \overline{v}) \triangleleft I \qquad \mathsf{alloc}(o) \notin h}{h, \textbf{new}\ C(\overline{v}, \overline{v}') \to h \uplus \{\!\!\{\mathsf{alloc}(o), \overline{o.f \mapsto v'}\}\!\!\}, o}$$

$$\frac{h, c \to h', c'}{h, \tau\ x := c; c'' \to h', \tau\ x := c'; c''} \qquad h, \tau\ x := v; c \to h, c[v/x] \qquad h, \textbf{assert}\ \mathsf{true} \to h, ()$$

$$h \uplus \{\!\!\{o.f \mapsto v\}\!\!\}, o.f \to h \uplus \{\!\!\{o.f \mapsto v\}\!\!\}, v \qquad h \uplus \{\!\!\{o.f \mapsto v\}\!\!\}, o.f := v' \to h \uplus \{\!\!\{o.f \mapsto v'\}\!\!\}, ()$$

$$\frac{\ell \notin \{\ell \mid \ell.\mathsf{lock}(\_) \in h\}}{h, \textbf{new lock}() \to h \uplus \{\!\!\{\ell.\mathsf{lock}(0)\}\!\!\}, \ell} \qquad h \uplus \{\!\!\{\ell.\mathsf{lock}(0)\}\!\!\}, \ell.\textbf{acquire}() \to h \uplus \{\!\!\{\ell.\mathsf{lock}(1)\}\!\!\}, ()$$

$$h \uplus \{\!\!\{\ell.\mathsf{lock}(1)\}\!\!\}, \ell.\textbf{release}() \to h \uplus \{\!\!\{\ell.\mathsf{lock}(0)\}\!\!\}, () \qquad \frac{t \notin \{t \mid t.\mathsf{thread}(\_) \in h\}}{h, \textbf{fork}\ c \to h \uplus \{\!\!\{t.\mathsf{thread}(c)\}\!\!\}, t}$$

$$h \uplus \{\!\!\{t.\mathsf{thread}(v)\}\!\!\}, t.\textbf{join}() \to h \uplus \{\!\!\{t.\mathsf{thread}(v)\}\!\!\}, v$$

$$\frac{\chi \notin \{\chi \mid \chi.\mathsf{channel}(\_) \in h\}}{h, \textbf{new channel}[\tau]() \to h \uplus \{\!\!\{\chi.\mathsf{channel}(\epsilon)\}\!\!\}, \chi}$$

$$h \uplus \{\!\!\{\chi.\mathsf{channel}(\overline{v})\}\!\!\}, \chi.\textbf{send}(v) \to h \uplus \{\!\!\{\chi.\mathsf{channel}(\overline{v}\,v)\}\!\!\}, ()$$

$$h \uplus \{\!\!\{\chi.\mathsf{channel}(v\,\overline{v})\}\!\!\}, \chi.\textbf{receive}() \to h \uplus \{\!\!\{\chi.\mathsf{channel}(\overline{v})\}\!\!\}, v$$

$$\frac{h \uplus \{\!\!\{t.\mathsf{thread}(\bot)\}\!\!\}, c \to h' \uplus \{\!\!\{t.\mathsf{thread}(\bot)\}\!\!\}, c'}{h \uplus \{\!\!\{t.\mathsf{thread}(c)\}\!\!\} \to h' \uplus \{\!\!\{t.\mathsf{thread}(c')\}\!\!\}} \qquad \mathsf{finished}(h) = \forall t.\mathsf{thread}(c) \in h.\ c \in \mathbb{V}$$

$$\frac{h \not\to \qquad \neg\mathsf{finished}(h)}{\mathsf{bad}\,h} \qquad \frac{h \to h' \qquad \mathsf{bad}\,h'}{\mathsf{bad}\,h}$$

Fig. 43. Mutation and concurrency: program execution

A.2.4. Assertions. In this section, method preconditions and postconditions are *assertions* rather than just boolean expressions. In addition to asserting the truth of boolean expressions, assertions may assert the ownership of (fractions of) logical resources.

Assertions may refer to predicates. To interpret an assertion, an interpretation for the predicates it refers to is needed. We define the set $\mathit{PredInterps}$ of *predicate interpretations*, ranged over by I, as the set of all sets of tuples of the form $(H, o, p, \overline{v})$, where H is a logical heap, and $o.p(\overline{v})$ is a predicate application that, according to the interpretation, is satisfied by H.

We define *satisfaction* of a closed assertion a by a logical heap H under a predicate interpretation I, denoted $I, H \vDash a$, inductively by the rules shown in Fig. 44.

$$\begin{array}{lcl}
I, H \vDash b & \Leftarrow & b = \mathsf{true} \\
I, H \vDash P * Q & \Leftarrow & \exists H_1, H_2.\ H = H_1 \uplus H_2 \wedge I, H_1 \vDash P \wedge I, H_2 \vDash Q \\
I, H \vDash P \odot Q & \Leftarrow & I, H \vDash P \odot I, H \vDash Q \quad \textbf{where } \odot \in \{\wedge, \vee\} \\
I, H \vDash \exists x : \tau.\ a & \Leftarrow & \exists v \in [\![\tau]\!].\ I, H \vDash a[v/x] \\
I, H \vDash o.p(\overline{v}) & \Leftarrow & (H, o, p, \overline{v}) \in I \\
I, H \vDash o.f \overset{r}{\mapsto} v & \Leftarrow & 0 < r \wedge H(o.f \mapsto v) \geq r \\
I, H \vDash \mathsf{cp}(\ell) & \Leftarrow & H(\mathsf{cp}(\ell)) \geq 1 \\
I, H \vDash 0 \cdot a & \Leftarrow & \mathsf{true} \\
I, H \vDash (n+1) \cdot a & \Leftarrow & I, H \vDash a * n \cdot a \\
I, H \vDash t.\mathsf{thread}(Q) & \Leftarrow & H(t.\mathsf{thread}(Q)) \geq 1 \\
I, H \vDash \ell.\mathsf{lock}(r, I) & \Leftarrow & 0 < r \wedge H(\ell.\mathsf{lock}(I)) \geq r \\
I, H \vDash \ell.\mathsf{locked}(r, I, t) & \Leftarrow & H(\ell.\mathsf{locked}(r, I, t)) \geq 1 \\
I, H \vDash \chi.\mathsf{channel}(P) & \Leftarrow & H(\chi.\mathsf{channel}(P)) > 0 \\
I, H \vDash \chi.\mathsf{credit}() & \Leftarrow & H(\chi.\mathsf{credit}()) \geq 1 \\
I, H \vDash t.\mathsf{obs}(O) & \Leftarrow & H(t.\mathsf{obs}(O)) \geq 1
\end{array}$$

Fig. 44. Assertion satisfaction

$$\frac{\forall I, H.\ I, H \vDash P \Rightarrow I, H \vDash P'}{P \sqsubseteq P'}$$

$$\frac{o \in [\![C]\!] \qquad \textbf{class } C \cdots \{ \cdots \textbf{ predicate } p(\overline{\tau\ x}) = a\ \cdots \} \in \Gamma}{o.p(\overline{v}) \sqsubseteq\sqsupseteq a[o/\mathsf{this}, \overline{v}/\overline{x}]}$$

$$\frac{\forall i.\ \ell_i < \ell}{\mathsf{cp}(\ell) \sqsubseteq \mathsf{cp}(\ell_1) * \mathsf{cp}(\ell_2) * \cdots * \mathsf{cp}(\ell_n)} \qquad \ell.\mathsf{lock}(1, I) \sqsubseteq I$$

$$t.\mathsf{obs}(O) * \chi.\mathsf{channel}(P) \sqsubseteq\sqsupseteq t.\mathsf{obs}(O \uplus \{\!\!\{\chi\}\!\!\}) * \chi.\mathsf{channel}(P) * \chi.\mathsf{credit}() \qquad \frac{P \sqsubseteq P'}{P * R \sqsubseteq P' * R}$$

$$\frac{P \sqsubseteq P' \qquad P' \sqsubseteq P''}{P \sqsubseteq P''}$$

Fig. 45. Assertion weakening. Read each judgment $P \sqsubseteq P'$ as $\Gamma \vdash P \sqsubseteq P'$.

Notice that, as is common in separation logics for garbage-collected programming languages, our definition of assertion satisfaction allows leaking: if $I, H \vDash a$ then $I, H \uplus H' \vDash a$ (provided that for all H and H', if $(H, o, p, \overline{v}) \in I$ then $(H \uplus H', o, p, \overline{v}) \in I$).

A.2.5. Proof rules. We define the assertion weakening relation $\Gamma \vdash P \sqsubseteq P'$ inductively by the rules shown in Fig. 45.

We define the command correctness judgment $t \vdash \{P\}\ c\ \{Q\}$, where t is a thread identifier, P is a closed assertion, and Q is an assertion whose only free variable is result, inductively by the rules of Figs. 21 and 46.

A.2.6. Soundness proof. We fix a program *prog*, and we assume that it is correct.

CNew
$$\frac{\textbf{class}\ C(\overline{\textbf{final}\ \tau\ \mathit{ff}}, \overline{\tau'\ f})\ \cdots \in \Gamma}{\{\mathsf{true}\}\ \textbf{new}\ C(\overline{v}, \overline{v'})\ \{\overline{\mathsf{result}.\mathit{ff} = v} \wedge \overline{\mathsf{result}.f \mapsto v'}\}}$$

CLookup
$$\{o.f \overset{\pi}{\mapsto} v\}\ o.f\ \{o.f \overset{\pi}{\mapsto} v \wedge \mathsf{result} = v\}$$

CMutate
$$\{o.f \mapsto \_\}\ o.f := v\ \{o.f \mapsto v\}$$

CLet
$$\frac{\{P\}\ c\ \{Q\} \qquad \forall v \in [\![\tau]\!].\ \{Q[v/\mathsf{result}]\}\ c'[v/x]\ \{R\}}{\{P\}\ \tau\ x := c; c'\ \{R\}}$$

CFrame
$$\frac{\{P\}\ c\ \{Q\}}{\{P * R\}\ c\ \{Q * R\}}$$

CConseq
$$\frac{\Gamma \vdash P \sqsubseteq P' \qquad \{P'\}\ c\ \{Q\} \qquad \Gamma \vdash Q \sqsubseteq Q'}{\{P\}\ c\ \{Q'\}}$$

CExists
$$\frac{\forall v \in [\![\tau]\!].\ \{P[v/x]\}\ c\ \{Q\}}{\{\exists x : \tau.\ P\}\ c\ \{Q\}}$$

CStaticCall
$$\frac{\textbf{class}\ C \cdots\ \{\ \cdots\ \textbf{static}\ \tau''\ m(\overline{\tau\ x})\ \textbf{forall}\ \overline{\tau'\ x'}\ \textbf{req}\ P\ \textbf{ens}\ Q\ \cdots\ \} \in \Gamma \qquad \overline{v'} \in [\![\overline{\tau'}]\!]}{\{P[\overline{v}/\overline{x}, \overline{v'}/\overline{x'}] * \mathsf{cp}(\_)\}\ C.m(\overline{v})\ \{Q[\overline{v}/\overline{x}, \overline{v'}/\overline{x'}]\}}$$

CInstanceCall
$$\frac{\begin{matrix} o : I \\ \textbf{interface}\ I \cdots\ \{\ \cdots\ \textbf{instance}\ \tau''\ m(\overline{\tau\ x})\ \textbf{forall}\ \overline{\tau'\ x'}\ \textbf{req}\ P\ \textbf{ens}\ Q\ \cdots\ \} \in \Gamma \qquad \overline{v'} \in [\![\overline{\tau'}]\!] \end{matrix}}{\{P[o/\mathsf{this}, \overline{v}/\overline{x}, \overline{v'}/\overline{x'}] * \mathsf{cp}(\_)\}\ o.m(\overline{v})\ \{Q[o/\mathsf{this}, \overline{v}/\overline{x}, \overline{v'}/\overline{x'}]\}}$$

CStaticMethod
$$\frac{\forall t, \overline{v} \in [\![\overline{\tau}]\!], \overline{v'} \in [\![\overline{\tau'}]\!].\ t \vdash \{P[\overline{v}/\overline{x}, \overline{v'}/\overline{x'}]\}\ c[\overline{v}/\overline{x}]\ \{Q[\overline{v}/\overline{x}, \overline{v'}/\overline{x'}]\}}{\Gamma \vdash_C \textbf{static}\ \tau''\ m(\overline{\tau\ x})\ \textbf{forall}\ \overline{\tau'\ x'}\ \textbf{req}\ P\ \textbf{ens}\ Q\ \{\ c\ \}\ \mathsf{correct}}$$

CInstanceMethod
$$\frac{\forall t, o \in [\![C]\!], \overline{v} \in [\![\overline{\tau}]\!], \overline{v'} \in [\![\overline{\tau'}]\!].\ t \vdash \{P[o/\mathsf{this}, \overline{v}/\overline{x}, \overline{v'}/\overline{x'}]\}\ c[o/\mathsf{this}, \overline{v}/\overline{x}]\ \{Q[o/\mathsf{this}, \overline{v}/\overline{x}, \overline{v'}/\overline{x'}]\}}{\Gamma \vdash_C \textbf{instance}\ \tau''\ m(\overline{\tau\ x})\ \textbf{forall}\ \overline{\tau'\ x'}\ \textbf{req}\ P\ \textbf{ens}\ Q\ \{\ c\ \}\ \mathsf{correct}}$$

Fig. 46. Proof rules. The rules from Fig. 21 are not repeated. Read each judgment $\{P\}\ c\ \{Q\}$ as $\Gamma, t \vdash \{P\}\ c\ \{Q\}$.

We fix the *program predicate interpretation* I_{prog} as $I_{prog} = \bigcap\{I \mid F(I) \subseteq I\}$ where for all I, the set $F(I)$ is defined by the following rule:

$$\frac{o \in [\![C]\!] \qquad \textbf{class}\ C \cdots\ \{\ \cdots\ \textbf{predicate}\ p(\overline{\tau\ x}) = a\ \cdots\ \} \in \mathit{prog} \qquad I, H \vDash a[o/\mathsf{this}, \overline{v}/\overline{x}]}{(H, o, p, \overline{v}) \in F(I)}$$

Lemma A.2. *Assertion satisfaction is monotonic:* $I \subseteq I' \wedge I, H \vDash a \Rightarrow I', H \vDash a$.

Proof. By induction on the structure of a. □

Lemma A.3. *F is monotonic:* $I \subseteq I' \Rightarrow F(I) \subseteq F(I')$.

Lemma A.4. *I_{prog} is a fixpoint of F:* $F(I_{prog}) = I_{prog}$.

Proof. By the Knaster-Tarski theorem. □

We define $H \vDash a$ by the rule $H \vDash a \Leftrightarrow I_{prog}, H \vDash a$.

$$
\begin{array}{l}
\mathsf{safe}(H, \mathsf{weakenCallPerms}, Q) = \exists H', \Lambda, \Lambda' \leq \Lambda.\ H = H' \uplus \Lambda \wedge Q(H' \uplus \Lambda') \\
\mathsf{safe}(H, \mathsf{destroyLock}, Q) = \exists H', \ell, I.\ H = H' \uplus \{\!\{\ell.\mathsf{lock}(1, I)\}\!\} \wedge \forall H''.\ H'' \vDash I \Rightarrow Q(H' \uplus H'') \\
\mathsf{safe}(H, \mathsf{createChannelDebit}, Q) = \\
\quad \exists H', t, \chi, O, \pi, P.\ H = H' \uplus \{\!\{t.\mathsf{obs}(O), \pi \cdot \chi.\mathsf{channel}(P)\}\!\} \\
\qquad \wedge\ Q(H' \uplus \{\!\{t.\mathsf{obs}(O \uplus \{\!\{\chi\}\!\}), \pi \cdot \chi.\mathsf{channel}(P), \chi.\mathsf{credit}()\}\!\}) \\
\mathsf{safe}(H, \mathsf{destroyChannelDebit}, Q) = \\
\quad \exists H', t, \chi, O, \pi, P.\ H = H' \uplus \{\!\{t.\mathsf{obs}(O \uplus \{\!\{\chi\}\!\}), \pi \cdot \chi.\mathsf{channel}(P), \chi.\mathsf{credit}()\}\!\} \\
\qquad \wedge\ Q(H' \uplus \{\!\{t.\mathsf{obs}(O), \pi \cdot \chi.\mathsf{channel}(P)\}\!\}) \\
\mathsf{safe}(H, \epsilon, Q) = Q(H) \\
\mathsf{safe}(H, \hat{c} \cdot \overline{\hat{c}}, Q) = \mathsf{safe}(H, \hat{c}, \lambda H'.\ \mathsf{safe}(H', \overline{\hat{c}}, Q))
\end{array}
$$

Fig. 47. Safety of a ghost command and of a list of ghost commands

We define the judgment h, c waitingFor ω, expressing that in machine configuration h, command c is blocked on waitable object ω, by the following rules:

$$
h \uplus \{\!\{\ell.\mathsf{lock}(1)\}\!\}, \ell.\mathbf{acquire}()\ \mathsf{waitingFor}\ \ell \qquad h \uplus \{\!\{\chi.\mathsf{channel}(\epsilon)\}\!\}, \chi.\mathbf{receive}()\ \mathsf{waitingFor}\ \chi
$$

$$
\frac{c \notin \mathbb{V}}{h \uplus \{\!\{t.\mathsf{thread}(c)\}\!\}, t.\mathbf{join}()\ \mathsf{waitingFor}\ t} \qquad \frac{h, c\ \mathsf{waitingFor}\ \omega}{h, \tau\ x := c; c'\ \mathsf{waitingFor}\ \omega}
$$

We define a set $\mathit{GhostCmds}$ of *ghost commands*, ranged over by $\hat{c}$, as follows:

$$
\hat{c} ::= \mathsf{weakenCallPerms} \mid \mathsf{destroyLock} \mid \mathsf{createChannelDebit} \mid \mathsf{destroyChannelDebit}
$$

We define safety of a ghost command and of a list of ghost commands by the rules shown in Fig. 47.

LEMMA A.5. *If* $P \sqsubseteq P'$ *and* $H \vDash P$, *then* $\exists \overline{\hat{c}}.\ \mathsf{safe}(H, \overline{\hat{c}}, \lambda H'.\ H' \vDash P')$.

PROOF. By induction on the derivation of the first premise. □

To relate the physical resources in a machine configuration to the logical resources owned by the threads, the locks, and the channels, we introduce *thread tables*, *lock tables*, and *channel tables*. These are finite sets of tuples, where each tuple describes both the physical and the logical aspects of one thread, lock, or channel, respectively.

Specifically, a thread table T is a finite set of tuples of the form $(t, j, O, \tilde{H}, \overline{\hat{c}}, \tilde{c}, Q)$, where t is a thread identifier, j is the thread's joinability (either joinable or nonJoinable), O is a multiset of waitable objects expressing the thread's obligations, $\tilde{H}$ is either the logical heap owned by the thread, or else $\bot$, indicating that the thread has been joined, $\overline{\hat{c}}$ is the list of ghost commands to be executed by the thread before it makes a machine step, $\tilde{c}$ is the thread's command, and Q is the thread's postcondition (an assertion whose only free variable is result). A lock table L is a finite set of tuples of the form (ℓ, v, π, I, t, H), where ℓ is a lock identifier, v is the lock's current value, π (a real number) is the fraction of the $\ell.\mathsf{lock}(I)$ logical resource consumed by the most recent $\ell.\mathbf{acquire}()$ operation, or 0 if the lock is currently available, I is the lock invariant (a closed assertion), t is the identifier of the thread currently holding the lock (if any), and H is the logical resources owned by the lock (if the lock is currently available). A channel table C is a finite set of tuples of the form $(\chi, \overline{v}, P, H)$, where χ is a channel identifier, $\overline{v}$ is the channel's contents, P is the channel element resources predicate (an assertion with one free variable, element, that describes the resources owned by the channel for each of its elements), and H is the resources owned by the channel.

$$
\begin{array}{l}
h, T, \Lambda\ \mathsf{ok} \Leftrightarrow \\
\quad \exists A, L, C. \\
\qquad (\forall (\ell, v, \pi, I, t, H) \in L. \\
\qquad\quad 0 \leq \pi \leq 1 \land (\pi = 0 \Leftrightarrow v = 0) \land (v = 0 \Rightarrow H \vDash I) \land \Sigma_{(\_,\_.O,\_,\_,\_,\_) \in T}\ O(\ell) = v) \\
\qquad \land\ (\forall (t, \mathsf{joinable}, O, H, \bar{\tilde{c}}, \tilde{c}, Q) \in T.\ \tilde{c} \notin \mathbb{V} \Rightarrow \Sigma_{(\_,\_.O,\_,\_,\_,\_) \in T}\ O(t) = 1) \\
\qquad \land\ (\forall (\chi, \overline{v}, P, H) \in C.\ H \vDash \circledast_{v \in \overline{v}} P[v/\mathsf{element}]) \\
\qquad \land\ h = \{\!\{\mathsf{alloc}(o), \overline{o.f \mapsto v} \mid (o, \overline{(f, v)}) \in A\}\!\} \\
\qquad\qquad\quad \uplus\ \{\!\{\ell.\mathsf{lock}(v) \mid (\ell, v, \pi, I, t, H) \in L\}\!\} \\
\qquad\qquad\quad \uplus\ \{\!\{t.\mathsf{thread}(\tilde{c}) \mid (t, \_, O, \tilde{H}, \bar{\tilde{c}}, \tilde{c}, Q) \in T\}\!\} \\
\qquad\qquad\quad \uplus\ \{\!\{\chi.\mathsf{channel}(\overline{v}) \mid (\chi, \overline{v}, P, H) \in C\}\!\} \\
\qquad \land\ (\biguplus_{(\ell,0,\pi,I,t,H) \in L} H) \uplus (\biguplus_{(t,\_,O,H,\bar{\tilde{c}},\tilde{c},Q) \in T} H) \uplus (\biguplus_{(\chi,\overline{v},P,H) \in C} H) \leq \\
\qquad\quad \{\!\{\overline{o.f \mapsto v} \mid (o, \overline{(f, v)}) \in A\}\!\} \\
\qquad\quad \uplus\ \{\!\{(1 - \pi) \cdot \ell.\mathsf{lock}(I) \mid (\ell, v, \pi, I, t, H) \in L, \pi < 1\}\!\} \\
\qquad\quad \uplus\ \{\!\{\ell.\mathsf{locked}(\pi, I, t) \mid (\ell, 1, \pi, I, t, H) \in L\}\!\} \\
\qquad\quad \uplus\ \{\!\{t.\mathsf{obs}(O) \mid (t, \_, O, H, \bar{\tilde{c}}, \tilde{c}, Q) \in T\}\!\} \\
\qquad\quad \uplus\ \{\!\{t.\mathsf{thread}(Q) \mid (t, \mathsf{joinable}, O, H, \bar{\tilde{c}}, \tilde{c}, Q) \in T\}\!\} \\
\qquad\quad \uplus\ \{\!\{\chi.\mathsf{channel}(P), n \cdot \chi.\mathsf{credit}() \mid (\chi, \overline{v}, P, H) \in C \land n = |\overline{v}| + \Sigma_{(\_,\_.O,\_,\_,\_,\_) \in T} O(\chi)\}\!\} \\
\qquad\quad \uplus\ \{\!\{\mathsf{cp}(\ell) \mid \ell \in \Lambda\}\!\}
\end{array}
$$

Fig. 48. Consistency of a machine configuration, a thread table, and a stock of call permissions

Furthermore, we introduce the notion of an *object allocation table*. This is simply a set of tuples of the form $(o, \overline{(f, v)})$, where o is an allocated object and $\overline{(f, v)}$ are the names and values of its mutable fields.

We define the judgment $h, T, \Lambda\ \mathsf{ok}$ that denotes *consistency* of physical heap h, thread table T, and stock of call permissions Λ, in Fig. 48. This is the case if an object allocation table A, a lock table L, and a channel table C exist such that the following conditions hold:

— For each lock ℓ, the fraction π of the $\ell.\mathsf{lock}(I)$ chunk consumed by the most recent $\ell.$**acquire**$()$ operation is between 0 and 1, and is zero only if the lock is currently available. Furthermore, if the lock is available, the logical resources H owned by the lock satisfy the lock invariant I. Also, the total number of obligations for ℓ held by all threads equals 0 if the lock is available, and 1 if the lock is held.
— If a joinable thread is not finished (i.e., its command is not a value), the total number of obligations for t held by all threads equals 1.
— A channel's bundle of owned logical resources H satisfies the separating conjunction of its channel element resources predicate instantiated for each element in the channel's queue.
— The physical heap consists of the physical chunks corresponding to the elements of the object allocation table, the lock table, the thread table, and the channel table.
— The sum of the logical resources owned by the locks that are available, the threads that have not been joined, and the channels, is at most the sum of the logical resources corresponding to the elements of the allocated object table, the lock table, the thread table, the channel table, and the stock of call permissions. Notice that the total number of credits for a channel equals the number of elements in the queue plus the total number of obligations for the channel held by all threads.

We define the *size* $|c|$ of a command c and $|T|$ of a thread table T as follows:

$$|\tau\ x := c; c'| = |c| + 1 + |c'| \quad |\mathbf{fork}\ c| = 1 + |c| \quad |\_| = 1 \quad |T| = \Sigma_{(t,\_,O,H,c,Q) \in T}\ |c|$$

$$\begin{aligned}
&\mathsf{safe}_{\Lambda,n}(t, H, \overline{\hat{c}}, c, Q) = \\
&\quad \mathsf{safe}(t, H, \overline{\hat{c}}, \lambda H. \\
&\quad (\forall v.\ c = v \Rightarrow H \vDash Q[v/\mathsf{result}]) \land \\
&\quad (\forall h, T, j, O, k, \Lambda_0, H_{\mathsf{F}}, Q_0. \\
&\qquad (\Lambda_0, |T| + |c| + k) \leq (\Lambda, n) \land h, T \uplus \{\!\!\{(t, j, O, H \uplus H_{\mathsf{F}}, \epsilon, \bot, Q_0)\}\!\!\}, \Lambda_0\ \mathsf{ok} \land \mathsf{safe}_{<(\Lambda,n)}(T) \Rightarrow \\
&\qquad (\forall h', c'.\ h, c \to h', c' \Rightarrow \\
&\qquad\quad \exists O', H', \overline{\hat{c}}', T', \Lambda'. \\
&\qquad\qquad (\Lambda', |T'| + |c'| + k) < (\Lambda_0, |T| + |c| + k) \\
&\qquad\qquad \land\ h', T' \uplus \{\!\!\{(t, j, O', H' \uplus H_{\mathsf{F}}, \epsilon, \bot, Q_0)\}\!\!\}, \Lambda'\ \mathsf{ok} \\
&\qquad\qquad \land\ \mathsf{safe}_{\Lambda', |T'|+|c'|+k}(t, H', \overline{\hat{c}}', c', Q) \land \mathsf{safe}_{\Lambda', |T'|+|c'|+k}(T')) \\
&\qquad \land\ (\forall \omega.\ h, c\ \mathsf{waitingFor}\ \omega \Rightarrow \mathsf{w}(\omega) \prec O \land \exists (t', \_, O', \_, \_, \_, \_) \in T.\ \omega \in O')))
\end{aligned}$$

$$\begin{aligned}
&\mathsf{safe}_{\Lambda,n}(T) = \\
&\quad (\forall (t, \mathsf{joinable}, O, H, \overline{\hat{c}}, c, Q) \in T.\ \mathsf{safe}_{\Lambda,n}(t, H, \overline{\hat{c}}, c, t.\mathsf{obs}(\{\!\!\{t\}\!\!\}) * Q)) \\
&\quad \land\ (\forall (t, \mathsf{nonJoinable}, O, H, \overline{\hat{c}}, c, Q) \in T.\ \mathsf{safe}_{\Lambda,n}(t, H, \overline{\hat{c}}, c, t.\mathsf{obs}(\mathbf{0})))
\end{aligned}$$

$$\mathsf{safe}_{<(\Lambda,n)}(T) = \forall (\Lambda', n') < (\Lambda, n).\ \mathsf{safe}_{\Lambda',n'}(T)$$

Fig. 49. Safety of a command and a thread table

The size of a thread table is an upper bound on the number of steps the machine can take before it performs a method call. During the execution of a correct program, the pair (Λ, n) of the available stock of call permissions Λ and the size n of the thread table, which we call the *rank* of the instrumented machine configuration, ordered lexicographically with Λ more significant, forms a descending chain.

To define the meaning of correctness judgments, we adopt the approach of Vafeiadis [2011] and extend it for total correctness and for unstructured thread forking. Specifically, we define an auxiliary judgment $\mathsf{safe}_{\Lambda,n}(t, H, \overline{\hat{c}}, c, Q)$, which denotes that execution of ghost commands $\overline{\hat{c}}$ followed by command c by thread t under ownership of logical resources H will terminate successfully with a resulting bundle of logical resources that satisfies postcondition Q, when started in an instrumented machine configuration whose rank is at most (Λ, n).

Definition A.6. We define safety $\mathsf{safe}_{\Lambda,n}(t, H, \overline{\hat{c}}, c, Q)$ of a command and $\mathsf{safe}_{\Lambda,n}(T)$ of a thread table by well-founded recursion on (Λ, n) in Fig. 49.

Specifically, $\mathsf{safe}_{\Lambda,n}(t, H, \overline{\hat{c}}, c, Q)$ expresses that the ghost commands $\overline{\hat{c}}$ execute safely under a postcondition that consists of two conjuncts. The first conjunct states that if the command is a value, then the owned logical resources H satisfy the postcondition, instantiated with the result value. The second conjunct quantifies over all possible circumstances under which the command c could be executed. Specifically, it quantifies over the machine configuration h, the thread table T (not including the thread t executing c), t's joinability j and obligations O, the size k of the context of c in t's command (i.e., the size of t's command minus the size of c), the current stock of call permissions Λ_0, the logical resources H_{F} owned by t minus those owned by c, and t's postcondition Q_0. It assumes that the instrumented machine configuration's rank does not exceed the bound, that the machine configuration, the completed thread table, and the stock of call permissions are consistent, and that the threads (not including t) are safe under all lesser rank bounds.

It then states two conjuncts. The first conjunct states that for any step that c can take in this machine configuration, an instrumentation exists for the resulting machine configuration such that it has a lesser rank, it is consistent, and both thread t and the other threads in the new thread table are safe to execute with the new rank as a bound. The second conjunct states that if in this machine configuration c is blocked on a waitable object ω, then ω's waiting level is below thread t's obligations and ω is among some other thread's obligations.

A thread table T is safe under a rank bound (Λ, n), denoted $\mathsf{safe}_{\Lambda,n}(T)$, if all threads that have not been joined are safe to execute under the specified rank bound and under a postcondition that depends on whether the thread is joinable.

LEMMA A.7.

$$\forall \Lambda, t, P, c, Q.\ t \vdash \{P\}\ c\ \{Q\} \Rightarrow \forall n, H.\ H \vDash P \Rightarrow \exists \bar{\hat{c}}.\ \mathsf{safe}_{\Lambda,n}(t, H, \bar{\hat{c}}, c, Q)$$

PROOF. By well-founded induction on Λ. Assume the induction hypothesis:

$$\forall \Lambda' < \Lambda.\ \forall t, P, c, Q.\ t \vdash \{P\}\ c\ \{Q\} \Rightarrow \forall n, H.\ H \vDash P \Rightarrow \exists \bar{\hat{c}}.\ \mathsf{safe}_{\Lambda',n}(t, H, \bar{\hat{c}}, c, Q) \quad (1)$$

By nested induction on the derivation of the correctness judgment. We detail a few cases:

— **Case** CSTATICCALL. We have $H \vDash P' * \mathsf{cp}(\_)$, where P' is the method's precondition, appropriately instantiated. It follows that there exists some level ℓ and logical heap H' such that $H = H' \uplus \{\!\![\mathsf{cp}(\ell)]\!\!\}$ and $H' \vDash P'$. Fix a consistent pre-step instrumented machine configuration. From consistency, it follows that there exists a Λ' such that $\Lambda = \Lambda' \uplus \{\!\![\ell]\!\!\}$. It follows that the post-step instrumented machine configuration is consistent with respect to Λ' as the stock of call permissions. Let c' be the body of the method, appropriately instantiated. By correctness of the program, we have $\{P'\}\ c'\ \{Q\}$. By (1) instantiated for Λ', we have safety of c'.

— **Case** CLET. Assume $c = \tau\ x := c_1; c_2$. Assume $\{P\}\ c_1\ \{Q_1\}$. Assume the induction hypotheses:

$$\forall n, H.\ H \vDash P \Rightarrow \exists \bar{\hat{c}}.\ \mathsf{safe}_{\Lambda,n}(t, H, \bar{\hat{c}}, c_1, Q_1) \quad (2)$$

$$\forall v \in [\![\tau]\!], n, H.\ H \vDash Q_1[v/\mathsf{result}] \Rightarrow \exists \bar{\hat{c}}.\ \mathsf{safe}_{\Lambda,n}(t, H, \bar{\hat{c}}, c_2[v/\mathsf{result}], Q) \quad (3)$$

Fix H, n. Assume $H \vDash P$. By (2), ghost commands $\bar{\hat{c}}$ exist such that $\mathsf{safe}_{\Lambda,n}(t, H, \bar{\hat{c}}, c_1, Q_1)$. We prove the following lemma:

LEMMA A.8.

$$\forall \Lambda' \leq \Lambda, n, H, \bar{\hat{c}}, c.\ \mathsf{safe}_{\Lambda',n}(t, H, \bar{\hat{c}}, c, Q_1) \Rightarrow \exists \bar{\hat{c}}.\ \mathsf{safe}_{\Lambda',n}(t, H, \bar{\hat{c}}, \tau\ x := c; c_2, Q)$$

PROOF. By well-founded induction on $(\Lambda', n, |\bar{\hat{c}}|)$. Assume the induction hypothesis:

$$\forall \Lambda'', n', \bar{\hat{c}}'.\ (\Lambda'', n', |\bar{\hat{c}}'|) < (\Lambda', n, |\bar{\hat{c}}|) \Rightarrow$$
$$\forall H, c.\ \mathsf{safe}_{\Lambda'',n'}(t, H, \bar{\hat{c}}, c, Q_1) \Rightarrow \exists \bar{\hat{c}}.\ \mathsf{safe}_{\Lambda'',n'}(t, H, \bar{\hat{c}}, \tau\ x := c; c_2, Q) \quad (4)$$

If $|\bar{\hat{c}}| > 0$, the goal follows easily by case analysis on the first ghost command and (4). Now assume $|\bar{\hat{c}}| = 0$. If c is a value v, we have $H \vDash Q_1[v/\mathsf{result}]$. The goal then follows from (3). Otherwise, assume some physical heap h and $h, \tau\ x := c; c_2 \rightarrow h', c'$ for some h', c'. Then c' is necessarily of the form $\tau\ x := c''; c_2$ and $h, c \rightarrow h', c''$. The goal then follows by the safety of c and (4). □

The goal follows immediately from the lemma and (2). □

We define $||T||$ as the total number of ghost commands in T: $||T|| = \Sigma_{(\_,\_,\_,\_,\bar{\hat{c}},\_,\_) \in T} |\bar{\hat{c}}|$.

LEMMA A.9.

$$\forall \Lambda, n, m, h, T.\ n = |T| \wedge m = ||T|| \wedge h, T, \Lambda\ \mathsf{ok} \wedge \mathsf{safe}_{\Lambda,n}(T) \Rightarrow \neg\mathsf{bad}(h)$$

PROOF. By well-founded induction on (Λ, n, m).

— If $m > 0$, construct a new $\Lambda' \leq \Lambda$ and a new T' such that $|T'| = n$ and $||T'|| = m - 1$ and h, T', Λ' ok and $\mathsf{safe}_{\Lambda',n}(T')$ by executing some thread's first ghost command. Then apply the induction hypothesis.
— Assume $m = 0$. We prove the goal by contradiction. Assume $\mathsf{bad}(h)$. We perform case analysis on the derivation of $\mathsf{bad}(h)$.
  — Assume $h \not\rightarrow$ and $\neg\mathsf{finished}(h)$. Then, by $\mathsf{safe}_{\Lambda,n}(T)$ each non-finished thread must be blocked on some waitable object whose wait level is less than its own obligations, and some other thread must have that waitable object as an obligation. Since, by safety of the thread table, finished threads have no obligations, this implies a cycle in the order on wait levels, which is absurd.
  — Assume $h \rightarrow h'$ and $\mathsf{bad}(h')$. By $\mathsf{safe}_{\Lambda,n}(T)$ we have that a Λ' and a T' exists such that $(\Lambda', |T'|, ||T'||) < (\Lambda, n, m)$ and h', T', Λ' ok and $\mathsf{safe}_{\Lambda',|T'|}(T')$. The goal follows by the induction hypothesis.

□

THEOREM A.10.

$$\forall t, \ell.\ t \vdash \{t.\mathsf{obs}(\mathbf{0}) * \mathsf{cp}(\ell)\}\ c\ \{t.\mathsf{obs}(\mathbf{0})\} \Rightarrow \neg\mathsf{bad}\ \{\!\!\{t.\mathsf{thread}(c)\}\!\!\}$$

PROOF. By Lemma A.7, there exists $\bar{\bar{c}}$ such that $\mathsf{safe}_{\Lambda,n}(t, H, \bar{\bar{c}}, c, t.\mathsf{obs}(\mathbf{0}))$, where $\Lambda = \{\!\!\{\ell\}\!\!\}$, $n = |c|$, and $H = \{\!\!\{t.\mathsf{obs}(\mathbf{0}), \mathsf{cp}(\ell)\}\!\!\}$. It follows that $\mathsf{safe}_{\Lambda,n}(T)$ where $T = \{\!\!\{(t, \mathsf{nonJoinable}, \mathbf{0}, H, \bar{\bar{c}}, c, \mathsf{true})\}\!\!\}$. Furthermore, we have h, T, Λ ok where $h = \{\!\!\{t.\mathsf{thread}(c)\}\!\!\}$. We obtain the goal by Lemma A.9. □

ACKNOWLEDGMENTS

The authors would like to thank K. Rustan M. Leino, Peter Müller, Matthew Parkinson, Jan Smans, Alexander J. Summers, and the anonymous reviewers for helpful suggestions. This work was funded in part by EU project ADVENT and by the Flemish Research Fund (FWO).

REFERENCES

Robert Atkey. 2011. Amortised Resource Analysis with Separation Logic. *LMCS* 7, 2:17 (2011).

Richard Bornat, Cristiano Calcagno, Peter W. O'Hearn, and Matthew J. Parkinson. 2005. Permission accounting in separation logic. In *POPL*.

Pontus Boström and Peter Müller. 2015. Modular verification of finite blocking in non-terminating programs. In *ECOOP*.

John Boyland. 2003. Checking Interference with Fractional Permissions. In *SAS*.

Arthur Charguéraud. 2011. Characteristic Formulae for the Verification of Imperative Programs. In *ICFP*.

Arthur Charguéraud and Fran cois Pottier. 2011. Machine-Checked Verification of the Correctness and Amortized Complexity of an Efficient Union-Find Implementation. In *ITP*.

Pedro da Rocha Pinto, Thomas Dinsdale-Young, Philippa Gardner, and Julian Sutherland. 2016. Modular termination verification for non-blocking concurrency. In *ESOP*.

Ádám Darvas and Peter Müller. 2006. Reasoning About Method Calls in Interface Specifications. *Journal of Object Technology* 5, 5 (2006).

Nachum Dershowitz and Zohar Manna. 1979. Proving Termination with Multiset Orderings. *Commun. ACM* 22, 8 (Aug. 1979).

Jan Hoffmann, Michael Marmar, and Zhong Shao. 2013. Quantitative reasoning for proving lock-freedom. In *LICS*.

Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. 2001. Featherweight Java: A Minimal Core Calculus for Java and GJ. *ACM TOPLAS* 23, 3 (May 2001).

Bart Jacobs. 2015. Provably live exception handling. In *FTfJP*.

Bart Jacobs. 2018. *Modular Termination Verification: Machine-Checked Proofs*. Zenodo. Record 1248801.

Bart Jacobs (Ed.). 2018. *VeriFast 18.02*. Zenodo. Record 1182724.

Bart Jacobs, Dragan Bosnacki, and Ruurd Kuiper. 2015a. Modular termination verification. In *ECOOP*.

Bart Jacobs, Dragan Bosnacki, and Ruurd Kuiper. 2015b. *Modular termination verification: extended version*. Technical Report CW 680. Dept. Comp. Sci., KU Leuven.

Bart Jacobs and Frank Piessens. 2009. Failboxes: provably safe exception handling. In *ECOOP*.

Daan Leijen. 2014. Koka: Programming with Row Polymorphic Effect Types. In *Mathematically Structured Functional Programming*.

K. Rustan M. Leino. 2010. Dafny: An Automatic Program Verifier for Functional Correctness. In *LPAR*.

K. Rustan M. Leino and Ronald Middelkoop. 2009. Proving Consistency of Pure Methods and Model Fields. In *FASE*.

K. Rustan M. Leino, Peter Müller, and Jan Smans. 2010. Deadlock-free channels and locks. In *ESOP*.

Keiko Nakata and Tarmo Uustalu. 2009. Trace-Based Coinductive Operational Semantics for While. In *TPHOLs*.

Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. 2001. Local Reasoning about Programs that Alter Data Structures. In *CSL*.

Matthew J. Parkinson and Gavin M. Bierman. 2005. Separation logic and abstraction. In *POPL*.

Willem Penninckx, Bart Jacobs, and Frank Piessens. 2015. Sound, modular and compositional verification of the input/output behavior of programs. In *ESOP*.

John C. Reynolds. 2002. Separation logic: a logic for shared mutable data structures. In *LICS*.

Arsenii Rudich, Ádám Darvas, and Peter Müller. 2008. Checking Well-Formedness of Pure-Method Specifications. In *FM*.

Viktor Vafeiadis. 2011. Concurrent separation logic and operational semantics. In *MFPS*.

Frédéric Vogels, Bart Jacobs, and Frank Piessens. 2015. Featherweight VeriFast. *LMCS* 11, 3 (2015).

Received January 2016; revised —; accepted —