Modular Termination Verification

Bart Jacobs[1], Dragan Bosnacki[2], and Ruurd Kuiper[2]

1 **iMinds-DistriNet, Department of Computer Science, KU Leuven, Belgium**
`bart.jacobs@cs.kuleuven.be`
2 **Eindhoven University of Technology, the Netherlands**
`{d.bosnacki,r.kuiper}@tue.nl`

Abstract

We propose an approach for the modular specification and verification of total correctness properties of object-oriented programs. We start from an existing program logic for partial correctness based on separation logic and abstract predicate families. We extend it with *call permissions* qualified by an arbitrary ordinal number, and we define a specification style that properly hides implementation details, based on the ideas of using methods and bags of methods as ordinals, and exposing the bag of methods reachable from an object as an abstract predicate argument. These enable each method to abstractly request permission to call all methods reachable by it any finite number of times, and to delegate similar permissions to its callees. We illustrate the approach with several examples.

1998 ACM Subject Classification F.3.1 Logics and Meanings of Programs

Keywords and phrases Termination, program verification, modular verification, separation logic, well-founded relations

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2015.999

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.

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 and interfaces, developed independently, that satisfy a well-defined *module specification*. A module need not be *closed*: it may refer to classes and interfaces not defined by the module itself, but by other modules which it *imports*. A module should use only those elements (classes, interfaces, methods) from an


© Bart Jacobs, Dragan Bosnacki, and Ruurd Kuiper;
licensed under Creative Commons License CC-BY
29th European Conference on Object-Oriented Programming (ECOOP'15).
Editor: John Tang Boyland; pp. 999–1023

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

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.

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 paper, we propose such an approach.

For sequential programs, termination means absence of infinite loops and absence of infinite recursion. In the remainder of this paper, 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 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 absence of infinite recursion is not directly a module specification issue. Indeed, the number of non-intra-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, if we define an intra-module call as a call where the caller module knows statically (based on its specification and the specifications of the modules it imports) that the callee is internal to itself, then the number of non-intra-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 paper.

For example, consider the program of Fig. 1. In this program, we can consider interface RealFunc as well as classes Math, Identity, and Loopy to each constitute a separate module, where Math imports RealFunc, and Identity and Loopy each import both RealFunc and Math. Notice that none of these modules contain intra-module method calls; nonetheless, whereas method Identity.test correctly returns the derivative of the identity function at argument 42,

```
interface RealFunc {
  float apply(float x);
}
class Math {
  static float derivative(RealFunc f, float x)
  { f.apply(x + 1) − f.apply(x) }
}
```

```
class Identity implements RealFunc {
  float apply(float x) { x }
  static float test()
  { Math.derivative(new Identity(), 42) }
}
class Loopy implements RealFunc {
  float apply(float x)
  { Math.derivative(this, x) }
  static float test()
  { Math.derivative(new Loopy(), 42) }
}
```

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

method Loopy.test performs infinite inter-module recursion between methods Loopy.apply and Math.derivative.

In this paper, 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 Identity.test;
- 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.test 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 Math.derivative uses only the specification of module RealFunc, and does not depend on the existence or non-existence, or the content, of modules Identity and Loopy, and the proof of Identity.test uses only the specification of Math.derivative 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 Math.derivative's specification is satisfied equally by alternative implementations that are more complex (and more accurate).

Our approach is based on the observation that any dynamically bound call is a call on an *object*. This object, together with the objects reachable from it via field dereferences, constitutes a *data structure*. At any point during program execution, the data structures existing in memory at that point are of finite size, and were composed of objects of classes from different modules in a finite number of composition steps. The core idea of our approach, then, is to associate with each data structure, at each point in time, a *dynamic depth*. This is roughly the number of objects in the data structure. More precisely, to allow each module, even while operating on a data structure, to create new data structures composed from classes in lower-layer modules and perform calls on them, we track each object's module, so the dynamic depth is (more or less) the multiset of modules contributing objects to the data structure. By using these dynamic depths as part of a recursion measure, we obtain a specification style that performs proper information hiding.

The rest of this paper is structured as follows. To define our approach precisely, we start from an existing modular specification and verification approach for partial correctness of object-oriented programs, based on *separation logic* to deal with aliased mutable memory and *abstract predicate families* to achieve properly abstract specifications. We recall this existing approach in Sec. 2. In Sec. 3, we extend the program logic of this partial correctness approach with *call permissions* to obtain a program logic for total correctness. This logic is based on the well-known tools for reasoning about termination, *well-founded relations* and *ordinal numbers*. However, this logic is not the main contribution of the paper. The main issue addressed in this paper is: how to use this logic to write module specifications that are both *expressive* and *abstract*? In Sec. 4, in three steps we build up our modular specification approach, and we illustrate and motivate it through a sequence of examples. In Sec. 5, we briefly discuss how we added support for our approach to our program verification tool VeriFast. We discuss related work in Sec. 6 and we conclude in Sec. 7.

2 Separation logic and abstract predicate families

We start from an existing approach for modular specification and verification of partial correctness properties of object-oriented programs, based on separation logic [9, 2] and abstract predicate families [10]. We introduce the approach informally in Sec. 2.1. We formally define the approach in Sec. 2.2.

2.1 An Example

We present our approach in the context of a simple Java-like programming language. Consider the program in Fig. 4. It defines an interface IntFunc, two classes, PlusN and Twice, that implement the interface, and a class Program with a method main that composes a complex object and performs a dynamically bound call on it. An object **new** PlusN(y) represents a function that takes an integer x and returns the value $x + y$. An object **new** Twice(f), where f is itself an IntFunc object, represents a function that maps a value x to $f(f(x))$.

Annotations, which have no effect on the run-time behavior of the program and serve only for modular specification and verification, are shown on a gray background. Besides the presence of annotations, the main differences of our programming language with Java are inspired by Scala: fields are declared in a parenthesized list after the class name instead of in the body of the class; **new** expressions specify an initial value for each field; and the final expression in a method body determines the return value, without the need for a **return** keyword. (Note: unlike in Scala, field names are not in scope in the class body; to access a field f, one must write this.f.)

To enable modular verification, each method has a *contract*, i.e. a specification consisting of a precondition and a postcondition (prefixed by keywords **req** and **ens**, respectively). Note, however, that no contract is declared explicitly for methods apply of classes PlusN and Twice; they inherit their contract from method apply of interface IntFunc, which they override.

While this example program does not perform heap mutation (i.e. it does not modify any fields of any objects), our approach supports this fully. Therefore, method specifications should specify not only what should be true on entry to the method and what should be true on exit from the method, but also which object fields are modified by the method, and which are not. This is known as the *frame condition*. For this purpose we use *separation logic*. One way to understand separation logic, when applied to a Java-like programming language with garbage collection, is by saying that it drops the assumption that reachable

objects are allocated. Furthermore, we drop the assumption that all fields of a given object are allocated together. When verifying a program using separation logic, we need to prove that whenever the program accesses a field, that field is allocated. As a result, a method's precondition needs to state which fields it expects to be allocated. Since a verified method accesses only fields whose allocatedness is asserted by its precondition, we can infer that any fields that are allocated at a given call site but whose allocatedness is not asserted by the callee's precondition, remain unchanged by the call.

For example, consider the following example program:

```
class Account(int balance) {
  static void transfer(Account from, Account to, int amount)
    req from.balance ↦ b1 ∗ to.balance ↦ b2;
    ens from.balance ↦ b1 − amount ∗ to.balance ↦ b2 + amount;
  {
    int bal1 := from.balance; from.balance := bal1 − amount;
    int bal2 := to.balance; to.balance := bal2 + amount;
  }
}
```

This program transfers an amount of money between two bank accounts. The separation logic assertion from.balance $\mapsto$ b1 asserts that field balance of object from is allocated and has value b1. Such a *points-to assertion* is the only way in separation logic assertions to specify the value of a field. This way, it is syntactically enforced that a separation logic assertion does not refer to the value of unallocated fields.

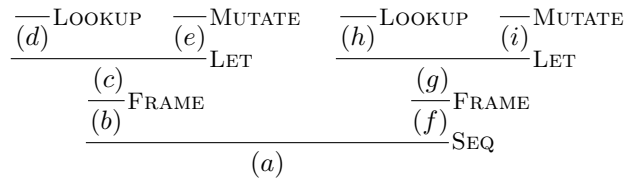
A *separating conjunction* $P * Q$, where P and Q are separation logic assertions, asserts that the heap (i.e. the set of allocated fields, with their values) can be split into two disjoint parts (i.e. where all of the fields that are allocated in one part are not allocated in the other part) such that P holds for one part, and Q holds for the other part. Therefore, it follows from the precondition of transfer that from and to are not the same object.

Notice that the variables b1 and b2 are free variables of the contract of transfer. The meaning of such free variables in this paper is as follows: free variables of the precondition are implicitly universally quantified at the level of the contract; their scope extends to the postcondition as well. That is, method transfer should satisfy its contract for all possible values of b1 and b2. (Variables that are free only in the postcondition are existentially quantified at the level of the postcondition; see the examples later in this paper.)

Verifying a method with precondition P, postcondition Q, and body B, means proving the *Hoare triple* $\vdash \{P\}\ B\ \{Q\}$ using the *proof rules* of the program logic. The proof rules of the logic of this section are shown in Fig. 7. Notice that the precondition of proof rule Mutate mentions only the field being modified. Information about other fields can be preserved using rule Frame. Notice also that there is no proof rule for sequential composition. Indeed, we treat a sequential composition $c; c'$ as a shorthand for a **let** command $\tau\ x = c; c'$, for some arbitrary type τ and some fresh variable x. As a result, we can derive the usual proof rule for sequential composition:

$$\textsc{Seq}\ \frac{\vdash \{P\}\ c\ \{Q\} \qquad \vdash \{Q\}\ c'\ \{R\}}{\vdash \{P\}\ c; c'\ \{R\}}$$

A proof tree for method transfer is shown in Fig. 2. We abbreviated the identifiers in obvious ways. Notice that we treat assertions semantically; e.g. since the assertions f.b $\mapsto$ b1−a ∗ t.b $\mapsto$ b2 and t.b $\mapsto$ b2 ∗ f.b $\mapsto$ b1−a are equivalent, we treat them as equal.



(a) $\{\mathsf{f.b} \mapsto \mathsf{b1} * \mathsf{t.b} \mapsto \mathsf{b2}\}$
$\mathbf{int}\ \mathsf{bal1} := \mathsf{f.b}; \mathsf{f.b} := \mathsf{bal1} - \mathsf{a}; \mathbf{int}\ \mathsf{bal2} := \mathsf{t.b}; \mathsf{t.b} := \mathsf{bal2} + \mathsf{a}$
$\{\mathsf{f.b} \mapsto \mathsf{b1} - \mathsf{a} * \mathsf{t.b} \mapsto \mathsf{b2} + \mathsf{a}\}$
(b) $\{\mathsf{f.b} \mapsto \mathsf{b1} * \mathsf{t.b} \mapsto \mathsf{b2}\}\ \mathbf{int}\ \mathsf{bal1} := \mathsf{f.b}; \mathsf{f.b} := \mathsf{bal1} - \mathsf{a}\ \{\mathsf{f.b} \mapsto \mathsf{b1} - \mathsf{a} * \mathsf{t.b} \mapsto \mathsf{b2}\}$
(c) $\{\mathsf{f.b} \mapsto \mathsf{b1}\}\ \mathbf{int}\ \mathsf{bal1} := \mathsf{f.b}; \mathsf{f.b} := \mathsf{bal1} - \mathsf{a}\ \{\mathsf{f.b} \mapsto \mathsf{b1} - \mathsf{a}\}$
(d) $\{\mathsf{f.b} \mapsto \mathsf{b1}\}\ \mathsf{f.b}\ \{\mathsf{f.b} \mapsto \mathsf{b1} \wedge \mathsf{res} = \mathsf{b1}\}$
(e) $\{\mathsf{f.b} \mapsto \mathsf{b1}\}\ \mathsf{f.b} := \mathsf{b1} - \mathsf{a}\ \{\mathsf{f.b} \mapsto \mathsf{b1} - \mathsf{a}\}$
(f) $\{\mathsf{f.b} \mapsto \mathsf{b1}{-}\mathsf{a} * \mathsf{t.b} \mapsto \mathsf{b2}\}\ \mathbf{int}\ \mathsf{bal2} := \mathsf{t.b}; \mathsf{t.b} := \mathsf{bal2} + \mathsf{a}\ \{\mathsf{f.b} \mapsto \mathsf{b1}{-}\mathsf{a} * \mathsf{t.b} \mapsto \mathsf{b2} + \mathsf{a}\}$
(g) $\{\mathsf{t.b} \mapsto \mathsf{b2}\}\ \mathbf{int}\ \mathsf{bal2} := \mathsf{t.b}; \mathsf{t.b} := \mathsf{bal2} + \mathsf{a}\ \{\mathsf{t.b} \mapsto \mathsf{b2} + \mathsf{a}\}$
(h) $\{\mathsf{t.b} \mapsto \mathsf{b2}\}\ \mathsf{t.b}\ \{\mathsf{t.b} \mapsto \mathsf{b2} \wedge \mathsf{res} = \mathsf{b2}\}$
(i) $\{\mathsf{t.b} \mapsto \mathsf{b2}\}\ \mathsf{t.b} := \mathsf{b2} + \mathsf{a}\ \{\mathsf{t.b} \mapsto \mathsf{b2} + \mathsf{a}\}$

Figure 2 Proof tree for method transfer

We show the same proof tree in the more convenient form of a *proof outline* in Fig. 3.

By the soundness of the program logic, the fact that we succeeded in proving this Hoare triple implies that every execution of method transfer that starts in a state that satisfies the precondition (for certain values of b1 and b2) will not access unallocated memory and, if it terminates, its final state satisfies the postcondition (for the same values of b1 and b2).

Returning now to the example program of Fig. 4, we see that method apply of class PlusN accesses field this.y; therefore, its precondition should assert that this field is allocated. However, since method apply overrides the corresponding method of interface IntFunc, we have to conclude that the precondition of method apply in interface IntFunc should assert that this.y is allocated. Clearly, it would not make sense for the interface method's contract to assert this directly. Rather, at the level of the interface, method apply's precondition should assert abstractly that whatever fields belong to the object's representation should be allocated. The specification approach supports this by means of *abstract predicate families*. An *abstract predicate* is simply a named and possibly parameterized separation logic assertion. An *abstract predicate family* is an abstract predicate declared at the level of an interface, and defined at the level of each of the classes that implement the interface. Predicate IntFunc declared in interface IntFunc is such an abstract predicate family.[1] Its intended meaning at the level of interface IntFunc is that it asserts the allocatedness of the fields belonging to the IntFunc object, as well as any validity constraints over their values. It corresponds to what is known as a *class invariant* in some other modular verification approaches. It is then natural that method apply asserts this predicate in its precondition and in its postcondition.

Notice that class PlusN defines this predicate to assert allocatedness of field this.y. (The underscore denotes existential quantification of the field value; i.e. the assertion does not

[1] In this paper we adopt the convention of using the name of the interface also as the name of its abstract predicate family (when it declares exactly one abstract predicate family, which is usually the case); however, this is an arbitrary choice. Another reasonable name would be valid.

$\{\mathsf{f.b} \mapsto \mathsf{b1} * \mathsf{t.b} \mapsto \mathsf{b2}\}$
$\{\mathsf{f.b} \mapsto \mathsf{b1}\}$ FRAME
int bal1 := f.b;
$\{\mathsf{f.b} \mapsto \mathsf{b1} \wedge \mathsf{bal1} = \mathsf{b1}\}$
f.b := bal1 − a;
$\{\mathsf{f.b} \mapsto \mathsf{b1} - \mathsf{a}\}$
$\{\mathsf{f.b} \mapsto \mathsf{b1} - \mathsf{a} * \mathsf{t.b} \mapsto \mathsf{b2}\}$
$\{\mathsf{t.b} \mapsto \mathsf{b2}\}$ FRAME
int bal2 := t.b;
$\{\mathsf{t.b} \mapsto \mathsf{b2} \wedge \mathsf{bal2} = \mathsf{b2}\}$
t.b := bal2 + a
$\{\mathsf{t.b} \mapsto \mathsf{b2} + \mathsf{a}\}$
$\{\mathsf{f.b} \mapsto \mathsf{b1} - \mathsf{a} * \mathsf{t.b} \mapsto \mathsf{b2} + \mathsf{a}\}$

Figure 3 Proof outline for method transfer

assert anything about the field value.) Therefore, method apply of class PlusN verifies.

Method apply of class Twice calls the apply method of the object pointed to by its f field. Therefore, the definition of predicate IntFunc at the level of class Twice asserts not only the allocatedness of field this.f, but also the predicate f.IntFunc.[2]

2.2 Formal Definition

We formally define the program logic for partial correctness that we start from.

The syntax of the programming language and the annotations is shown in Fig. 5.

We assume a set of interface names $\iota \in \mathit{ItfNames}$ and a set of class names $C \in \mathit{ClassNames}$. The types τ of the programming language include at least the types **int** of integers and **bool** of booleans, and the interface types ι and class types C. Correspondingly, the values $v \in \mathit{Values}$ of the programming language include at least the integers $z \in \mathbb{Z}$ and the booleans $b \in \mathbb{B}$, and the object references $o \in \mathit{ObjRefs}$. We will assume additional types and values whenever useful for particular examples.

The expressions include the literal values v, the variable references x, and the pure operations $op(\overline{e})$ that map a sequence of argument values to a result value. The separation logic assertions P include the boolean expressions e, asserting that the expression evaluates to true; the points-to assertions $e.f \mapsto e$, asserting that the indicated field is present in the heap and holds the indicated value; the separating conjunction $P*P$, asserting that the heap can be split into two parts such that one conjunct holds in one part, and the other conjunct holds in the other part; regular conjunction and disjunction; and *predicate assertions* $e.p(\overline{e})$, asserting that the indicated predicate holds with the indicated argument values. p ranges

[2] Note that such a recursive reference to predicate IntFunc inside a definition of IntFunc never causes well-definedness problems, provided that each reference to a predicate inside a predicate definition is in a *positive position*, i.e. not underneath a negation or on the left-hand side of an implication; in that case, the set of predicate definitions of a program, seen as a system of equations, always has a solution. See also Sec. 2.2.

```
class PlusN(int y) implements IntFunc {
  predicate IntFunc() = this.y ↦ _;
  int apply(int x)
  { int y := this.y; x + y }
  static IntFunc createPlusN(int y)
    req true; ens result.IntFunc();
  { new PlusN(y) }
}
class Program {
  static void main()
    req true; ens true;
  {
    IntFunc f1 := PlusN.createPlusN(10);
    IntFunc f2 := Twice.createTwice(f1);
    IntFunc f3 := Twice.createTwice(f2);
    f3.apply(42)
  }
}
```

```
interface IntFunc {
  predicate IntFunc();
  int apply(int x);
    req this.IntFunc();
    ens this.IntFunc();
}
class Twice(IntFunc f)
    implements IntFunc {
  predicate IntFunc() =
    this.f ↦ f * f.IntFunc();
  int apply(int x) {
    IntFunc f := this.f;
    int y := f.apply(x);
    f.apply(y)
  }
  static IntFunc createTwice(IntFunc f)
    req f.IntFunc();
    ens result.IntFunc();
  { new Twice(f) }
}
```

Figure 4 Example program annotated with partial correctness specifications

$$
\begin{array}{rcl}
\tau & ::= & \textbf{int} \mid \textbf{bool} \mid \iota \mid C \mid \cdots \\
e & ::= & v \mid x \mid op(\overline{e}) \\
P & ::= & e \mid e.f \mapsto e \mid P * P \mid P \wedge P \mid P \vee P \mid e.p(\overline{e}) \\
c & ::= & e \mid \tau\ x := c; c \mid \textbf{if}\ e\ \textbf{then}\ c\ \textbf{else}\ c \mid \{\ c\ \} \\
 & & \mid C.m(\overline{e}) \mid e.m(\overline{e}) \mid \textbf{new}\ C(\overline{e}) \mid e.f \mid e.f := e \\
pdecl & ::= & \textbf{predicate}\ p(\overline{\tau\ x}); \\
pdef & ::= & \textbf{predicate}\ p(\overline{\tau\ x}) = P; \\
imdef & ::= & \tau\ m(\overline{\tau\ x});\ \textbf{req}\ P;\ \textbf{ens}\ P; \\
mkind & ::= & \textbf{static} \mid \textbf{instance} \\
cmdef & ::= & mkind\ \tau\ m(\overline{\tau\ x})\ \textbf{req}\ P;\ \textbf{ens}\ P;\ \{\ c\ \} \\
idef & ::= & \textbf{interface}\ \iota\ \{\ \overline{pdecl}\ \ \overline{imdef}\ \} \\
cdef & ::= & \textbf{class}\ C(\overline{\tau\ f})\ \textbf{implements}\ \iota\ \{\ \overline{pdef}\ \ \overline{cmdef}\ \} \\
tdef & ::= & idef \mid cdef \\
program & ::= & \overline{tdef}
\end{array}
$$

Figure 5 Syntax of the programming language and the annotations

over predicate names.

Like expressions, commands c return a value; unlike expressions, they may also access the heap and have side-effects. The commands include the expressions; a **let**-like construct $\tau\ x := c; c'$ that first executes c, binds the result to variable x of type τ, and then executes c'; conditional commands; parenthesized commands; static and instance method calls; object creation, field lookup, and field mutation commands.

A predicate declaration specifies a predicate name and a parameter list; a predicate definition additionally specifies a body.

An interface method specifies a return type, a method name, a parameter list, and a contract consisting of a precondition and a postcondition. A class method additionally specifies a kind (kind **instance** is the default and is usually left implicit) and a body.

An interface definition declares a number of predicate families and a number of interface methods. A class definition declares a list of fields (empty if omitted), an implemented interface (interface Empty, that declares no predicate families and no methods, if omitted), a number of predicate family instances, and a number of class methods.

The type definitions are the interface definitions and the class definitions. A program is a sequence of type definitions.

We assume a function $\mathsf{classOf} : \mathit{ObjRefs} \to \mathit{ClassNames}$ such that infinitely many object references map to any given class. A heap $h \in \mathit{Heaps} = \mathit{ObjRefs} \times \mathit{FieldNames} \rightharpoonup \mathit{Values}$ is a partial function from pairs of object references and field names to values. We do not allow instantiation of classes that have no fields; therefore, the set of allocated objects can be derived from $\mathrm{dom}(h)$.

We define the semantics of programs by means of a big-step relation $(h, c) \Downarrow \gamma$ that relates a pre-heap and a closed command (i.e. a command with no free variables) to an outcome γ, which is either of the form (n, v, h') where $n \in \mathbb{N}$ is the number of execution steps performed, v is the result value, and h' is the post-heap, or an *exception* E, which is either $\textbf{Failure}(n)$, where $n \in \mathbb{N}$ is the number of execution steps performed, or **Divergence**. We define $n + \gamma$ as follows: $n + (n', v, h') = (n + n', v, h')$; $n + \textbf{Failure}(n') = \textbf{Failure}(n + n')$; $n + \textbf{Divergence} = \textbf{Divergence}$. We define the big-step relation coinductively [8] by means of the rules shown in Fig. 6.

$$\begin{array}{rcl} \gamma & ::= & (n, v, h) \mid E \\ E & ::= & \mathbf{Failure}(n) \mid \mathbf{Divergence} \end{array} \qquad (h, v) \Downarrow (1, v, h)$$

$$\frac{(h, c) \Downarrow (n, v, h') \qquad (h', c'[v/x]) \Downarrow \gamma}{(h, \tau\ x := c; c') \Downarrow n + \gamma} \qquad \frac{(h, c) \Downarrow E}{(h, \tau\ x := c; c') \Downarrow 1 + E}$$

$$\frac{\mathbf{class}\ C \cdots \{ \cdots\ \mathbf{static}\ \tau\ m(\overline{\tau\ x})\ \{\ c\ \}\ \cdots \} \qquad (h, c[\overline{v}/\overline{x}]) \Downarrow \gamma}{(h, C.m(\overline{v})) \Downarrow 1 + \gamma}$$

$$\frac{\begin{array}{c}\mathsf{classOf}(o) = C \\ \mathbf{class}\ C \cdots \{ \cdots\ \mathbf{instance}\ \tau\ m(\overline{\tau\ x})\ \{\ c\ \}\ \cdots \} \qquad (h, c[o, \overline{v}/\mathsf{this}, \overline{x}]) \Downarrow \gamma\end{array}}{(h, o.m(\overline{v})) \Downarrow 1 + \gamma}$$

$$\frac{\mathsf{classOf}(o) = C \qquad \mathbf{class}\ C(\overline{\tau\ f})\ \cdots \qquad h' = h \uplus \{\overline{o.f \mapsto v}\}}{(h, \mathbf{new}\ C(\overline{v})) \Downarrow (1, o, h')} \qquad \frac{(o, f) \in \mathrm{dom}(h)}{(h, o.f) \Downarrow (1, h((o, f)), h)}$$

$$\frac{(o, f) \notin \mathrm{dom}(h)}{(h, o.f) \Downarrow \mathbf{Failure}(1)} \qquad \frac{(o, f) \in \mathrm{dom}(h)}{(h, o.f := v) \Downarrow (1, v, h[(o, f) := v])} \qquad \frac{(o, f) \notin \mathrm{dom}(h)}{(h, o.f := v) \Downarrow \mathbf{Failure}(1)}$$

Figure 6 Coinductive big-step semantics $(h, c) \Downarrow \gamma$ of the programming language

Note that $h \uplus h'$ is undefined if $\mathrm{dom}(h) \cap \mathrm{dom}(h') \neq \emptyset$.

We now define the meaning of assertions. To interpret an assertion, we need an interpretation for the predicates it uses. A predicate interpretation I is a set $I \subseteq \mathit{ObjRefs} \times \mathit{PredNames} \times \mathit{Values}^* \times \mathit{Heaps}$. If $(o, p, \overline{v}, h) \in I$, this means that according to interpretation I, predicate assertion $o.p(\overline{v})$ is true in heap h. We now define the truth $I, h \vDash P$ of a closed assertion P under a predicate interpretation I and a heap h:

$$\begin{array}{lcl} I, h \vDash v & \Leftrightarrow & v = \mathsf{true} \\ I, h \vDash o.f \mapsto v & \Leftrightarrow & (o, f) \mapsto v \in h \\ I, h \vDash P * P' & \Leftrightarrow & \exists h_1, h_2.\ h = h_1 \uplus h_2 \wedge I, h_1 \vDash P \wedge I, h_2 \vDash P' \\ I, h \vDash P \wedge P' & \Leftrightarrow & I, h \vDash P \wedge I, h \vDash P' \\ I, h \vDash P \vee P' & \Leftrightarrow & I, h \vDash P \vee I, h \vDash P' \\ I, h \vDash o.p(\overline{v}) & \Leftrightarrow & (o, p, \overline{v}, h) \in I \end{array}$$

Given a predicate interpretation I, we can interpret the predicate definitions of a program to obtain a new predicate interpretation $F(I)$:

$$\frac{\begin{array}{c}\mathsf{classOf}(o) = C \\ \mathbf{class}\ C \cdots \{ \cdots\ \mathbf{predicate}\ p(\overline{\tau\ x}) = P;\ \cdots \} \qquad \overline{y} = \mathsf{FV}(P[\overline{v}/\overline{x}]) \qquad I, h \vDash P[\overline{v}/\overline{x}, \overline{w}/\overline{y}]\end{array}}{(o, p, \overline{v}, h) \in F(I)}$$

Notice that free variables in a predicate body are implicitly existentially quantified.

It is easy to check that F is monotonic: $I \subseteq I' \Rightarrow F(I) \subseteq F(I')$. (This would not be the case if our assertion language included negation or implication of assertions.) Therefore, by the Knaster-Tarski theorem, $I_{\mathrm{fix}} = \bigcap\{I \mid F(I) \subseteq I\}$ is the least fixpoint of F. We adopt I_{fix} as the meaning of predicates.

We are now ready to define the meaning of Hoare triples (for partial correctness):

$$\vDash \{P\}\ c\ \{Q\} \quad \Leftrightarrow \quad \forall h, \gamma.\ I_{\text{fix}}, h \vDash P \wedge (h, c) \Downarrow \gamma \Rightarrow \gamma \vDash Q$$

where satisfaction $\gamma \vDash Q$ of a postcondition by an outcome is defined as:

$$\textbf{Divergence} \vDash Q \qquad \frac{I_{\text{fix}}, h \vDash Q[v/\mathsf{res}]}{(n, v, h) \vDash Q}$$

The proof rules are shown in Fig. 7. Notice that in method contracts, free variables in the precondition are universally quantified across the contract; their scope extends to the postcondition as well. Variables that are free only in the postcondition are existentially quantified in the postcondition.

We say that a class implements an interface method if the class has a method of the same name, return type, and parameter list, whose body satisfies the interface method's contract. An alternative approach would be to check *compatibility* of the class method's contract with the interface method's contract [11].

The rule of consequence uses validity of implications. We define $\vDash P \Rightarrow P'$ as $\forall h.\ I_{\text{fix}}, h \vDash P \Rightarrow I_{\text{fix}}, h \vDash P'$. In particular, we can fold and unfold predicates if we know the class of the object:

$$\frac{\textbf{class}\ C \cdots \{\ \cdots\ \textbf{predicate}\ p(\overline{\tau\ x}) = P;\ \cdots\ \}}{\vDash \mathsf{classOf}(o) = C \wedge o.p(\overline{v}) \Rightarrow P[o/\mathsf{this}, \overline{v}/\overline{x}] \qquad \vDash \mathsf{classOf}(o) = C \wedge P[o/\mathsf{this}, \overline{v}/\overline{x}] \Rightarrow o.p(\overline{v})}$$

We assume $\vdash$ *program* ok.

The proof rules are sound: $\vdash \{P\}\ c\ \{Q\} \Rightarrow \vDash \{P\}\ c\ \{Q\}$. By the following lemma:

▶ **Lemma 1.**

$$\begin{array}{l}\forall n, P, c, Q.\ \vdash \{P\}\ c\ \{Q\} \Rightarrow \\ \quad \forall h, h_0, h_F, \gamma.\ h = h_0 \uplus h_F \wedge I_{\text{fix}}, h_0 \vDash P \wedge (h, c) \Downarrow \gamma \Rightarrow \\ \qquad \gamma \neq \textbf{Failure}(n) \\ \qquad \wedge\ (\forall h', v.\ \gamma = (n, v, h') \Rightarrow \exists h_0'.\ h' = h_0' \uplus h_F \wedge I_{\text{fix}}, h_0' \vDash Q[v/\mathsf{res}])\end{array}$$

Proof. By well-founded induction on n. Fix some n_0 and assume the lemma holds for all $n < n_0$. We prove that it holds for $n = n_0$. By nested induction on the derivation of $\vdash \{P\}\ c\ \{Q\}$. ◀

3 Call Permissions

In the preceding section, we recalled a state-of-the-art approach from the literature for modular specification and verification of *partial correctness* properties of object-oriented programs. We are now ready to present the contributions of this paper. In this section, we extend the program logic of the preceding section to obtain a logic for *total correctness* properties. However, it is possible to write specifications in the logic of this section that overly constrain implementations, i.e. that distinguish implementations that are observationally indistinguishable. In the next section, we present a specification style for writing specifications in the logic of this section that perform proper information hiding.

How to extend the program logic of Sec. 2 so that it verifies the absence of infinite recursion? We wish to impose an additional proof obligation at method call sites, such that during any program execution only a finite number of method calls occur. Since we

$$\textsc{Expr}\quad \vdash \{\mathsf{true}\}\ v\ \{\mathsf{res} = v\}$$

$$\textsc{Let}\quad \frac{\vdash \{P\}\ c\ \{Q\} \qquad \forall v.\ \vdash \{Q[v/\mathsf{res}]\}\ c'[v/x]\ \{R\}}{\vdash \{P\}\ \tau\ x := c; c'\ \{R\}}$$

$$\textsc{StaticCall}\quad \frac{\begin{array}{c}\textbf{class}\ C \cdots\ \{\ \cdots\ \textbf{static}\ \tau\ m(\overline{\tau\ x})\ \textbf{req}\ P;\ \textbf{ens}\ Q;\ \cdots\ \} \\ \overline{y} = \mathsf{FV}(P) \setminus \overline{x} \qquad \overline{z} = \mathsf{FV}(Q) \setminus \overline{x}, \mathsf{result}, \overline{y}\end{array}}{\vdash \{P[\overline{v}/\overline{x}, \overline{w}/\overline{y}]\}\ C.m(\overline{v})\ \{\exists \overline{w'}.\ Q[\overline{v}/\overline{x}, \overline{w}/\overline{y}, \overline{w'}/\overline{z}, \mathsf{res}/\mathsf{result}]\}}$$

$$\textsc{DynamicCall}\quad \frac{\begin{array}{c}\textbf{interface}\ \iota\ \{\ \cdots\ \tau\ m(\overline{\tau\ x});\ \textbf{req}\ P;\ \textbf{ens}\ Q;\ \cdots\ \} \\ \overline{y} = \mathsf{FV}(P) \setminus \mathsf{this}, \overline{x} \qquad \overline{z} = \mathsf{FV}(Q) \setminus \mathsf{this}, \overline{x}, \mathsf{result}, \overline{y}\end{array}}{\vdash \{P[o/\mathsf{this}, \overline{v}/\overline{x}, \overline{w}/\overline{y}]\}\ o.m(\overline{v})\ \{\exists \overline{w'}.\ Q[o/\mathsf{this}, \overline{v}/\overline{x}, \overline{w}/\overline{y}, \overline{w'}/\overline{z}, \mathsf{res}/\mathsf{result}]\}}$$

$$\textsc{New}\quad \frac{\textbf{class}\ C(\overline{\tau\ f})\ \cdots}{\vdash \{\mathsf{true}\}\ \textbf{new}\ C(\overline{v})\ \{\circledast_{(f,v)\in\overline{(f,v)}}\ \mathsf{res}.f \mapsto v\}}$$

$$\textsc{Lookup}\quad \vdash \{o.f \mapsto v\}\ o.f\ \{o.f \mapsto v \wedge \mathsf{res} = v\}$$

$$\textsc{Mutate}\quad \vdash \{o.f \mapsto \_\}\ o.f := v\ \{o.f \mapsto v\}$$

$$\textsc{Conseq}\quad \frac{\vDash P \Rightarrow P' \qquad \vdash \{P'\}\ c\ \{Q'\} \qquad \vDash Q' \Rightarrow Q}{\vdash \{P\}\ c\ \{Q\}}$$

$$\textsc{Frame}\quad \frac{\vdash \{P\}\ c\ \{Q\}}{\vdash \{P * R\}\ c\ \{Q * R\}}$$

$$\textsc{Disj}\quad \frac{\vdash \{P\}\ c\ \{Q\} \qquad \vdash \{P'\}\ c\ \{Q\}}{\vdash \{P \vee P'\}\ c\ \{Q\}}$$

$$\textsc{Program}\quad \frac{\mathit{program} = \overline{\mathit{tdef}} \qquad \overline{\vdash \mathit{tdef}\ \mathsf{ok}}}{\vdash \mathit{program}\ \mathsf{ok}}$$

$$\textsc{Interface}\quad \vdash \mathit{idef}\ \mathsf{ok}$$

$$\textsc{Class}\quad \frac{\overline{C \vdash \mathit{cmdef}\ \mathsf{ok}} \qquad \textbf{interface}\ \iota\ \{\ \overline{\mathit{pdecl}}\ \overline{\mathit{imdef}}\ \} \qquad \overline{\vdash C\ \mathsf{implements}\ \mathit{imdef}}}{\vdash \textbf{class}\ C(\cdots)\ \textbf{implements}\ \iota\ \{\ \overline{\mathit{pdef}}\ \overline{\mathit{cmdef}}\ \}\ \mathsf{ok}}$$

$$\textsc{StaticMethod}\quad \frac{\begin{array}{c}\overline{y} = \mathsf{FV}(P) \setminus \overline{x} \qquad \overline{z} = \mathsf{FV}(Q) \setminus \overline{x}, \mathsf{result}, \overline{y} \\ \forall \overline{v}, \overline{w}.\ \vdash \{P[\overline{v}/\overline{x}, \overline{w}/\overline{y}]\}\ c[\overline{v}/\overline{x}]\ \{\exists \overline{w'}.\ Q[\overline{v}/\overline{x}, \mathsf{res}/\mathsf{result}, \overline{w}/\overline{y}, \overline{w'}/\overline{z}]\}\end{array}}{C \vdash \textbf{static}\ \tau\ m(\overline{\tau\ x})\ \textbf{req}\ P;\ \textbf{ens}\ Q;\ \{\ c\ \}\ \mathsf{ok}}$$

$$\textsc{InstanceMethod}\quad \frac{\begin{array}{c}\overline{y} = \mathsf{FV}(P) \setminus \mathsf{this}, \overline{x} \qquad \overline{z} = \mathsf{FV}(Q) \setminus \mathsf{this}, \overline{x}, \mathsf{result}, \overline{y} \\ \forall o, \overline{v}, \overline{w}.\ \vdash \{P[o/\mathsf{this}, \overline{v}/\overline{x}, \overline{w}/\overline{y}]\}\ c[o/\mathsf{this}, \overline{v}/\overline{x}]\ \{\exists \overline{w'}.\ Q[o/\mathsf{this}, \overline{v}/\overline{x}, \mathsf{res}/\mathsf{result}, \overline{w}/\overline{y}, \overline{w'}/\overline{z}]\}\end{array}}{C \vdash \textbf{instance}\ \tau\ m(\overline{\tau\ x})\ \textbf{req}\ P;\ \textbf{ens}\ Q;\ \{\ c\ \}\ \mathsf{ok}}$$

$$\textsc{Implements}\quad \frac{\begin{array}{c}\textbf{class}\ C(\cdots)\ \cdots\ \{\ \cdots\ \textbf{instance}\ \tau\ m(\overline{\tau\ x})\ \textbf{req}\ P';\ \textbf{ens}\ Q';\ \{\ c\ \}\ \cdots\ \} \\ \overline{y} = \mathsf{FV}(P) \setminus \mathsf{this}, \overline{x} \qquad \overline{z} = \mathsf{FV}(Q) \setminus \mathsf{this}, \overline{x}, \mathsf{result}, \overline{y} \\ \forall o, \overline{v}, \overline{w}.\ \vdash \{P[o/\mathsf{this}, \overline{v}/\overline{x}, \overline{w}/\overline{y}]\}\ c[o/\mathsf{this}, \overline{v}/\overline{x}]\ \{\exists \overline{w'}.\ Q[o/\mathsf{this}, \overline{v}/\overline{x}, \mathsf{res}/\mathsf{result}, \overline{w}/\overline{y}, \overline{w'}/\overline{z}]\}\end{array}}{\vdash C\ \mathsf{implements}\ \tau\ m(\overline{\tau\ x});\ \textbf{req}\ P;\ \textbf{ens}\ Q;}$$

Figure 7 Proof rules of the program logic for partial correctness

are already using separation logic, which can be interpreted as a logic of permissions, we introduce the notion of *call permissions*. If we make available to a program's main method only a finite stock of call permissions, and each call consumes a call permission, then it follows that an execution can perform only finitely many calls.

Note that we should not count call permissions merely using a natural number. This would mean each method's specification would state an upper bound on the number of calls it performs. That would seem to require much tedious and brittle bookkeeping, and cause problems if the number of calls depends on nondeterministic phenomena such as user input.

The well-known solution to the counting issue in termination proofs is the use of *well-founded relations*. A well-founded relation is one that admits no infinite descending chains, or, equivalently, where each nonempty set has a minimal element. In this paper, we will more specifically use *ordinals*, well-founded relations that are additionally strict total orders, and for which useful conventional notations exist.[3][4]

We briefly review the ordinal theory used in this paper. The *finite ordinals* are the natural numbers, with their usual order. The set of finite ordinals is denoted ω. The *product* $\alpha \cdot \beta$ of two sets of ordinals is the set of pairs $(a, b) \in \alpha \times \beta$, with their *lexicographical ordering* (with the least significant element first): $(a, b) < (a', b')$ iff $b < b'$ or $b = b$ and $a < a'$. The *exponentiation* α^β of two sets of ordinals is the set of functions $f : \beta \to \alpha$ where only finitely many arguments map to nonzero values; the order is a generalization of the lexicographic order: $f < f'$ iff $f \neq f'$ and $f(b) < f'(b)$ where b is the maximum argument such that $f(b) \neq f'(b)$. In particular, ω^X yields the *multisets (or bags)* of elements of X, with *multiset order*. We denote bags using fat braces: $\{\!\{a, b, c\}\!\} = \mathbf{0} \uplus \{\!\{a\}\!\} \uplus \{\!\{b\}\!\} \uplus \{\!\{c\}\!\}$, where $\mathbf{0}$ denotes the empty multiset: $\mathbf{0} = \lambda x.\ 0$, and $M \uplus M'$ denotes multiset union: $M \uplus M' = (\lambda x.\ M(x) + M'(x))$. In order to descend down the multiset order starting from a multiset M, one can replace any element of M with any number of lesser elements of X, any number of times. For example, $\{\!\{0, 0, 1, 2, 2, 2\}\!\} < \{\!\{0, 0, 0, 3\}\!\}$.

Our program logic is based on the notion that at each point during a program's execution, it has a stock of call permissions in the form of a *bag of ordinals* $\Lambda \in \omega^{Ordinals}$ (for some fixed set of ordinals *Ordinals*). We admit ghost execution steps that reduce the stock of call permissions to a lesser one. Furthermore, at each call, an element is removed from the bag. It follows that the program terminates: an infinite execution would constitute an infinite descending chain in $\omega^{Ordinals}$.

We extend our separation logic with an assertion for call permissions:

$$P ::= o.f \mapsto v \mid P * P \mid \mathsf{call\_perm}(\alpha) \mid \cdots$$

We interpret assertions under a predicate interpretation, a heap and a stock of call permissions:

$$\begin{array}{lcl} I, h, \Lambda \vDash \mathsf{call\_perm}(\alpha) & \Leftrightarrow & \alpha \in \Lambda \\ I, h, \Lambda \vDash P * P' & \Leftrightarrow & \exists h_1, \Lambda_1, h_2, \Lambda_2.\ h = h_1 \uplus h_2 \wedge \Lambda = \Lambda_1 \uplus \Lambda_2 \\ & & \wedge\ I, h_1, \Lambda_1 \vDash P \wedge I, h_2, \Lambda_2 \vDash P' \end{array}$$

The meaning of Hoare triples is now defined as follows:

$$\vDash \{P\}\ c\ \{Q\} \quad \Leftrightarrow \quad \forall h, \Lambda, \gamma.\ I_{\text{fix}}, h, \Lambda \vDash P \wedge (h, c) \Downarrow \gamma \Rightarrow \gamma \vDash_\Lambda Q$$

[3] Our implementation of the proposed proof system (see Sec. 5) supports arbitrary well-founded relations.

[4] Technically, the ordinals are the equivalence classes of well-ordered sets under isomorphism. A well-ordered set is a set with a well-ordering, i.e. a well-founded strict total order. In an abuse of terminology, we will identify each such equivalence class with each of its members.

where satisfaction $\gamma \vDash_\Lambda Q$ of a postcondition by an outcome under a given stock of call permissions is now defined as follows:

$$\frac{\Lambda' \leq \Lambda \qquad I_{\text{fix}}, h, \Lambda' \vDash Q[v/\mathsf{res}]}{(n, v, h) \vDash_\Lambda Q}$$

Notice that divergence is no longer considered to satisfy a postcondition.

The only proof rules that change are the rule of consequence and the rules for method calls. For the rule of consequence of our logic, we use a notion of implication that allows weakening of the stock of call permissions:

$$\textsc{Conseq}\quad \frac{P \sqsubseteq P' \qquad \vdash \{P'\}\ c\ \{Q'\} \qquad Q' \sqsubseteq Q}{\vdash \{P\}\ c\ \{Q\}}$$

$$P \sqsubseteq P' \quad \Leftrightarrow \quad \forall h, \Lambda.\ I_{\text{fix}}, h, \Lambda \vDash P \Rightarrow \exists \Lambda' \leq \Lambda.\ I_{\text{fix}}, h, \Lambda' \vDash P'$$

We then have $\mathsf{call\_perm}(1) \sqsubseteq \mathsf{call\_perm}(0) * \mathsf{call\_perm}(0)$, and, more generally, $\mathsf{call\_perm}(1) \sqsubseteq \circledast_{i=1}^{n} \mathsf{call\_perm}(0)$, for any n, where $\circledast_{i=a}^{b} P(i)$ represents iterated separating conjunction:

$$\circledast_{i=a}^{b} P(i) = \begin{cases} \mathsf{true} & \text{if } b < a \\ P(a) * \circledast_{i=a+1}^{b} & \text{otherwise} \end{cases}$$

In case i does not appear in P, we abbreviate $\circledast_{i=1}^{n} P$ to $n \cdot P$. So, for any $\alpha' < \alpha$ and any n, we have $\mathsf{call\_perm}(\alpha) \sqsubseteq n \cdot \mathsf{call\_perm}(\alpha')$.

The proof rules for method calls are as follows:

$$\textsc{StaticCall}\quad \frac{\mathbf{class}\ C \cdots \{ \cdots\ \mathbf{static}\ \tau\ m(\overline{\tau\ x})\ \mathbf{req}\ P;\ \mathbf{ens}\ Q;\ \cdots \} \qquad \overline{y} = \mathsf{FV}(P) \setminus \overline{x} \qquad \overline{z} = \mathsf{FV}(Q) \setminus \overline{x}, \mathsf{result}, \overline{y}}{\vdash \{\mathsf{call\_perm}(\_) * P[\overline{v}/\overline{x}, \overline{w}/\overline{y}]\}\ C.m(\overline{v})\ \{\exists \overline{w'}.\ Q[\overline{v}/\overline{x}, \overline{w}/\overline{y}, \overline{w'}/\overline{z}, \mathsf{res}/\mathsf{result}]\}}$$

$$\textsc{InstanceCall}\quad \frac{\mathbf{interface}\ \iota\ \{ \cdots\ \tau\ m(\overline{\tau\ x});\ \mathbf{req}\ P;\ \mathbf{ens}\ Q;\ \cdots \} \qquad \overline{y} = \mathsf{FV}(P) \setminus \mathsf{this}, \overline{x} \qquad \overline{z} = \mathsf{FV}(Q) \setminus \mathsf{this}, \overline{x}, \mathsf{result}, \overline{y} \qquad \theta = o/\mathsf{this}, \overline{v}/\overline{x}, \overline{w}/\overline{y}}{\vdash \{\mathsf{call\_perm}(\_) * P[\theta]\}\ o.m(\overline{v})\ \{\exists \overline{w'}.\ Q[\theta, \overline{w'}/\overline{z}, \mathsf{res}/\mathsf{result}]\}}$$

Soundness follows from the following lemma:

▶ **Lemma 2.**

$$\begin{aligned} &\forall \Lambda, c, P, Q.\ \vdash \{P\}\ c\ \{Q\} \Rightarrow \\ &\quad \forall h, h_0, h_F, \Lambda_0, \Lambda_F, \gamma.\ h = h_0 \uplus h_F \wedge \Lambda = \Lambda_0 \uplus \Lambda_F \wedge I_{\text{fix}}, h_0, \Lambda_0 \vDash P \wedge (h, c) \Downarrow \gamma \Rightarrow \\ &\quad\quad \exists n, h', h_0', v, \Lambda', \Lambda_0'.\ \gamma = (n, v, h') \wedge h' = h_0' \uplus h_F \wedge \Lambda' = \Lambda_0' \uplus \Lambda_F \wedge \Lambda' \leq \Lambda \\ &\quad\quad\quad \wedge I_{\text{fix}}, h_0', \Lambda_0' \vDash Q[v/\mathsf{res}] \end{aligned}$$

Proof. By well-founded induction on $(|c|, \Lambda)$, where $|c|$ is the syntactic size of command c. By nested induction on the derivation of $\vdash \{P\}\ c\ \{Q\}$. ◀

4 Modular Specifications for Total Correctness

Now that we have call permissions, how do we use them to write modular specifications? Clearly, each method should require some call permissions from its caller in order to be able to perform calls itself. But how much? Which ordinal?

```
class Math {
  static int sqrtHelper(int x)
    req 0 ≤ x ∧ call_perm(Math.sqrtHelper);
    ens true
  { ··· }
  static int sqrt(int x)
    req 0 ≤ x ∧ call_perm(Math.sqrt);
    ens true;
  {
    {2 · call_perm(Math.sqrtHelper)}
    Math.sqrtHelper(x)
  }
}
```

```
class Program {
  static void main()
    req call_perm(Program.main);
    ens true;
  {
    {2 · call_perm(Math.sqrt)}
    Math.sqrt(42)
  }
}
```

Figure 8 A program with calls to lower-layer static methods only

We introduce our modular specification approach incrementally, as follows. We first consider the case where the program performs *upcalls*, static method calls into lower[5] layers, only, in Sec. 4.1, and obtain a modular specification approach for this setting. In Sec. 4.2, we extend this approach so that it supports recursive static methods. Finally, in Sec. 4.3, we consider the general case, with dynamically bound instance method calls, and obtain the final version of the approach. Each of Sec. 4.1–4.3 corresponds to a different value for parameter *Ordinals* of the logic of Sec. 3. In Sec. 4.4, we show additional examples, illustrating how the approach deals with the advanced scenarios of interface methods taking objects as arguments and programs written in continuation-passing style.

4.1 Upcalls Only

It should not be necessary for a method to ask specifically for call permissions for each call it performs. In particular, as discussed in Sec. 1, we assume a model where a program consists of layers where each layer is built on top of lower layers to offer functionality to higher layers. In this model, a method's contract should not reveal whether, or how often, the method calls into lower layers. This is clearly an implementation detail that should not concern the method's clients. To support this notion, we assume that class definitions that appear earlier in a program text constitute lower layers with respect to class definitions that appear later. Similarly, within a class definition, we assume that methods that appear earlier constitute lower layers with respect to methods that appear later. To enable abstraction over calls to lower-layer methods, we will use class methods as ordinals, ordered by their position in the program text.

Using this approach, if all calls in a program call static methods in lower layers, it is sufficient for each method $C.m$ to require $\mathsf{call\_perm}(C.m)$. Indeed, a valid proof outline is shown in Fig. 8.

Methods main and sqrt, before performing their nested call, using property $m' < m \Rightarrow \mathsf{call\_perm}(m) \sqsubseteq 2 \cdot \mathsf{call\_perm}(m')$, replace the incoming call permission qualified by their

[5] The clash of metaphors is unfortunate, but both terms are well-established.

```
class Math {
  static bool isOddIter(int x)
    req 0 ≤ x ∧
      x · call_perm(Math.isEvenIter);
    ens result = (x mod 2 = 1);
  {
    if x = 0 then false else
      Math.isEvenIter(x − 1)
  }
  static bool isEvenIter(int x)
    req 0 ≤ x ∧
      x · call_perm(Math.isEvenIter);
    ens result = (x mod 2 = 0);
  {
    if x = 0 then true else
      Math.isOddIter(x − 1)
  }
  static bool isOdd(int x)
    req 0 ≤ x ∧ call_perm(Math.isOdd);
    ens result = (x mod 2 = 1);
  { Math.isOddIter(x) }
  static bool isEven(int x)
    req 0 ≤ x ∧ call_perm(Math.isEven);
    ens result = (x mod 2 = 0);
  { Math.isEvenIter(x) }
}
```

Figure 9 Recursion measured by a natural number

own name with two copies of a call permission qualified by their callee's name (using rule Conseq on p. 1012). One copy is consumed at the start of the call, the other is passed into the callee as required by its precondition (per rule StaticCall on p. 1012).

Formally, in this subsection we take

$$\begin{aligned} \mathit{ClassMethods} &= \{C.m \mid \textbf{class } C \ \cdots \ \{ \ \cdots \ \tau \ m(\cdots) \ \cdots \ \}\} \\ \mathit{Ordinals} &= \mathit{ClassMethods} \end{aligned}$$

4.2 Static Recursion

Of course, in most programs not all calls, even if they call static methods, call lower-layer methods, especially since our programming language does not have loops. How to deal with recursive methods? First of all, we assume that lower layers are developed before higher layers, and therefore lower layers never call static methods in higher layers. It follows that each cycle in a program's graph of calls to static methods is contained entirely within a single module. In our approach, then, all such members of a recursive cycle should be private to the module, and therefore their contracts need not be abstract. Separate public methods should be provided that call into the cycle but are not part of it.

An example is shown in Fig. 9.

Notice that each of the members of the cycle requires call permission for the maximum member. The contracts of isOddIter and isEvenIter are not abstract, but those of isOdd and isEven are.

Notice that the coefficient of the call permission in the contracts of the recursive methods serves the role of the classical recursion measure.

However, sometimes a recursion measure cannot easily be expressed as a simple natural number. To support such recursion measures, we move, for the set of ordinals that we use to qualify call permissions, from methods to pairs of *local ordinals* and methods (where the

method is the most significant component), given some fixed set *LocOrd* of local ordinals. Formally, we take

$$Ordinals = LocOrd \cdot ClassMethods$$

Public methods $C.m$ should request call_perm$((0, C.m))$. A classic example is the Ackermann function:

```
class Math {
  static int ackermannIter(int m, int n)
    req 0 ≤ m ∧ 0 ≤ n ∧ call_perm(((m, n), Math.ackermannIter));
    ens result = Ack(m, n);
  {
    if n = 0 then m + 1
    else if m = 0 then Math.ackermannIter(1, n − 1)
    else {
      int r := Math.ackermannIter(m − 1, n);
      Math.ackermannIter(r, n − 1)
    }
  }
  static int ackermann(int m, int n)
    req 0 ≤ m ∧ 0 ≤ n ∧ call_perm((0, Math.ackermann));
    ens result = Ack(m, n);
  { Math.ackermannIter(m, n) }
}
```

The proof of method ackermann uses the property $\forall m, n.\ ((m, n), \mathsf{Math.ackermannIter}) < (0, \mathsf{Math.ackermann})$. This example assumes $\omega \cdot \omega \subseteq LocOrd$.

4.3 Dynamic Binding

We are now ready to consider the case of programs that call interface methods. Consider a method integrate for computing the integral of a real function over an interval:

```
interface RealFunc {
  double apply(double x);
}
class Math {
  static double integrate(double a, double b, RealFunc f)
  { ··· }
}
```

What call permissions should method integrate request of its caller? Clearly, the method should be allowed to call method apply of object f. And it should be allowed to call it not just once, but arbitrarily often. Since the calls of f.apply might occur inside recursive helper functions measured by arbitrary ordinals, there is no single ordinal that can obviously serve as an upper bound. Furthermore, method integrate should be allowed to pass f to library methods in lower layers, and those should themselves be specified abstractly without revealing how often they call f.

To solve this problem, we move, for the set of ordinals that we use to qualify call permissions, from pairs of local ordinals and methods to pairs of local ordinals and *bags of*

```
interface RealFunc {
  predicate RealFunc();
  double apply(double x);
    req this.RealFunc();
    ens this.RealFunc();
}
```

(a) Partial correctness

```
interface RealFunc {
  predicate RealFunc(MethodBag d);
  double apply(double x);
    req this.RealFunc(d) ∗ call_perm((0, d));
    ens this.RealFunc(d);
}
```

(b) Total correctness

Figure 10 Annotations for interface RealFunc

methods:

$$\begin{aligned} \mathit{MethodBags} &= \omega^{\mathit{ClassMethods}} \\ \mathit{Ordinals} &= \mathit{LocOrd} \cdot \mathit{MethodBags} \end{aligned}$$

Note that set *ClassMethods* includes both the static methods and the instance methods.

Public methods $C.m$ in programs that do not call interface methods should request call_perm$((0, \{\!|C.m|\!\}))$.

The following contract for method integrate allows it to call f.apply arbitrarily often, and to delegate a similar permission to lower-layer static methods:

```
static double integrate(double a, double b, RealFunc f)
  req call_perm((0, {|Math.integrate, f.apply|}));
```

Note that we use $o.m$ as a shorthand for classOf$(o).m$.

However, we are not done. Indeed, when calling f.apply, we need not just the call permission that is consumed by the call itself, but also the call permissions requested by f.apply's precondition. What should those be?

f.apply should be allowed to call static methods at layers below itself, so it should at least receive a call permission qualified by its own name. However, there are other methods that it should be allowed to call as well. Indeed, through the fields of its this object, i.e. through the fields of f, this method may be able to reach directly or indirectly any number of objects and might need to perform any number of calls on any number of methods thereof. The method should request permission for those calls as well.

But how can it abstractly request permission to call these methods, hidden inside its private data structures, with whose existence its clients should not otherwise be concerned?

We solve this problem by allowing the bag of methods reachable from an object to be named abstractly by exposing it as an argument of the predicate family that describes the object.

Consider first the partial-correctness specification for interface RealFunc in Fig. 10(a). We extend it for total correctness as shown in Fig. 10(b).

We adapt the contract of method integrate accordingly:

```
static double integrate(double a, double b, RealFunc f)
  req f.RealFunc(d) ∗ call_perm((0, {|Math.integrate|} ⊎ d));
```

A simple implementation of interface RealFunc is shown in Fig. 11. Notice that method createLinearFunc's postcondition provides an upper bound on d. This enables the caller (who is necessarily in a higher layer than createLinearFunc) to produce the call permissions

```
class LinearFunc(double a, double b) implements RealFunc {
  predicate RealFunc(MethodBag d) = this.a ↦ _ ∗ this.b ↦ _ ∧ d = {|this.apply|};
  double apply(double x) { double a := this.a; double b := this.b; a · x + b }
  static RealFunc createLinearFunc(double a, double b)
    req call_perm((0, {|LinearFunc.createLinearFunc|}));
    ens result.RealFunc(d) ∧ d < {|LinearFunc.createLinearFunc|};
  { new LinearFunc(a, b) }
}
```

Figure 11 A simple implementation of interface RealFunc

```
class Sum(RealFunc f1, RealFunc f2) implements RealFunc {
  predicate RealFunc(MethodBag d) =
    this.f1 ↦ f1 ∗ f1.RealFunc(d1) ∗ this.f2 ↦ f2 ∗ f2.RealFunc(d2)
    ∧ d = {|this.apply|} ⊎ d1 ⊎ d2;
  double apply(double x) {
    RealFunc f1 := this.f1; RealFunc f2 := this.f2;
    double r1 := f1.apply(x); double r2 := f2.apply(x); r1 + r2
  }
  static RealFunc createSum(RealFunc f1, RealFunc f2)
    req f1.RealFunc(d1) ∗ f2.RealFunc(d2) ∗ call_perm((0, {|Sum.createSum|} ⊎ d1 ⊎ d2));
    ens result.RealFunc(d) ∧ d < {|Sum.createSum|} ⊎ d1 ⊎ d2;
  { new Sum(f1, f2) }
}
```

Figure 12 An implementation of interface RealFunc

required to call result.apply. Notice also that this upper bound does not constrain method createLinearFunc's implementation, if we assume that a method only allocates (through **new**) objects of classes defined in its own layer or in lower layers.

A slightly more involved implementation is shown in Fig. 12. Notice that class Sum's instance of predicate family RealFunc defines its d parameter (which we call the *dynamic depth* since it gives a measure of the number of layers of abstraction of which the object is composed) as the multiset union of its own apply method and the referenced objects' dynamic depths. Indeed, as a general pattern, an object's dynamic depth should typically be defined as the union of its own methods and the dynamic depths of the objects stored in its fields. This allows the object to call those objects' methods, assuming their contracts follow the standard pattern exemplified by the contract of RealFunc.apply.

Notice also how this definition enables the successful verification of method apply.

Method createSum's precondition follows the general pattern: request a call permission qualified by a multiset that is the union of the method itself and the dynamic depths of any objects being passed into the method. Its postcondition follows a general pattern for methods that return a new object: the new object's dynamic depth is bounded by the same

```
class Math {
  static double integrateIter(double a, double dx, int n, RealFunc f)
    req 0 ≤ n ∧ f.RealFunc(d) ∗ call_perm((n, {|Math.integrateIter|} ⊎ d));
    ens f.RealFunc(d);
  {
    if n = 0 then 0 else {
      double y := f.apply(a); double ys := Math.integrateIter(a + dx, dx, n − 1, f);
      y × dx + ys
    }
  }
  static double integrate(double a, double b, RealFunc f)
  { Math.integrateIter(a, (b − a)/1000, 1000, f) }
}
class Program {
  static main()
    req call_perm((0, {|Program.main|}));
    ens true;
  {
    {12 · call_perm((0, {|2 · LinearFunc, 2 · Sum, Math|}))}
    RealFunc f1 := LinearFunc.createLinearFunc(2, 3);
    RealFunc f2 := LinearFunc.createLinearFunc(5, 6);
    RealFunc f3 := LinearFunc.createLinearFunc(5, 6);
    RealFunc f4 := Sum.createSum(f1, f2);
    RealFunc f5 := Sum.createSum(f3, f4);
    Math.integrate(0, 100, f5)
  }
}
```

Figure 13 An implementation of method integrate and a client program

multiset used to qualify the call permission in the precondition. It follows that any caller of this method can also call the new object's methods.

An implementation of method integrate and an example client program are shown in Fig. 13. The proof outline for method main starts by reducing the incoming call permission to twelve copies of a call permission that is greater than each of the call permissions required for the six calls and the six preconditions. (We use a class name as an abbreviation for its greatest method.) Indeed, we have the inequalities shown in Fig. 14.

4.4 Further Examples

An example of an interface method that takes as an argument another object is shown in Fig. 15. Notice that method intersects' precondition asserts a single call permission qualified by the multiset union of the dynamic depth of the receiver and the dynamic depth of the argument object.

Our approach supports methods written in continuation-passing style (CPS). To illustrate this, we add CPS versions of methods contains and intersects to the IntSet example. See

$$
\begin{array}{ll}
\mathsf{d1}, \mathsf{d2}, \mathsf{d3} & < \{\!|\mathsf{LinearFunc.createLinearFunc}|\!\} \\
\mathsf{d4} & < \{\!|\mathsf{Sum.createSum}|\!\} \uplus \mathsf{d1} \uplus \mathsf{d2} \\
& < \{\!|\mathsf{Sum.createSum}, \mathsf{LinearFunc.createLinearFunc}|\!\} \\
\mathsf{d5} & < \{\!|\mathsf{Sum.createSum}|\!\} \uplus \mathsf{d3} \uplus \mathsf{d4} \\
& < \{\!|2 \cdot \mathsf{LinearFunc.createLinearFunc}, 2 \cdot \mathsf{Sum.createSum}|\!\}
\end{array}
$$

Figure 14 Inequalities relevant for the proof of the integrate client program. Symbols $\mathsf{d1}, \ldots, \mathsf{d5}$ denote the dynamic depths of objects $\mathsf{f1}, \ldots, \mathsf{f5}$.

```
interface IntSet {
  predicate IntSet(MethodBag d);
  bool contains(int x);
    req this.IntSet(d) ∗ call_perm((0, d));
    ens this.IntSet(d);
  bool intersects(IntSet other);
    req this.IntSet(d) ∗ other.IntSet(do) ∗ call_perm((0, d ⊎ do));
    ens this.IntSet(d) ∗ other.IntSet(do);
}
class Empty() implements IntSet {
  predicate IntSet(MethodBag d) = d = {|this.∗|};
  bool contains(int x) { false }
  bool intersects(IntSet other) { false }
  static IntSet createEmpty()
    req call_perm((0, {|Empty.createEmpty|}));
    ens result.IntSet(d) ∧ d < {|Empty.createEmpty|};
  { new Empty() }
}
class Insert(int elem, IntSet set) implements IntSet {
  predicate IntSet(MethodBag d) =
    this.elem ↦ elem ∗ this.set ↦ set ∗ set.IntSet(ds) ∧ d = {|this.∗|} ⊎ ds;
  bool contains(int x) {
    int elem := this.elem;
    if x = elem then true else { IntSet set := this.set; set.contains(x) }
  }
  bool intersects(IntSet other) {
    int elem := this.elem; bool contains := other.contains(elem);
    if contains then true else { IntSet set := this.set; set.intersects(other) }
  }
  static IntSet createInsert(int elem, IntSet set)
    req set.IntSet(ds) ∗ call_perm((0, {|Insert.createInsert|} ⊎ ds));
    ens result.IntSet(d) ∧ d < {|Insert.createInsert|} ⊎ ds;
  { new Insert(elem, set) }
}
```

Figure 15 An interface method that takes as an argument another object. $\{|o.*|\}$ denotes the bag of all instance methods of o.

```
interface ContainsCont {
  predicate ContainsCont(IntSet set, MethodBag d);
  Nothing invoke(bool result);
    req this.ContainsCont(set, d) ∗ set.IntSet(d);
}
interface IntersectsCont {
  predicate IntersectsCont(IntSet set, MethodBag d, IntSet other, MethodBag do);
  Nothing invoke(bool result);
    req this.IntersectsCont(set, d, other, do) ∗ set.IntSet(d) ∗ other.IntSet(do);
}
interface IntSet {
  Nothing containsCPS(int x, ContainsCont cont);
    req this.IntSet(d) ∗ call_perm((0, d)) ∗ cont.ContainsCont(this, d);
  Nothing intersectsCPS(IntSet other, IntersectsCont cont);
    req this.IntSet(d) ∗ other.IntSet(do) ∗ call_perm((0, d ⊎ do))
      ∗ cont.ContainsCont(this, d, other, do);
}
```

Figure 16 Continuation-passing-style versions of contains and intersects. All postconditions are false and are not shown.

Figs. 16 and 17. After methods containsCPS and intersectsCPS are finished computing their result value, they do not return to the caller (as indicated by return type Nothing, which has no values); rather, they call method invoke of the *continuation* object cont, passing the result value as an argument. The predicate definitions and constructor methods remain unchanged and are not repeated.

Notice that the continuation interfaces do not follow the general specification pattern. Indeed, since the invoke methods are invoked only once, any call permissions they need can be passed via the ContainsCont or IntersectsCont predicate, respectively.

Notice that InvokeCont1.invoke and InvokeCont2.invoke each require a single call permission in order to perform the nested invoke call. It is passed via the predicate. The ordinal is irrelevant and is existentially quantified. In contrast, InvokeCont3.invoke needs to call set.intersectsCPS and for that needs a properly qualified call permission. The call permission that the InvokeCont3 object needs to pass to the InvokeCont2 object can be derived from it.

5 Implementation

We integrated the logic into the program verification tool VeriFast. Although VeriFast supports C and Java, for now we have added support for verification of termination only for C programs. We introduced the function specification clause **terminates**, to indicate that a function should terminate. In order to reduce specification overhead for functions that do not perform callbacks, our implementation offers a ghost command that allows a function to produce out of thin air any call permission whose bag of functions is less than

```
class Empty() implements IntSet {
  Nothing containsCPS(int x, ContainsCont cont) { cont.invoke(false) }
  Nothing intersectsCPS(IntSet other, IntersectsCont cont) { cont.invoke(false) }
}
class InsertCont1(ContainsCont cont) implements ContainsCont {
  predicate ContainsCont(IntSet set, MethodBag d) =
    this.cont ↦ cont ∗ cont.ContainsCont(set0, d0)
    ∗ set0.elem ↦ _ ∗ set0.set ↦ set ∗ call_perm(_) ∧ d0 = {|set0.∗|} ⊎ d;
  Nothing invoke(bool result) { ContainsCont cont := this.cont; cont.invoke(result) }
}
class InsertCont2(IntersectsCont cont) implements IntersectsCont {
  predicate IntersectsCont(IntSet set, MethodBag d, IntSet other, MethodBag do) =
    this.cont ↦ cont ∗ cont.IntersectsCont(set0, d0, other, do)
    ∗ set0.elem ↦ _ ∗ set0.set ↦ set ∗ call_perm(_) ∧ d0 = {|set0.∗|} ⊎ d;
  Nothing invoke(bool result) { IntersectsCont cont := this.cont; cont.invoke(result) }
}
class InsertCont3(Insert set0, IntSet other, IntersectsCont cont) implements ContainsCont {
  predicate ContainsCont(IntSet set, MethodBag d) =
    this.other ↦ set ∗ this.cont ↦ cont ∗ cont.IntersectsCont(set0, ds0, set, d)
    ∗ this.set0 ↦ set0 ∗ (set0.IntSet(ds0) ∧ ds0 = {|set0.∗|} ⊎ ds1)
    ∗ call_perm((0, {|InsertCont3.invoke|} ⊎ ds1 ⊎ d));
  Nothing invoke(bool result) {
    Insert set0 := this.set0; IntSet other := this.other; IntersectsCont cont := this.cont;
    if result then cont.invoke(true) else {
      IntSet set := set0.set; IntersectsCont cont1 := new InsertCont2(cont);
      set.intersectsCPS(other, cont1)
    }
  }
}
class Insert(int elem, IntSet set) implements IntSet {
  Nothing containsCPS(int x, ContainsCont cont) {
    int elem := this.elem; if x = elem then cont.invoke(true) else {
      IntSet set := this.set; ContainsCont cont1 := new InsertCont1(cont);
      set.containsCPS(x, cont1)
    }
  }
  Nothing intersectsCPS(IntSet other, IntersectsCont cont) {
    int elem := this.elem; ContainsCont cont1 := new InsertCont3(this, other, cont);
    other.containsCPS(elem, cont1)
  }
}
```

Figure 17 Implementations of containsCPS and intersectsCPS in classes Empty and Insert

itself (considered as a singleton bag). In exchange, our implementation consumes at a call site not just any call permission, but only a call permission whose function bag includes the function being called:

$$\frac{f \in d \qquad \overline{y} = \mathsf{FV}(P) \setminus \overline{x} \qquad \overline{z} = \mathsf{FV}(Q) \setminus \overline{x}, \mathsf{result}, \overline{y}}{f_0 \vdash \{\mathsf{call\_perm}((\alpha, d)) * P[\overline{v}/\overline{x}, \overline{w}/\overline{y}]\}\ f(\overline{v})\ \{\exists \overline{w'}.\ Q[\overline{v}/\overline{x}, \overline{w}/\overline{y}, \overline{w'}/\overline{z}, \mathsf{res}/\mathsf{result}]\}}\ \mathbf{function}\ f(\overline{x})\ \mathbf{req}\ P;\ \mathbf{ens}\ Q;\ \{\ c\ \}$$

$$\frac{\forall \overline{v}, \overline{w}.\ f \vdash \{P[\overline{v}/\overline{x}, \overline{w}/\overline{y}]\}\ c\ \{\exists \overline{w'}.\ Q[\overline{v}/\overline{x}, \overline{w}/\overline{y}, \overline{w'}/\overline{z}, \mathsf{res}/\mathsf{result}]\} \qquad \overline{y} = \mathsf{FV}(P) \setminus \overline{x} \qquad \overline{z} = \mathsf{FV}(Q) \setminus \overline{x}, \mathsf{result}, \overline{y}}{\vdash \mathbf{function}\ f(\overline{x})\ \mathbf{req}\ P;\ \mathbf{ens}\ Q;\ \{\ c\ \}\ \mathsf{ok}}$$

$$\frac{d < \{\!\{f\}\!\}}{f \vdash \{\mathsf{true}\}\ \mathbf{produce\_call\_perm}\ \{\mathsf{call\_perm}((\alpha, d))\}}$$

Our implementation is included in the latest VeriFast release, which is available at http://www.cs.kuleuven.be/~bartj/verifast/.

The distribution includes, in the directory examples/termination, the examples simple_recursion.c (the isEven example), ackermann.c, funcptr.c (corresponding to the IntFunc example of this paper), and cons.c (corresponding to the IntSet example).

6 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.

Koka [4] 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 [5] is a programming language that supports verification of termination, with powerful metrics. However, Dafny does not support dynamic binding of method calls.

Most closely related to ours is the work, e.g. [1, 12, 6], 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*. Our notion of *dynamic depth* of 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 [1, 6]. 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 objects comprising a data structure; rather, we track the multiset of the modules that define their classes.

7 Conclusion

We propose an approach for the modular specification and verification of total correctness properties of sequential 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.

We have implemented 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.

In the extended version of this paper [3], we discuss how to combine our approach with an approach for modular verification of absence of deadlock, such as [7], to obtain an approach for modular verification of termination of multithreaded programs. By using dynamic depths as wait levels, the approach allows acquisitions of private locks to be introduced into existing methods without changing their contracts. We also show how the approach supports proving termination of compare-and-swap loops, and proving liveness of non-terminating programs.

Acknowledgements

The authors would like to thank Matthew Parkinson for his helpful comments, and K. Rustan M. Leino for pointing out to them the usefulness of ordinals and multiset order for termination verification. This work was supported in part by EU project ADVENT and by project G.0058.13 of the Research Foundation - Flanders (FWO).

References

1 Ádám Darvas and Peter Müller. Reasoning about method calls in interface specifications. *Journal of Object Technology*, 5(5):59–85, 2006.
2 Samin S. Ishtiaq and Peter W. O'Hearn. BI as an assertion language for mutable data structures. In *POPL*, 2001.
3 Bart Jacobs, Dragan Bosnacki, and Ruurd Kuiper. Modular termination verification: extended version. Technical Report CW 680, Dept. Comp. Sci., KU Leuven, 2015.
4 Daan Leijen. Koka: Programming with row polymorphic effect types. In *Mathematically Structured Functional Programming*, 2014.
5 K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In *LPAR*, 2010.
6 K. Rustan M. Leino and Ronald Middelkoop. Proving consistency of pure methods and model fields. In *FASE*, 2009.
7 K. Rustan M. Leino, Peter Müller, and Jan Smans. Deadlock-free channels and locks. In *ESOP*, 2010.
8 Keiko Nakata and Tarmo Uustalu. Trace-based coinductive operational semantics for While. In *TPHOLs*, 2009.
9 Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In *CSL*, 2001.
10 Matthew J. Parkinson and Gavin M. Bierman. Separation logic and abstraction. In *POPL*, 2005.
11 Matthew J. Parkinson and Gavin M. Bierman. Separation logic, abstraction and inheritance. In *POPL*, 2008.
12 Arsenii Rudich, Ádám Darvas, and Peter Müller. Checking well-formedness of pure-method specifications. In *FM*, 2008.