Technical Report
CMU-CS-13-132: Object Propositions

Ligia Nistor+, Jonathan Aldrich+, Stephanie Balzer, and Hannes Mehnert*

+School of Computer Science, Carnegie Mellon University
*IT University of Copenhagen
{lnistor,aldrich,balzers}@cs.cmu.edu, hame@itu.dk

Abstract. The presence of aliasing makes modular verification of object-oriented code difficult. If multiple clients depend on the properties of an object, one client may break a property that others depend on.
We have developed a modular verification approach based on the novel abstraction of *object propositions*, which combine predicates and information about object aliasing. In our methodology, even if shared data is modified, we know that an object invariant specified by a client holds. Our permission system allows verification using a mixture of linear and nonlinear reasoning. This allows it to provide more modularity in some cases than competing separation logic approaches, because it can more effectively hide the exact aliasing relationships within a module. We validate our approach on an instance of the composite pattern that illustrates our system's practicality.

1 Introduction

We propose a method for modular verification of object-oriented code in the presence of aliasing, i.e., the existence of multiple references to the same object. The seminal work of Parnas [22] describes the importance of modular programming, where the information hiding criteria is used to divide the system into modules. In a world where software systems have to be continually changed in order to satisfy new (client) requirements, the software engineering principle of modular programming becomes crucial: it brings flexibility by allowing changes to one module without modifying the others. Since Java is one of the most popular programming languages used in industry, our verification methodology targets a Java-like language. The formal verification of modules should ideally follow the same principle: the specification and verification of one method should not depend on details that are private to the implementation of another method. This means that modular verification should allow classes (i.e., modules) to be verified independently from each other. An important instance of this principle comes in the presence of aliasing: if two methods share an object, yet their specification is not affected by this sharing, then the specification should not reveal the presence of the sharing.

We introduce the notion of an *object proposition* for the modular verification of object-oriented code in the presence of aliasing. Object propositions combine

predicates on objects with aliasing information about the objects (represented by fractional permissions). They are associated with object references and declared by programmers as part of method pre- and post-conditions. Through the use of object propositions, we are able to hide the shared data that two objects have in common. The implementations of the two objects use fractions to describe how to access the common data, but this common data need not be exposed in their external interface. Our solution is therefore more modular than the state of the art with respect to hiding shared data, and furthermore generalizes systems [3] for which there is good automated tool support.

Our main contributions are the following:

- A verification methodology that unifies substructural logic-based reasoning with invariant-based reasoning. Linear permissions (object propositions where the fraction is equal to 1) permit reasoning similar to separation logic, while fractional permissions (object propositions where the fraction is less than 1) introduce non-linear invariant-based reasoning. Unlike prior work [5], fractions do not restrict mutation of the shared data; instead, they require that the specified invariant be preserved.
- A proof of soundness in support of the system.
- Validation of the approach by specifying and proving partial correctness of the composite pattern, demonstrating benefits in modularity and abstraction compared to other solutions with the same code structure.

2 Overview

Our methodology uses abstract predicates [21] to characterise the state of an object. We embed those predicates in a logical framework, and specifies sharing using *fractions* [5]. A fraction can be equal to 1 or it can be less than 1. Fractions are useful when there are multiple aliases that reference the same object in the program. With the help of fractions, we can track how a reference is allowed to read and modify the referenced object and, in the case of other references to the object, how those references might access the object.

If the fraction is 1, it grants read/write access to some particular fields of the object (depending on the fields that are mentioned in the accompanying predicate). If the fraction is less than 1, there might be other references that also have read/write access to the same object. In the case of fractions less than 1 to be usable, they must guarantee that they will never violate some object invariant, expressed as a predicate.

Our main technical contribution is the novel abstraction called *object proposition*, that combines predicates with aliasing information about objects. For example, to express that the object q in Figure 1 has full control of a queue of integers in range $[0, 10]$, we use the object proposition $q@1\ \mathit{Range}(0, 10)$.

We want our checking approach to be modular and to verify that implementations follow their design intent. In our approach, method pre- and post-conditions are expressed using object propositions over the receiver and arguments of the method. To verify the method, the *abstract* predicate in the object proposition

for the receiver object is interpreted as a *concrete* formula over the current values of the receiver object's fields (including for fields of primitive type *int*). Following Fähndrich and DeLine [10], our verification system maintains a *key* for each field of the receiver object, which is used to track the current values of those fields through the method. A key $o.f \rightarrow x$ represents read/write access to field f of object o holding a value represented by the concrete value x. At the end of a public method, we *pack* [7] the keys back into an object proposition and check that object proposition against the method post-condition.

As a simple example, we consider two linked queues q and r that share a common tail p, in Figure 1. In prior work on separation logic or dynamic frames, the specification of any method has to describe the entire footprint of the method, i.e., all heap locations that are being touched through reading or writing in the body of the method. That is, the shared data p has to be specified in the specification of all methods that access the objects in the lists q and r. Using our object propositions, we have to mention only a permission $q@1\ ValidQueue$ ($r@1$... respectively) in the specification of a method accessing q (or r). The fact that p is shared between the two aliases is hidden by the abstract predicate $ValidQueue$.

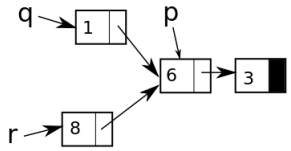

Fig. 1. Linked queues sharing the tail

In Section 4 we discuss this example in more detail in order to illustrate the technical differences in verifying this example using object propositions versus prior approaches.

3 Current Approaches

The verification of object-oriented code can be achieved using the classical invariant-based technique [2]. When using this technique, all objects of the same class have to satisfy the same invariant. The invariant has to hold in all visible states of an object, i.e. before a method is called on the object and after the method returns. The invariant can be broken inside the method as long as it is restored upon exit from the method. This leads to a key limitation: the specifications that can be written using the proof language and the verification that can be performed are limited. This limitation shows itself in a number of ways. One sign is: the methods that can be written for each class are restricted because now each method of a particular class has to have the invariant of that class as a

post-condition. Another sign is that the invariant of an object cannot depend on another object's state, unless additional features such as ownership are added. Leino and Müller [23] have added ownership to organize objects into contexts. In their approach using object invariants, the invariant of an object is allowed to depend on the fields of the object, on the fields of all objects in transitively-owned contexts, and on fields of objects reachable via given sequences of fields. A related restriction is that from outside the object, one cannot make an assertion about that object's state, other than that its invariant holds. Thus the classic technique for checking object invariants ensures that objects remain well-formed, but it does not help with reasoning about how they change over time (other than that they do not break the invariant).

Separation logic approaches [21], [8], [6], etc. bypass the limitations of invariant-based verification techniques by requiring that each method describe its footprint and the predicates that should hold for the objects in that footprint. In this way not all objects of the same class have to satisfy the same predicate. Separation logic allows us to reason about how objects' state changes over time. On the downside, now the specification of a method has to reveal the structures of objects that it uses. This is not a problem if the objects in the footprint are completely encapsulated. But if they are shared between two structures, that sharing must be revealed when transitioning between the "inside" and "outside" of the encapsulating abstraction. This is not desirable from an information hiding point of view.

On the other hand, permission-based work [3], [7], [5] gives another partial solution for the verification of object-oriented code in the presence of aliasing. By using share and/or fractional permissions referring to the multiple aliases of an object, it is possible for objects of the same class to have different invariants. This is different from the traditional thinking that an object invariant is always the same for all objects. What share and/or fractions do is allow us to make different assertions about different objects; we are not limited to a single object invariant. This relaxes the classical invariant-based verification technique and it makes it much more flexible.

Moreover, developers can use *access permissions* [3] to express the design intent of their protocols in annotations on methods and classes. Our work uses fractional permissions [5] (which are similar to access permissions in some respects) in the creation of the novel concept of object propositions. The main difference between the way we use permissions and existing work about permissions is that we do not require the state referred to by a fraction less than 1 to be immutable. Instead, that state has to satisfy an invariant that can be relied on by other objects. Our goal is to modularly check that implementations follow their design intent. The typestate [7] formulation has certain limits of expressiveness: it is only suited to finite state abstractions. This makes it unsuitable for describing fields that contain integers (which can take an infinite number of values) and can satisfy various arithmetical properties. Our object propositions have the advantage that they can express predicates over an infinite domain, such as the integers.

Our fractional permissions system allows verification using a mixture of linear and nonlinear reasoning, combining ideas from previous systems. The existing work on separation logic is an example of linear reasoning, while the work on fractional permissions is an example of nonlinear reasoning. In a linear system there can be only one assertion about each piece of state (such as each field of an object), while in a nonlinear system there can be multiple mentions about the same piece of state inside a formula. The combination of ideas from these two distinct areas allows our system to provide more modularity than each individual approach. For example, in some cases our work can be more modular than separation logic approaches because it can more effectively hide the exact aliasing relationships.

4 Example: Queues of integers

In Figure 2, we present a class that defines object propositions which are useful for reasoning about the correctness of client code and about whether the implementation of a method respects its specification. Our specification logic is based on linear logic[12], a simplification of separation logic that retains the advantages of separation logic's frame rule[1]. Object propositions are thus treated as resources that may not be duplicated, and which are consumed upon usage. Pre- and post-conditions are separated with a linear implication $\multimap$ and use multiplicative conjunction ($\otimes$), additive disjunction ($\oplus$) and existential/universal quantifiers (where there is a need to quantify over the parameters of the predicates).

Newly created objects have a fraction of 1, and their state can be manipulated to satisfy different predicates defined in the class. At the point where the fraction to the object is first split into two fractions less than 1 (see Figure 8), the predicate currently satisfied by the object's state becomes an invariant that the object will always satisfy in future execution. Different references pointing to the same object will always be able to rely on that invariant when calling methods on the object.

A critical part of our work is allowing clients to depend on a property of a shared object. In this queue example, clients depend on the shared *Link* items being in a consistent range. Other methodologies such as Boogie [1] allow a client to depend only on properties of objects that it owns. Our verification technique also allows a client to depend on properties of objects that it doesn't (exclusively) own.

To gain read or write access to the fields of an object, we have to *unpack* it [7]. When we unpack an object proposition (with a fraction of 1 or with a fraction of less than 1), we gain modifying access. After a method finishes working with the fields of a shared object, our proof rules in Section 6 require us to ensure that the same predicate as before the unpacking holds of that shared object. If the same predicate holds, we are allowed to pack back the shared object to

[1] we believe our methodology can be applied in a separation logic setting as well, but leave this extension to future work, as linear logic is sufficient to express our ideas.

```
class Link {
      int val;
      Link next;

  predicate Range(int x, int y) ≡ ∃v, o, k
     val→ v ⊗ next→ o
     ⊗ v ≥ x ⊗ v ≤ y
     ⊗ [o@k Range(x, y) ⊕ o == null]

  predicate UniRange(int x, int y) ≡ ∃v, o
     val→ v ⊗ next→ o ⊗ v ≥ x ⊗ v ≤ y
     ⊗ [o@1 UniRange(x, y) ⊕ o == null]

  void addModulo11(int x)
    this@k Range(0, 10) −∘ this@k Range(0, 10)
    {val = (val + x)% 11;
     if (next!=null) {next.addModulo11(x);}
    }

  void add(int z)
    ∀ x:int, y:int, x<y .this@1 UniRange(x, y)
              −∘ this@1 UniRange(x + z, y + z)
    {val = val + z;
     if (next!=null) {next.add(z);}
    }
 }
```

Fig. 2. Link class and range predicates

that predicate. Since for a unique object (an object that is referenced with a fraction of 1), there is only one reference in the system pointing to it and no risk of interferences, we don't require packing to the same predicate for unique objects. We avoid inconsistencies by allowing multiple object propositions to be unpacked at the same time only if the objects are not aliased, or if the unpacked propositions cover disjoint fields of a single object.

The predicate *Range(int x, int y)* in Figure 2 ensures that all the elements in a linked queue starting from the current Link are in the range $[x, y]$. The object proposition mentions $o@k$, thus requiring the existence of a fraction giving access to each Link of the queue. In contrast, the predicate *UniRange(int x, int y)* requires the presence of a unique permission to all elements in the queue. These restrictions mean that the only method that can be called on a shared Link object satisfying invariant $Range(0, 10)$ is *addModulo11*. The specification of the *addModulo11(int x)* method is the only one that does not break the invariant. If the programmer wants to modify some value in the queue using the *add(int z)* method, the queue must be accessed through a fraction of 1.

Given a pre-condition and a desired post-condition, expressed as logical formulas over object propositions, the proof rules in Section 6 are used to verify a segment of code. There is a proof rule for each expression form, as well as a rule for sequencing statements/expressions (technically with a let expression in our formalism). The soundness of the proof rules (defined later) means that given a heap that satisfies the pre-condition formula, a program that typechecks and verifies according to our proof rules will execute, and if it terminates, will result in a heap that satisfies the postcondition formula. While our proof rules are not complete, they are sufficient to verify an interesting set of programs and properties, including the examples in this paper.

We show some client code and its verification using object propositions:

```
Link la = new (Link(3, null),Range(0,10));
```
$unpacked(la, 1) \otimes \exists v, o\ .\ la.val \to v \otimes la.next \to o$
$\otimes\ v == 3 \otimes o == null$
$la@1\ Range(0, 10)$

$la@\frac{1}{2}\ Range(0, 10) \otimes la@\frac{1}{2}\ Range(0, 10)$
```
Link p = new (Link(6, la),Range(0,10));
```
$p@1\ Range(0, 10) \otimes la@\frac{1}{2}\ Range(0, 10)$

$p@\frac{1}{2}\ Range(0, 10) \otimes p@\frac{1}{2}\ Range(0, 10)$
```
Link q = new (Link(1, p),Range(0,10));
```
$q@1\ Range(0, 10) \otimes p@\frac{1}{2}\ Range(0, 10)$
```
Link r = new (Link(8, p),Range(0,10));
```
$r@1\ Range(0, 10) \otimes q@1\ Range(0, 10)$

$r@\frac{1}{2}\ Range(0, 10) \otimes r@\frac{1}{2}\ Range(0, 10) \otimes q@1\ Range(0, 10)$
```
r.addModulo11(9);
```
$q@1\ Range(0, 10) \otimes r@\frac{1}{2} Range(0, 10)$

$q@\frac{1}{2}\ Range(0,10)\ \otimes\ q@\frac{1}{2}\ Range(0,10) \otimes\ r@\frac{1}{2}\ Range(0,10)$

```
q.addModulo11(7);
```

$q@\frac{1}{2}\ Range(0,10)\ \otimes\ r@\frac{1}{2}\ Range(0,10)$

When first creating object la, we get an unpacked object proposition with a fraction of 1, along with keys for each of the fields and assertions about the field values. We can pack these permissions up into the predicate $Range(0,10)$, then split the fraction of 1 into two half fractions, as we have to pass a permission to la to the constructor of *Link* when creating p. Since la can be accessed through p from now on, we do not need to mention la in the following object proposition sets. In this respect, our system is an affine one (as opposed to a linear one), because we can drop a resource if we do not need it any more.

When creating the object q, we need to pass in a fraction to p. We obtain it by splitting the current fraction of 1 to p into two fractions. Before calling method *addModulo11* on r, we also have to split the fraction to r so that a fraction can be passed to the method. The splitting of fractions is similar when we call *addModulo11* on q.

The specification in separation logic is more cumbersome and unable to hide shared data. To express the fact that all values in a segment of linked elements are in the interval $[n_1, n_2]$, we need to define the following predicate :

$$Listseg(r,p,n_1,n_2) \equiv (r = p) \vee (r \rightarrow (i,s)\star \\ Listseg(s,p,n_1,n_2) \wedge n_1 \leq i \leq n_2).$$

This predicate states that either the segment is null, or the *val* field of r points to i and the *next* field points to s, such that $n_1 \leq i \leq n_2$, and the elements on the segment from s to p are in the interval $[n_1, n_2]$. The verification of the same code in separation logic is shown below:

```
...
```
$\{\}$
```
Link la = new Link(3, null);
```
$\{Listseg(la, null, 0, 10)\}$
```
Link p = new Link(6, la);
```
$\{Listseg(p, null, 0, 10)\}$
```
Link q = new Link(1, p);
```
$\{Listseg(q,p,0,10) \star Listseg(p, null, 0, 10)\}$
```
Link r = new Link(8, p);
```
$\{Listseg(q,p,0,10) \star Listseg(r,p,0,10)$
$\star\ Listseg(p, null, 0, 10)\}$

$\{Listseg(q,p,0,10) \star Listseg(r, null, 0, 10)\}$
```
r.addModulo11(9);
```
$\{Listseg(q,p,0,10) \star Listseg(r, null, 0, 10)\}$
```
  *****missing step*****
```
$\{Listseg(q, null, 0, 10) \star Listseg(r,p,0,10)\}$
```
q.addModule11(7);
```
$\{Listseg(q, null, 0, 10) \star Listseg(r,p,0,10)\}$
```
...
```

In separation logic, the natural pre- and post-conditions of the method *addModulo11* are $Listseg(this, null, 0, 10)$, i.e., the method takes in a list of elements in $[0, 10]$ and returns a list in the same range.

Thus, before calling `addModulo11` on r, we have to combine $Listseg(r, p, 0, 10) \star Listseg(p, null, 0, 10)$ into $Listseg(r, null, 0, 10)$. We observe the following problem: in order to call `addModulo11` on q, we have to take out $Listseg(p, null, 0, 10)$ and combine it with $Listseg(q, p, 0, 10)$, to obtain $Listseg(q, null, 0, 10)$. But the specification of the method does not allow it, hence the missing step in the verification above. The specification of `addModulo11` has to be modified instead, by mentioning that there exists some sublist $Listseg(p, null, 0, 10)$ that we pass in and which gets passed back out again. The modification is unnatural: the specification of `addModulo11` should not care that it receives a list made of two separate sublists, it should only care that it receives a list in range $[0, 10]$.

This situation is very problematic because the specification of `addModulo11` involving sublists becomes awkward. We can imagine an even more complicated example, where there are three sublists that we need to pass in and out of `addModulo11`. It is impossible to know, at the time when we write the specification of a method, on what kind of shared data that method will be used. Separation logic approaches will thus have a difficult time trying to verify this kind of code, while object propositions allow us to call methods on both lists, without requiring the combination of two predicates.

4.1 Example: Cells in a spreadsheet

I consider the example of a spreadsheet, as described in [17]. In my spreadsheet each cell contains an add formula that adds two integer inputs. Each cell may refer to other two cells. The general case would be for each cell to have a dependency list of cells, but since my grammar does not support arrays yet, I am not considering that case. Whenever the user changes a cell, each of the two cells which transitively depend upon it must be updated.

A visual representation of this example is presented in Figure 3. In separation logic, the specification of any method has to describe the entire footprint of the method, i.e., all heap locations that are being touched through reading or writing in the body of the method. That is, the shared cells $a3$ and $a6$ have to be specified in the specification of all methods that modify the cells $a1$ and $a2$.

In Figure 4, I present the code implementing a cell in a spreadsheet.

The specification in separation logic is unable to hide shared data. To express the fact that all cells are in a consistent state where the dependencies are respected and the sum of the inputs is equal to the output for each cell, I define the following predicate :

$SepOK(cell) \equiv (cell.in1 \rightarrow x1) \star (cell.in2 \rightarrow x2) \star (cell.out \rightarrow o) \star (cell.dep1 \rightarrow d1) \star (cell.dep2 \rightarrow d2) \star (x1 + x2 = o) \star (SepOK(d1.ce) \wedge d1.ce.\text{“}in + input\text{”} \rightarrow o) \star (SepOK(d2.ce) \wedge ((d2.ce.in1 \rightarrow o \wedge d2.input = 1) \vee (d2.ce.in2 \rightarrow o \wedge d2.input = 2)))$.

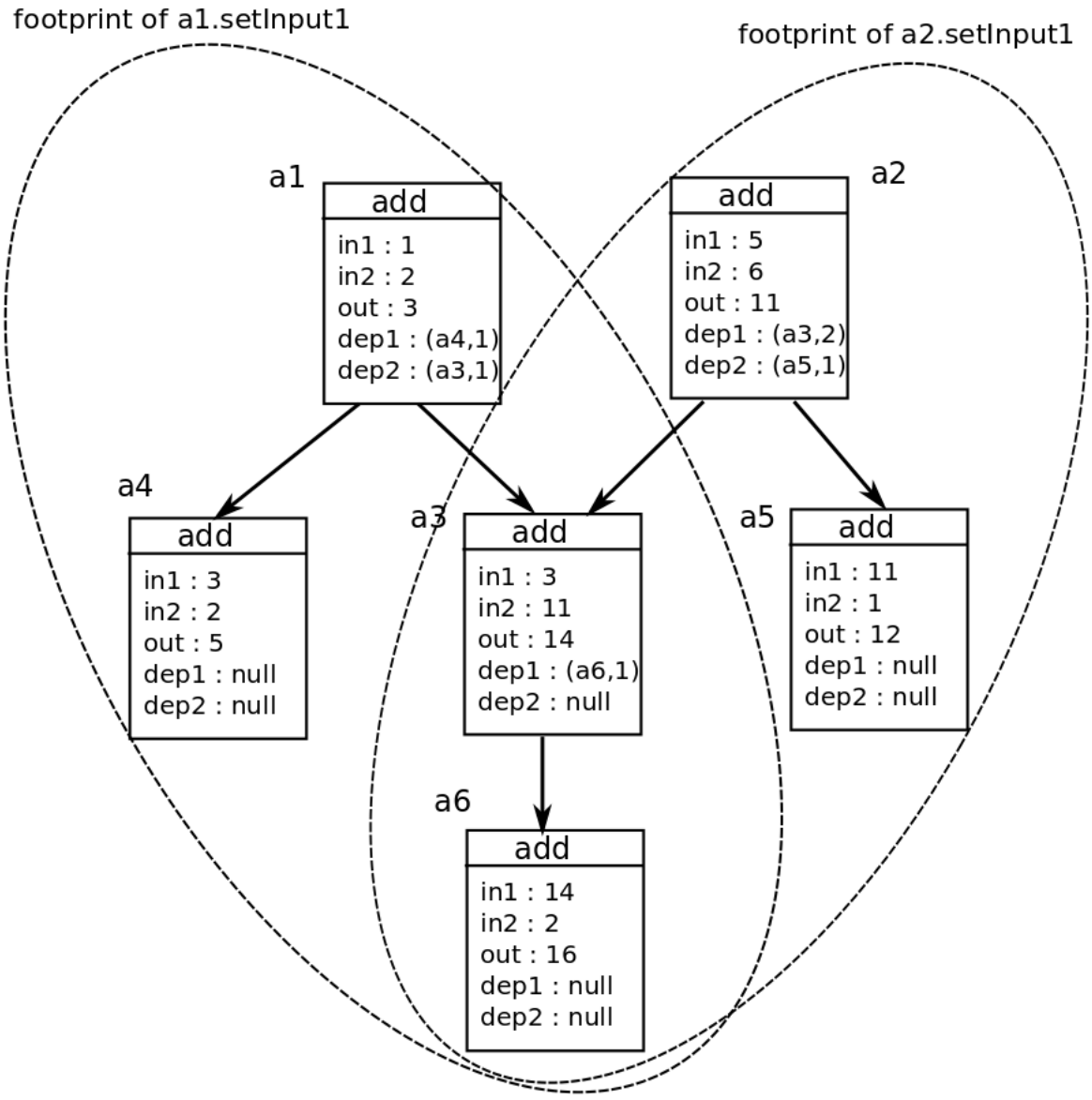

Fig. 3. Add cells in spreadsheet

```
class Dependency {
      Cell ce;
      int input;
 }
class Cell {
      int in1, in2, out;
      Dependency dep1, dep2;

  void setInputDep(int newInput) {
    if (dep1!=null) {
          if (dep1.input == 1) dep1.ce.setInput1(newInput);
          else dep1.ce.setInput2(newInput);
    }
    if (dep2!=null) {
          if (dep2.input == 1) dep2.ce.setInput1(newInput);
          else dep2.ce.setInput2(newInput);
    }
    }

  void setInput1(int x) {
    this.in1 = x;
    this.out = this.in1 + this.in2;
    this.setInputDep(out);
    }

  void setInput2(int x) {
    this.in2 = x;
    this.out = this.in1 + this.in2;
    this.setInputDep(out);
    }

 }
```

Fig. 4. Cell class

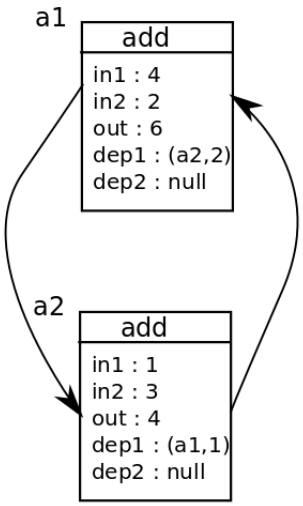

Fig. 5. Cells in a cycle

This predicate states that the sum of the two inputs of *cell* is equal to the output, and that the predicate $SepOK$ is verified by all the cells that directly depend on the output of the current cell. Additionally, the predicate $SepOK$ also checks that the corresponding input for each of the two dependency cells is equal to the output of the current cell. This predicate only works in the case when the cells form a directed acyclic graph (DAG). The predicate $SepOK$ causes problems when there is a diamond structure (not shown in Figure 3) or if one wants to assert the predicate about two separate nodes whose subtrees overlap due to a DAG structure (e.g. a1 and a2 in Figure 3). If the dependencies between the cells form a cycle, as in Figure 5, the predicate $SepOK$ cannot possibly hold.

Additionally we need another predicate to express simple properties about the cells:

$Basic(cell) \equiv \exists x1, x2, o, d.(cell.in1 \rightarrow x1) \star (cell.in2 \rightarrow x2) \star (cell.out \rightarrow o) \star (cell.dep \rightarrow d)$.

Below I show a fragment of client code and its verification using separation logic.

$\{Basic(a2) \star Basic(a5) \star SepOK(a1)\}$
```
a1.setInput1(10);
```
$\{Basic(a2) \star Basic(a5) \star SepOK(a1)\}$
$\{******** missing\ step ********\}$
$\{Basic(a4) \star Basic(a1) \star SepOK(a2)\}$
```
a2.setInput1(20);
```

In the specification above,

$$SepOK(a1) \equiv a1.in1 \to x1 \star a1.in2 \to x2 \star a1.out \to o \star x1 + x2 = o \star (SepOK(a4) \wedge a4.in1 = o) \star (SepOK(a3) \wedge a3.in1 = o)$$

and

$$SepOK(a2) \equiv a2.in1 \to z1 \star a2.in2 \to z2 \star a2.out \to p \star z1 + z2 = p \star (SepOK(a3) \wedge a3.in2 = p) \star (SepOK(a5) \wedge a5.in1 = p)$$

In separation logic, the natural pre- and post-conditions of the method *setInput1* are $SepOK(this)$, i.e., the method takes in a cell that is in a consistent state in the spreadsheet and returns a cell with the input changed, but that is still in a consistent state in the spreadsheet. Note that the pre-condition does not need to be of the form $SepOK(this, carrier)$ where *carrier* is all the cells involved as in Jacobs *et al.*'s work [15]. This is because $SepOK$ is a recursive abstract predicate that states in its definition properties about the cells that depend on the current *this* cell and thus we do not need to explicitly carry around all the cells involved. The natural specification of *setInput1* would be $SepOK(this) \Rightarrow SepOK(this)$.

Thus, before calling `setInput1` on $a2$, we have to combine $SepOK(a3) \star SepOK(a5)$ into $SepOK(a2)$. We observe the following problem: in order to call `setInput1` on $a2$, we have to take out $SepOK(a3)$ and combine it with $SepOK(a5)$, to obtain $SepOK(a2)$. But the specification of the method does not allow it, hence the missing step in the verification above. The specification of `setInput1` has to be modified instead, by mentioning that there exists some cell $a3$ that satisfies $SepOK(a3)$ that we pass in and which gets passed back out again. Thus, if we want to call `setInput1` on $a2$, the specification of `setInput1` would have to know about the specific cell $a3$, which is not possible.

The specification of *setInput1* would become

$\forall \alpha, \beta, x \;.\; (SepOK(this) \wedge SepOK(this) \equiv \alpha \star SepOK(x) \star \beta)$

$\Rightarrow (SepOK(this) \wedge SepOK(this) \equiv \alpha \star SepOK(x) \star \beta)$.

The modification is unnatural: the specification of `setInput1` should not care about which are the dependencies of the current cell, it should only care that it modified the current cell.

This situation is very problematic because the specification of `setInput1` involving shared cells becomes awkward. One can imagine an even more complicated example, where there are multiple shared cells that need to be passed in and out of different calls to `setInput1`. It is impossible to know, at the time when we write the specification of a method, on what kind of shared data that method will be used.

In Figure 6, we present the Java class from Figure 4 augmented with predicates and object propositions, which are useful for reasoning about the correctness of client code and about whether the implementation of a method respects its specification. Since they contain fractional permissions which represent resources that have to be consumed upon usage, the object propositions are consumed upon usage and their duplication is forbidden. Therefore, we use linear logic [12] to write the specifications. Pre- and post-conditions are separated with a linear implication $\multimap$ and use multiplicative conjunction ($\otimes$), additive disjunc-

tion ($\oplus$) and existential/universal quantifiers (where there is a need to quantify over the parameters of the predicates).

Newly created objects have a fractional permission of 1, and their state can be manipulated to satisfy different predicates defined in the class. A fractional permission of 1 can be split into two fractional permissions which are less than 1, see Figure 8. The programmer can specify an invariant that the object will always satisfy in future execution. Different references pointing to the same object, will always be able to rely on that invariant when calling methods on the object.

A critical part of our work is allowing clients to depend on a property of a shared object. Other methodologies such as Boogie [1] allow a client to depend only on properties of objects that it owns. Our verification technique also allows a client to depend on properties of objects that it doesn't (exclusively) own.

To gain read or write access to the fields of an object, we have to *unpack* it [7]. After a method finishes working with the fields of a shared object (an object for which we have a fractional permission, with a fraction less than 1), our proof rules in Section 6 require us to ensure that the same predicate as before the unpacking holds of that shared object. If the same predicate holds, we are allowed to pack back the shared object to that predicate. Since for an object with a fractional permission of 1 there is no risk of interferences, we don't require packing to the same predicate for this kind of objects. We avoid inconsistencies by allowing multiple object propositions to be unpacked at the same time only if the objects are not aliased, or if the unpacked propositions cover disjoint fields of a single object.

Packing/unpacking [7] is a very important mechanism in our system. The benefits of this mechanism are the following:

- it achieves information hiding (e.g. like abstract predicates)
- it describes the valid states of the system (similar to visible states in invariant-based approaches)
- it is a way to store resources in the heap. When a field key is put (packed) into a predicate, it disappears and cannot be accessed again until it is unpacked
- it allows us to characterize the correctness of the system in a simple way when everything is packed

Another central idea of our system is *sharing* using fractions less than 1. The insights about sharing are the following:

- with a fractional permission of 1, no sharing is permitted. There is only one of each abstract predicate asserted for each object at run time, and the asserted abstract predicates have disjoint fields.
- fractional permissions less than 1 enable sharing of particular abstract predicates, but only one instance of a particular abstract predicate P on a particular object o can be unpacked at once. This ensures that field permissions cannot be duplicated via shared permissions.

An important aspect of our system is the ability to allow predicates to depend on each other. Intuitively, this allows "chopping up" an invariant into its modular constituent parts.

```
class Dependency {
      Cell ce;
      int input;

  predicate OKdep(int o) ≡ ∃c, k, i.this.ce → c ⊗ this.input → i⊗
     ((i = 1 ⊗ c@½ In1(o) ⊗ c@k OK()) ⊕ (i = 2 ⊗ c@½ In2(o) ⊗ c@k OK()))
 }
class Cell {
      int in1, in2, out;
      Dependency dep1, dep2;

  predicate In1(int x1) ≡ this.in1 → x1

  predicate In2(int x2) ≡ this.in2 → x2

  predicate OK() ≡ ∃x1, x2, o, d1, d2.this@½ In1(x1) ⊗ this@½ In2(x2)⊗
      x1 + x2 = o ⊗ this.out → o ⊗ this.dep1 → d1 ⊗ this.dep2 → d2⊗
     d1@1 OKdep(o) ⊗ d2@1 OKdep(o)

  void setInputDep(int i, int newInput) {
    if (dep1!=null) {
          if (dep1.input == 1) dep1.ce.setInput1(newInput);
          else dep1.ce.setInput2(newInput);
    }
    if (dep2!=null) {
          if (dep2.input == 1) dep2.ce.setInput1(newInput);
          else dep2.ce.setInput2(newInput);
    }
    }

  void setInput1(int x)
  ∀k.(this@k OK() ⊸ this@k OK())
  { this.in1 = x;
    this.out = this.in1 + this.in2;
    this.setInputDep(out);
  }

  void setInput2(int x)
  ∀k.(this@k OK() ⊸ this@k OK())
  { this.in2 = x;
    this.out = this.in1 + this.in2;
    this.setInputDep(out);
  }

 }
```

Fig. 6. Cell class and OK predicate

Like other previous systems, our system uses abstraction, which allows clients to treat method pre/post-conditions opaquely.

The predicate *OK()* in Figure 6 ensures that all the cells in the spreadsheet are in a consistent state, where the sum of their inputs is equal to their output. Since we only use a fractional permissions $k1, k2 < 1$ for the dependency cells, it is possible for multiple predicates $OK()$ to talk about the same cell without exposing the sharing. More specifically, using object propositions we only need to know $a1@k\ OK()$ before calling $a1.setInput1(10)$. Before calling $a2.setInput1(20)$ we only need to know $a2@k\ OK()$. Since inside the recursive predicate $OK()$ there are fractional permissions less than 1 that refer to the dependency cells, we are allowed to share the cell $a3$ (which can depend on multiple cells). Thus, using object propositions we are not explicitly revealing the shared cells in the structure of the spreadsheet.

5 Grammar

The programming language that we are using is inspired by Featherweight Java [13], extended to include object propositions. We retained only Java concepts relevant to the core technical contribution of this paper, omiting features such as inheritance, casting or dynamic dispatch that are important but are handled by orthogonal techniques.

Below we show the syntax of our simple class-based object-oriented language. In addition to the usual constructs, each class can define one or more abstract predicates Q in terms of concrete formulas R. Each method comes with pre and post-condition formulas. Formulas include object propositions P, terms, primitive binary predicates, conjunction, disjunction, keys, and quantification. We distinguish effectful expressions from simple terms, and assume the program is in let-normal form. The pack and unpack expression forms are markers for when packing and unpacking occurs in the proof system. References o and indirect references l do not appear in source programs but are used in the dynamic semantics, defined later. In the grammar, r represents a reference to an object and i represents a reference to an integer.

In our system, we will assume that all the formulas R are in disjunctive normal form. A formula R of our system is in disjunctive normal form if and only if it is an additive disjunction of one or more multiplicative conjunctions of one or more of the predicates P , t_1 `binop` t_2 , $r.f \to x$, $\exists z.\mathsf{P}$, $\forall z.\mathsf{P}$.

5.1 Permission Splitting

In order to allow objects to be aliased, we must split a fraction of 1 into multiple fractions less than 1 [5]. The fraction splitting rule is defined in Figure 8. An invariant of the rules is that a fraction of 1 is never duplicated. We also allow the inverse of splitting permissions: joining, where we define the rules in Figure 9.

$$
\begin{array}{rcl}
\mathsf{Prog} & ::= & \overline{\mathsf{ClDecl}}\ \mathtt{e} \\
\mathsf{ClDecl} & ::= & \mathtt{class}\ C\ \{\ \overline{\mathsf{FldDecl}}\ \overline{\mathsf{PredDecl}}\ \overline{\mathsf{MthDecl}}\ \} \\
\mathsf{FldDecl} & ::= & \mathsf{T}\ f \\
\mathsf{PredDecl} & ::= & \mathtt{predicate}\ Q(\overline{\mathsf{T}\ \mathtt{x}})\ \equiv\ \mathsf{R} \\
\mathsf{MthDecl} & ::= & \mathsf{T}\ m(\overline{\mathsf{T}\ \mathtt{x}})\ \mathsf{MthSpec}\ \{\ \overline{\mathtt{e}};\ \mathtt{return}\ \mathtt{e}\ \} \\
\mathsf{MthSpec} & ::= & \mathsf{R} \multimap \mathsf{R} \\
\mathsf{R} & ::= & \mathsf{P}\ \mid\ \mathsf{R} \otimes \mathsf{R}\ \mid\ \mathsf{R} \oplus \mathsf{R}\ \mid \\
 & & \exists \overline{z}.\mathsf{R}\ \mid\ \forall \overline{z}.\mathsf{R}\ \mid\ r.f \to \mathtt{x}\ \mid\ \mathtt{t}\ \mathtt{binop}\ \mathtt{t} \\
\mathsf{P} & ::= & r@\mathtt{k}\ Q(\overline{\mathtt{t}})\ \mid\ \mathtt{unpacked}(r@\mathtt{k}\ Q(\overline{\mathtt{t}})) \\
\mathtt{k} & ::= & \frac{n_1}{n_2}\ (\text{where } n_1, n_2 \in \mathbb{N} \text{ and } 0 < n_1 \leq n_2) \\
\mathtt{e} & ::= & \mathtt{t}\ \mid\ r.f\ \mid\ r.f\ \mathtt{=}\ \mathtt{t}\ \mid\ r.m(\overline{\mathtt{t}})\ \mid\ \mathtt{new}\, C(\overline{\mathtt{t}})\ \mid \\
 & & \mathtt{if}\ (\mathtt{t})\ \{\ \mathtt{e}\ \}\ \mathtt{else}\ \{\ \mathtt{e}\ \}\ \mid \\
 & & \mathtt{let}\ \mathtt{x}\ \mathtt{=}\ \mathtt{e}\ \mathtt{in}\ \mathtt{e}\ \mid \\
 & & \mathtt{t}\ \mathtt{binop}\ \mathtt{t}\ \mid\ \mathtt{t}\ \mathtt{\&\&}\ \mathtt{t}\ \mid\ \mathtt{t} \parallel \mathtt{t}\ \mid\ \mathtt{!}\ \mathtt{t}\ \mid \\
 & & \mathtt{pack}\ r@\mathtt{k}\ Q(\overline{\mathtt{t}})\,\mathtt{in}\ \mathtt{e}\ \mid \\
 & & \mathtt{unpack}\ r@\mathtt{k}\ Q(\overline{\mathtt{t}})\,\mathtt{in}\ \mathtt{e} \\
\mathtt{t} & ::= & \mathtt{x}\ \mid\ n\ \mid\ \mathtt{null}\ \mid\ \mathtt{true}\ \mid\ \mathtt{false} \\
\mathtt{x} & ::= & r\ \mid\ i \\
\mathtt{binop} & ::= & +\mid -\mid \%\mid =\mid !=\mid \leq\mid <\mid \geq\mid > \\
\mathsf{T} & ::= & C\ \mid\ \mathtt{int}\ \mid\ \mathtt{boolean}
\end{array}
$$

Fig. 7. Grammar of language and specification

6 Proof Rules

This section describes the proof rules that can be used to verify correctness properties of code. The judgement to check an expression e is of the form $\Gamma; \Pi \vdash e : \exists x.T; R$. This is read "in valid context Γ and linear context Π, an expression e executed has type T with postcondition formula R".This judgement is within a receiver class C, which is mentioned when necessary in the assumptions of the rules. By writing $\exists x$, we bind the variable x to the result of the expression e in the postcondition. Γ gives the types of variables and references, while Π is a pre-condition in disjunctive normal form. The linear context Π should be just as general as R.

$$
\begin{array}{rcl}
\textit{type context}\ \Gamma & ::= & \cdot \mid \Gamma, x : T \\
\textit{linear context}\ \Pi & ::= & \bigoplus_{i=1}^{n} \Pi_i \\
\Pi_i & ::= & \cdot \mid \Pi_i \otimes \mathsf{P} \mid \Pi_i \otimes t_1\ \mathtt{binop}\ t_2 \mid \\
 & & \Pi_i \otimes r.f \to x \mid \exists \overline{z}.\mathsf{P} \mid \forall \overline{z}.\mathsf{P}
\end{array}
$$

The static proof rules also contain the following judgements: $\Gamma \vdash r : C$, $\Gamma; \Pi \vdash R$ and $\Gamma; \Pi \vdash r.T; R$. The judgement $\Gamma \vdash r : C$ means that in valid type context Γ, the reference r has type C. The judgement $\Gamma; \Pi \vdash R$ means that from valid type context Γ and linear context Π we can deduce that object proposition R holds. The judgement $\Gamma; \Pi \vdash r.T; R$ means that from valid type context Γ and linear context Π we can deduce that reference r has type T and object proposition R is true about r. The $\otimes$ linear logic operator is symmetric. Thus in the rules for adding fractions, we can have a rule symmetric to (ADD2) that

adds the fraction of a packed object propositions to the fraction of an unpacked object proposition.

Before presenting the detailed rules, I provide the intuition for why my system is sound (the formal soundness theorem is given below in Section9.1). The first invariant enforced by my system is that there will never be two conflicting object propositions to the same object. The fraction splitting rule can give rise to only one of two situations, for a particular object: there exists a reference to the object with a fraction of 1, or all the references to this object have fractions less than 1. For the first case, sound reasoning is easy because aliasing is prohibited.

$$\frac{k \in (0,1]}{r@k\ Q(\bar{t}) \vdash\ r@\frac{k}{2}\ Q(\bar{t}) \otimes r@\frac{k}{2}\ Q(\bar{t})}\ (\textsc{Split})$$

Fig. 8. Rule for splitting fractions

$$\frac{\epsilon \in (0,1) \qquad k \in (0,1] \qquad \epsilon < k}{r@\epsilon\ Q(\overline{t_1}) \otimes r@(k-\epsilon)\ Q(\overline{t_1}) \vdash\ r@k\ Q(\overline{t_1})}\ (\textsc{Add1})$$

$$\frac{\epsilon \in (0,1) \qquad k \in (0,1] \qquad \epsilon < k}{\begin{array}{c} unpacked(r@\epsilon\ Q(\overline{t_1})) \otimes r@(k-\epsilon)\ Q(\overline{t_1}) \vdash \\ unpacked(r@k\ Q(\overline{t_1})) \end{array}}\ (\textsc{Add2})$$

Fig. 9. Rules for adding fractions

The second case, concerning fractional permissions less than 1, follows an inductive argument in nature. The argument is based on the property that the invariant of a shared object (one can think of an object with a fraction less than 1 as being shared) always holds whenever that object is packed. The base case in the induction occurs when an object with a fraction of 1, whose invariant holds, first becomes shared. In order to access the fields of an object, we must first unpack the corresponding predicate; by induction, we can assume its invariant holds as long as the object is packed. But we know the object is packed immediately before the unpack operation, because the rules of my system ensure that a given predicate over a particular object can only be unpacked once; therefore, we know the object's invariant holds. Assignments to the object's fields may later violate the invariant, but in order to pack the object back up we must restore its invariant. For a shared object, packing must restore the same predicate the object had when it was unpacked; thus the invariant of an object never changes once that object is shared, avoiding inconsistencies between aliases to the object. (Note that if at a later time we add the fractions corresponding to that object

and get a fraction of 1, we will be able to change the predicates that hold of that object. But as long as the object is shared, the invariant of that object must hold.) Although theoretically an object may have several different invariants, in all my examples in my thesis proposal each object has only one invariant. In future work I would like to support multiple invariants for the same object, but this thesis does not deal with this case.

This completes the inductive case for soundness of shared objects. The induction is done on the steps when a predicate is packed or unpacked. All of the predicates we might infer will thus be sound because we will never assume anything more about that object than the predicate invariant, which should hold according to the above argument.

In the following paragraphs, we describe the proof rules while inlining the rules in the text. In the rules below we assume that there is a class C that is the same for all the rules.

The rule TERM below formalizes the standard logical judgement for existential introduction. The notation $[e'/x]e$ substitutes e' for occurrences of x in e. The FIELD rule checks field accesses analogously.

$$\frac{\Gamma \vdash t : T \quad \Gamma; \Pi \vdash [t/x]R}{\Gamma; \Pi \vdash t : \exists x.T; R} \text{ TERM}$$

$$\frac{\begin{array}{c}\Gamma \vdash r : C \quad r.f_i : T \textit{ is a field of } C \\ \Pi \vdash r.f_i \to r_i \qquad \Gamma; \Pi \vdash [r_i/x]R\end{array}}{\Gamma; \Pi \vdash r.f_i : \exists x.T; R} \text{ FIELD}$$

NEW checks object construction. We get a key for each field and the remaining linear context Π_1. The context Π_1 contains the object propositions taken from Π from which we consumed the keys.

$$\frac{\textit{fields}(C) = \overline{T\ f} \quad \Gamma \vdash \overline{t : T}}{\Gamma; \Pi \vdash \texttt{new}\ C(\overline{t}) : \exists z.C; z.\overline{f} \to \overline{t} \otimes \Pi_1} \text{ NEW}$$

IF introduces disjunctive types in the system and checks if-expressions. A corresponding $\oplus$ rule eliminates disjunctions in the pre-condition by verifying that an expression checks under either disjunct.

$$\frac{\begin{array}{c}\Gamma; (\Pi \otimes t = \texttt{true}) \vdash e_1 : \exists x.T; R_1 \\ \Gamma \vdash t : \texttt{bool} \quad \Gamma; (\Pi \otimes t = \texttt{false}) \vdash e_2 : \exists x.T; R_2\end{array}}{\Gamma; \Pi \vdash \ \texttt{if(t)\{}e_1\texttt{\}else\{}e_2\texttt{\}} : \exists x.T; R_1 \oplus R_2} \text{ IF}$$

LET checks a *let* binding, extracting existentially bound variables and putting them into the context (a limitation of my current system is that universal quantification is supported only in method specifications).

$$\frac{\Gamma; \Pi \vdash e_1 : \exists x.T_1; R_1 \otimes \Pi_2 \quad (\Gamma, x : T_1); (R_1 \otimes \Pi_2) \vdash e_2 : \exists w.T_2; R_2}{\Gamma; \Pi \vdash \texttt{let } x = e_1 \texttt{ in } e_2 : \exists w.T_2; R_2} \text{ Let}$$

$$\frac{\Gamma; \Pi_1 \vdash e : \exists x.T; R_1 \qquad \Gamma; \Pi_2 \vdash e : \exists x.T; R_2}{\Gamma; (\Pi_1 \bigoplus \Pi_2) \vdash e : \exists x.T; R_1 \oplus R_2} \oplus$$

The Call rule simply states what is the object proposition that holds about the result of the method being called. This rule first identifies the specification of the method (using the helper judgement Mtype) and then goes on to state the object proposition holding for the result. The $\vdash$ notation in the fourth premise of the Call rule represents entailment in linear logic.

The reader might see that there are some concerns about the modularity of the CALL rule: Π_1 shouldn't contain unpacked predicates. Indeed, it is important that the CALL rule tracks all shared predicates that are unpacked. It does not track predicates that are packed, nor unpacked predicates that have a fractional permission of 1. The normal situation is that all shared predicates are packed, and any method can be called in this situation. In the intended mode of use, we only make calls with a shared unpacked predicate when traversing a data structure hand-over-hand as in the Composite pattern, and we claim that modularity problems are minimized in this situation. This does represent a limitation in our system, however, it is one that goes hand in hand with the advantage of supporting shared predicates.

$$\frac{\begin{array}{c}\Gamma \vdash r_0 : C_0 \quad \Gamma \vdash \overline{t_1 : T} \\ \Gamma; \Pi \vdash [r_0/this][\overline{t_1}/\overline{x}]R_1 \otimes \Pi_1 \\ mtype(m, C_0) = \forall \overline{x : T}.\exists result.T_r; R_1' \multimap R \\ R_1 \vdash R_1' \\ \Pi_1 \textit{ cannot contain unpacked predicates}\end{array}}{\Gamma; \Pi \vdash r_0.m(\overline{t_1}) : \exists\ result.T_r; [r_0/this][\overline{t_1}/\overline{x}]R \otimes \Pi_1} \text{ Call}$$

$$\frac{\begin{array}{c}class\ C\{...\overline{M}...\} \in \overline{CL} \\ T_r\ m(\overline{Tx})R_1 \multimap R\ \{\overline{e_1};\ return\ e_2\} \in \overline{M}\end{array}}{mtype(m, C) = \forall \overline{x : T}.\exists result.T_r; R_1 \multimap R} \text{ Mtype}$$

The rule Assign assigns an object t to a field f_i and returns the old field value as an existential x. For this rule to work, the current object *this* has to be unpacked, thus giving us permission to modify the fields.

$$\frac{\begin{array}{c}\Gamma; \Pi \vdash t_1 : T_i; t_1@k_0\ Q_0(\overline{t_0}) \otimes \Pi_1 \\ \Gamma; \Pi_1 \vdash r_1.f_i : T_i; r_i'@k'\ Q'(\overline{t'}) \otimes \Pi_2 \\ \Pi_2 \vdash r_1.f_i \to r_i' \otimes \Pi_3\end{array}}{\begin{array}{l}\Gamma; \Pi \vdash r_1.f_i = t_1\ : \exists x.T_i; x@k'\ Q'(\overline{t'}) \otimes t_1@k_0\ Q_0(\overline{t_0}) \\ \qquad \otimes\ r_1.f_i \to t_1 \otimes \Pi_3\end{array}} \text{ Assign}$$

The rules for packing and unpacking are PACK1, PACK2, UNPACK1 and UNPACK2. As mentioned before, when we pack an object to a predicate with a fraction less than 1, we have to pack it to the same predicate that was true before the object was unpacked. The restriction is not necessary for a predicate with a fraction of 1: objects that are packed to this kind of predicate can be packed to a different predicate than the one that was true for them before unpacking. For example, in Figure 6, the method *setInput*1 has as pre-condition the object proposition $this@k\ OK()$. Since the fraction k is universally quantified and can be less than 1, the *this* object in the post-condition must be sure to satisfy the predicate $OK()$ (which happens to also be an invariant in this example).

$$\frac{\begin{array}{c}\Gamma; \Pi \vdash r : C; [\overline{t_2}/\overline{x}]R_2 \otimes \Pi_1 \\ \texttt{predicate}\ Q_2(\overline{Tx}) \equiv R_2 \in C \\ \Gamma; (\Pi_1 \otimes r@1\ Q_2(\overline{t_2})) \vdash e : \exists x.T; R\end{array}}{\Gamma; \Pi \vdash \texttt{pack}\ r@1\ Q_2(\overline{t_2})\ \texttt{in}\ e : \exists x.T; R}\ \textsc{Pack1}$$

$$\frac{\begin{array}{c}\Gamma; \Pi \vdash r : C; [\overline{t_1}/\overline{x}]R_1 \otimes \mathsf{unpacked}(r@k\ Q(\overline{t_1})) \otimes \Pi_1 \\ \texttt{predicate}\ Q(\overline{Tx}) \equiv R_1 \in C \qquad 0 < k < 1 \\ \Gamma; (\Pi_1 \otimes r@k\ Q(\overline{t_1})) \vdash e : \exists x.T; R\end{array}}{\Gamma; \Pi \vdash \texttt{pack}\ r@k\ Q(\overline{t_1})\ \texttt{in}\ e : \exists x.T; R}\ \textsc{Pack2}$$

As mentioned earlier, we allow unpacking of multiple predicates, as long as the objects don't alias. We also allow unpacking of multiple predicates of the same object, because we have a single linear write permission to each field. There can't be any two packed predicates containing write permissions to the same field.

$$\frac{\begin{array}{c}\Gamma; \Pi \vdash r : C; r@1\ Q(\overline{t_1}) \otimes \Pi_1 \\ \texttt{predicate}\ Q(\overline{Tx}) \equiv R_1 \in C \\ \Gamma; (\Pi_1 \otimes [\overline{t_1}/\overline{x}]R_1) \vdash e : \exists x.T; R\end{array}}{\Gamma; \Pi \vdash \texttt{unpack}\ r@1\ Q(\overline{t_1})\ \texttt{in}\ e : \exists x.T; R}\ \textsc{Unpack1}$$

$$\frac{\begin{array}{c}\Gamma; \Pi \vdash r : C; r@k\ Q(\overline{t_1}) \otimes \Pi_1 \\ \texttt{predicate}\ Q(\overline{Tx}) \equiv R_1 \in C \qquad 0 < k < 1 \\ \Gamma; (\Pi_1 \otimes [\overline{t_1}/\overline{x}]R_1 \otimes \texttt{unpacked}(r@k\ Q(\overline{t_1})) \vdash e : \exists x.T; R \\ \forall r', \bar{t} : (\ \mathit{unpacked}(r'@k'\ Q(\bar{t})) \in \Pi \Rightarrow \Pi \vdash r \neq r')\end{array}}{\Gamma; \Pi \vdash \texttt{unpack}\ r@k\ Q(\overline{t_1})\ \texttt{in}\ e : \exists x.T; R}\ \textsc{Unpack2}$$

We have also developed rules for the dynamic semantics, that are used in proving the soundness of our system. Section 9 describes in detail the dynamic semantics rules and the soundness theorem.

7 Specification for Modularity

To illustrate the modularity issues, we present here a more realistic example than the queue example from Section 4. In Figure 10 we depict a simulator for two queues of jobs, containing large jobs (size>10) and small jobs (size<11). The example is relevant in queueing theory, where an optimal scheduling policy might separate the jobs in two queues, according to some criteria. The role of the control is to make each producer/consumer periodically take a step in the simulation. We have modeled two FIFO queues, two producers, two consumers and a control object. Each producer needs a pointer to the end of each queue, for adding a new job, and a pointer to the start of each queue, for initializing the start of the queue in case it becomes empty. Each consumer has a pointer to the start of one queue because it consumes the element that was introduced first in that queue. The control has a pointer to each producer and to each consumer. The queues are shared by the producers and consumers, thus giving rise to a number of aliased objects with fractions less than 1.

Now, let's say the system has to be modified, by introducing two queues for the small jobs and two queues for the large jobs, see right image of Figure 10. Ideally, the specification of the control object should not change, since the consumers and the producers have the same behavior as before: each producer produces both large and small jobs and each consumer accesses only one kind of job. We will show in the following sections that our methodology does not modify the specification of the control object, thus allowing us to make changes locally without influencing other code, while (first-order) separation logic approaches [9] will modify the specification of the controller.

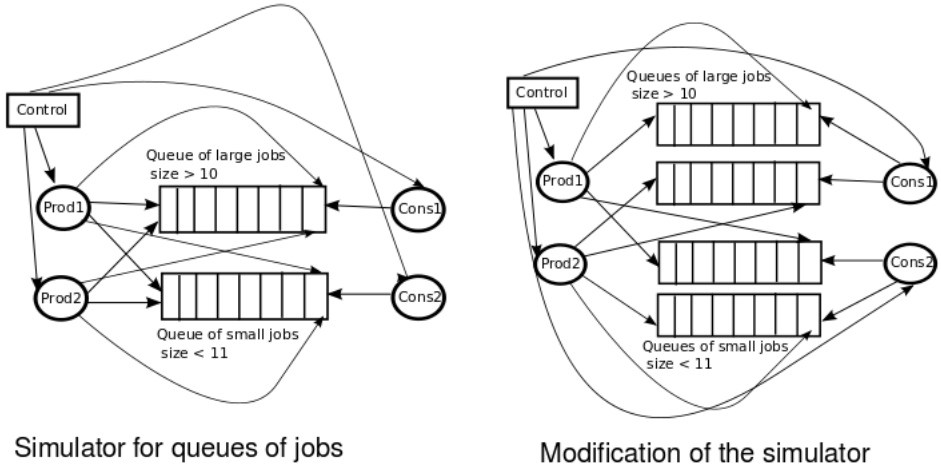

Fig. 10.

The code in Figures 11, 12 and 13 represents the initial running example from Figure 10. The predicates and the specifications of each class explain how the objects and methods should be used and what is their expected behavior. For example, the Producer object has access to the two queues, it expects the queues to be shared with other objects, but also that the elements of one queue will stay in the range [0,10], while the elements of the second queue will stay in the range [11,100].

Now, let's imagine changing the code to reflect the modifications in the right image of Figure 10. The internal representation of the predicates changes, but their external semantics stays the same: the producers produce jobs and they direct them to the appropriate queue, each consumer accesses only one kind of queue (either the queue of small jobs or the queue of big jobs), and the controller is still the manager of the system. The predicate `BothInRange()` of the Producer class is exactly the same. The predicate `ConsumeInRange(x,y)` of the Consumer class changes to

$$\texttt{ConsumeInRange(x,y)} \equiv \texttt{startJobs1} \rightarrow o_1 \otimes \texttt{startJobs2} \rightarrow o_2 \otimes o_1@k\ \ Range(x,y)\ \ \otimes\ \ o_2@k\ \ Range(x,y).$$

The predicate `WorkingSystem()` of the Control class does not change.

The local changes did not influence the specification of the Control class, thus conferring greater flexibility and modularity to the code.

The current separation logic approaches do not provide this modularity. Distefano and Parkinson [9] introduced jStar, an automatic verification tool based on separation logic aiming at programs written in Java. Although they are able to verify various design patterns and they can define abstract predicates that hide the name of the fields, they do not have a way of hiding the aliasing. In all

```
public class Producer {
    Link startSmallJobs,
         startLargeJobs;
    Link endSmallJobs,
         endLargeJobs;

  predicate BothInRange() ≡
    ∃o1, o2. startSmallJobs→ o1
          ⊗ startLargeJobs→ o2
          ⊗ o1@k1 Range(0, 10)
          ⊗ o2@k2 Range(11, 100)

  public Producer
    (Link ss, Link sl,
    Link es, Link el) {
        startSmallJobs = ss;
        startLargeJobs = sl;
        ...}

  public void produce()
   this@k BothInRange() —∘
     this@k BothInRange() {
   Random generator = new Random();
   int r = generator.nextInt(101);
   Link l = new Link(r, null);
   if (r <= 10)
   {   if (startSmallJobs == null)
         { startSmallJobs = l;
          endSmallJobs = l;}
    else
       {endSmallJobs.next = l;
         endSmallJobs= l;}
   }
   else
   {   if (startLargeJobs == null)
         { startLargeJobs = l;
          endLargeJobs = l;}
    else
       {endLargeJobs.next = l;
        endLargeJobs = l;}
   }
  }
}
```

Fig. 11. Producer class

```
public class Consumer {
      Link startJobs;

  predicate ConsumeInRange(int x, int y) ≡
    startJobs→ o ⊗ o@k Range(x, y)

  public Consumer(Link s) {
       startJobs = s;

  public void consume()
    ∀ x:int, y:int.
      this@k ConsumeInRange(x, y)
        —∘ this@k ConsumeInRange(x, y)
    { if (startJobs != null)
       {System.out.println(startJobs.val);
       startJobs = startJobs.next;}
    }
```

Fig. 12. Consumer class

```
public class Control {
      Producer prod1, prod2;
      Consumer cons1, cons2;

  predicate WorkingSystem() ≡
       prod1→ o1⊗ prod2→ o2
         ⊗ cons1→ o3⊗ cons2 → o4
         ⊗ o1@k1 BothInRange()
         ⊗ o2@k2 BothInRange()
         ⊗ o3@k3 ConsumeInRange(0, 10)
         ⊗ o4@k4 ConsumeInRange(11, 100)

  public Control(Producer p1, Producer p2,
              Consumer c1, Consumer c2) {
    prod1 = p1; prod2 = p2;
    cons1 = c1; cons2 = c2; }

  public void makeActive( int i)
   this@k WorkingSystem() —∘
       this@k WorkingSystem() {
    Random generator = new Random();
    int r = generator.nextInt(4);
    if (r == 0) {prod1.produce();}
      else if (r == 1) {prod2.produce();}
        else if (r == 2) {cons1.consume();}
          else {cons2.consume();}
    if (i > 0) { makeActive(i-1);}
        }
    }
```

Fig. 13. Control class

cases, they reveal which references point to the same shared data, and this violates the information hiding principle. By using access permissions, we can hide what is the data that two objects share. We present the specifications needed to verify the code in Figure 10 using separation logic.

The predicate for the Producer class is $Prod(this, ss, es, sl, el)$, where :

$Prod(p, ss, es, sl, el) \equiv p.startSmallJobs \rightarrow ss \ \star \ p.endSmallJobs \rightarrow es \ \star$
$p.startLargeJobs \rightarrow sl \ \star \ p.endLargeJobs \rightarrow el.$

The precondition for the `produce()` method is:

$Prod(p, ss, es, sl, el) \ \star \ Listseg(ss, null, 0, 10) \ \star \ Listseg(sl, null, 11, 100).$

The predicate for the Consumer class is

$Cons(c, s) \equiv c \rightarrow s.$

The precondition for the `consume()` method is:

$Cons(c, s) \ \star \ Listseg(s, null, 0, 10).$

The predicate for the Control class is :

$Ctrl(ct, p1, p2, c1, c2) \equiv ct.prod1 \rightarrow p1 \ \star \ ct.prod2 \rightarrow p2 \ \star$
$ct.cons1 \rightarrow c1 \ \star \ ct.cons2 \rightarrow c2.$

The precondition for `makeActive()` is:

$Ctrl(this, p1, p2, c1, c2) \ \star \ Prod(p1, ss, es, sl, el) \ \star \ Prod(p2, ss, es, sl, el) \ \star$
$Cons(c1, sl) \star Cons(c2, ss) \star Listseg(ss, null, 0, 10) \star Listseg(sl, null, 11, 100).$

The lack of modularity will manifest itself when we add the two queues as in the right image of Figure 10.

The predicates $Prod(p, ss, es, sl, el)$ and $Ctrl(ct, p1, p2, c1, c2)$ do not change, while the predicate $Cons(c, s1, s2)$ changes to

$Cons(c, s1, s2) \equiv c.startJobs1 \rightarrow s1 \ \star \ c.startJobs2 \rightarrow s2.$

The precondition for the `consume()` method becomes:

$Cons(c, s1, s2) \ \star \ Listseg(s1, null, 0, 10) \ \star \ Listseg(s2, null, 0, 10).$

Although the behavior of the Consumer and Producer classes have not changed, the precondition for `makeActive()` in class Control does change:

$Ctrl(this, p1, p2, c1, c2) \star Prod(p1, ss1, es1, sl1, el1) \star Prod(p2, ss2, es2, sl2, el2) \star$
$Cons(c1, sl1, sl2) \ \star \ Cons(c2, ss1, ss2) \ \star \ Listseg(ss1, null, 0, 10) \ \star$
$Listseg(ss2, null, 0, 10) \star Listseg(sl1, null, 11, 100) \star Listseg(sl2, null, 11, 100)$

The changes occur because the pointers to the job queues have been modified and the separation logic specifications have to reflect the changes. This leads to a loss of modularity.

8 Composite

The Composite design pattern [11] expresses the fact that clients treat individual objects and compositions of objects uniformly. Verifying implementations of the Composite pattern is challenging, especially when the invariants of objects in the tree depend on each other [18], and when interior nodes of the tree can be modified by external clients, without going through the root. As a result, verifying the Composite pattern is a well-known challenge problem, with some attempted solutions presented at SAVCBS 2008 (e.g. [4, 15]). We describe a new formalization and proof of the Composite pattern using fractions and object propositions

that provides more local reasoning than prior solutions. For example, in Jacobs et al. [15] a global description of the precise shape of the entire Composite tree must be explicitly manipulated by clients; in our solution a client simply has a fraction to the node in the tree it is dealing with.

We implement a popular version of the Composite design pattern, as an acyclic binary tree, where each Composite has a reference to its left and right children and to its parent. The code is given below.

```
public class Composite {

private Composite left, right, parent;
private int count;

public Composite()
{
this.count = 1;
this.left = null;
this.right = null;
this.right = null;
}

private void updateCountRec()
{
if (this.parent != null)
 {
  this.updateCount();
  this.parent.updateCountRec();
 }
else
  this.updateCount();
}

private void updateCount()
{
int newc = 1;
if (this.left != null)
  newc = newc + left.count;
if (this.right != null)
  newc = newc + right.count;
this.count = newc;
}

public void setLeft(Composite l)
{
l.parent= this;
this.left = l;
this.updateCountRec();
}

public void setRight(Composite r){
```

```
r.parent = this;
this.right = r;
this.updateCountRec();
}
}
```

Each Composite caches the size of its subtrees in a count field, so that a parent's count depends on its children's count. The dependency is in fact recursive, as the parent and right/left child pointers must be consistent. Clients can add a new subtree at any time, to any free position (where the current reference is null). This operation changes the count of all ancestors, which is done through a notification protocol. The pattern of circular dependencies and the notification mechanism are hard to capture with verification approaches based on ownership or uniqueness.

We assume that the notification terminates (that the tree has no cycles) and we verify that the Composite tree is well-formed: parent and child pointers line up and counts are consistent.

Previously the Composite pattern has been verified with a related approach based on access permissions and typestate [4]. This verification abstracted counts to an even/odd typestate and relied on non-formalized extensions of a formal system, whereas we have formalized the proof system and provide a full proof in the supplemental material. Our verification proves partial correctness of this version of the Composite pattern.

8.1 Specification

A Composite tree is well-formed if the field count of each node n contains the number of nodes of the tree rooted in n. A node of the Composite tree is a leaf when the left and right fields are null.

The goals of the specification are to allow clients to add a child to any node of the tree that has no left (or right) child. Since the count field of a node depends on the count fields of its children nodes, inserting a child must not violate the transitive parents' invariants.

We use the following methodology for verification: each node has a fractional permission to its children, and each child has a fractional permission to its parent. We allow unpacking of multiple object propositions as long as they satisfy the heap invariant: if two object propositions are unpacked and they refer to the same object then we require that they do not have fields in common.(Note that the invariant needs to hold irrespective of whether the object propositions are packed or unpacked.)

The predicates of the Composite class are presented in Figure 14.

The predicate *count* has a parameter c, which is an integer representing the value at the count field. There are two existentially quantified variables lc and rc, for the *count* fields of the left child lc and the right child rc. By $c = lc+rc+1$ we make sure that the count of *this* is equal to the sum of the counts for the children plus 1. By $this@\frac{1}{2}\ left(\mathsf{ol}, \mathsf{lc}) \otimes this@\frac{1}{2}\ right(\mathsf{or}, \mathsf{rc})$ we connect lc to

$$\begin{aligned}
&\text{predicate } \mathit{count}\ (\text{int c}) \equiv \exists \text{ol, or, lc, rc. this.count} \to \text{c} \otimes \\
&\qquad \text{c} = \text{lc} + \text{rc} + 1 \ \otimes \text{this@}\frac{1}{2}\ \mathit{left}(\text{ol, lc}) \\
&\qquad \otimes \mathit{this}@\frac{1}{2}\ \mathit{right}(\text{or, rc})
\end{aligned}$$

$$\begin{aligned}
&\text{predicate } \mathit{left}\ (\text{Composite ol}, \text{int lc}) \equiv \text{this.left} \to \text{ol} \otimes \\
&\qquad \big((\text{ol} \neq \text{null} \ \multimap \text{ol@}\frac{1}{2}\ \mathit{count}(\text{lc})) \\
&\qquad \oplus (\text{ol} = \text{null} \ \multimap \ \text{lc} = 0)\big)
\end{aligned}$$

$$\begin{aligned}
&\text{predicate } \mathit{right}\ (\text{Composite or}, \text{int rc}) \equiv \text{this.right} \to \text{or} \otimes \\
&\qquad \big((\text{or} \neq \text{null} \ \multimap \text{or@}\frac{1}{2}\ \mathit{count}(\text{rc})) \\
&\qquad \oplus (\text{or} = \text{null} \ \multimap \ \text{rc} = 0)\big)
\end{aligned}$$

$$\begin{aligned}
&\text{predicate } \mathit{parent}\ () \equiv \exists \text{op, c, k. this.parent} \to \text{op} \otimes \\
&\qquad \text{op} \neq \text{this} \ \otimes \ \text{this@}\frac{1}{2}\ \mathit{count}(\text{c}) \otimes \\
&\qquad \Big((\text{op} \neq \text{null} \ \multimap \text{op@k}\ \mathit{parent}() \otimes \\
&\qquad\quad (\text{op@}\frac{1}{2}\ \mathit{left}(\text{this, c}) \\
&\qquad \oplus \text{op@}\frac{1}{2}\ \mathit{right}(\text{this, c}))) \oplus \\
&\qquad (\text{op} = \text{null} \ \multimap \ \mathit{this}@\frac{1}{2}\ \mathit{count}(\text{c}))\Big)
\end{aligned}$$

Fig. 14. Predicates for Composite

the left child (through the *left* predicate) and *rc* to the right child (through the *right* predicate).

The predicate *left* expresses that the predicate *count*(*lc*) holds for *this.left*, the left child of *this*. The predicate *right* expresses that the predicate *count*(*rc*) holds for *this.right*, the right child of *this*. The permission for the *left* (*right*) predicate is split in equal fractions between the *count* predicate and the left (right) child's *parent* predicate.

Inside the *parent* predicate of *this*, there is a fractional permission to the *count* predicate (and implicitly to its *count* field) of *this*. The *parent* predicate contains only a fraction of $k < \frac{1}{2}$ to the parent of *this* so that any clients can use the remaining fraction to reference the node and add children to the parent. A client can actually use this to update the parent field, but in order to pack the *parent* predicate, the client has to conform to the well-formedness condition mentioned earlier.

If a new node is added to the tree as the left child of *this*, we need to change the *count* field of *this*. The field left of *this* must be null and the permission with a half fraction has to be acquired by unpacking the *count* predicate of *this*. This requires us to unpack the parent's *left* predicate, which requires the parent's *count* predicate, and so on to the root node. We can only pack it back when the tree is in a well-formed state. As the notification algorithm goes up the tree, from the current node to the root, we successively unpack the predicates corresponding to each node and we pack them back when the tree is well-formed. This ensures that if a new node is added, in order to pack the predicates again, the count fields must be updated and consistent!

The proof of partial correctness of the Composite pattern is presented in the supplemental material. The proof is currently done by hand.

The complete specification for each method is given below:

```
class Composite {
  private Composite left, right, parent;
  private int count;

  public Composite
⊸ this@½ parent() ⊗
this@½ left(null, 0) ⊗ this@½ right(null, 0)
    {...}

  private void updateCountRec ()
∃ k1. (unpacked(this@ k1 parent()) ⊗
∃ opp, lcc, k<½.unpacked(this@½ count(lcc))
⊗this.parent → opp ⊗ opp ≠ this ⊗
(((opp ≠ null ⊸ opp@k parent() ⊗
(opp@½ left(this, lcc) ⊕ opp@½ right(this, lcc))) ⊕
(opp = null ⊸ this@½ count(lcc)) ) ⊗
∃ ol, lc, or, rc, lcc'. this.count → lcc ⊗ lcc' = lc + rc + 1 ⊗
this@½ left(ol, lc) ⊗ this@½ right(or, rc)
⊸ this@k1 parent())
```

```
    {...}

  private void updateCount ()
∃ c, c1, c2, nc. unpacked(this@1 count(c)) ⊗
this.count → c ⊗ c = c1 + c2 + 1 ⊗
∃ ol, lc, or, rc. this@½ left(ol, lc) ⊗
this@½ right(or, rc) ⊗
∃ k1. unpacked(this@k1 parent())
⊸ this@1 count(nc) ⊗
nc = lc + rc + 1 ⊗ ∃ k. unpacked(this@k parent())
    {...}

  public void setLeft (Composite l)
 this ≠ l ⊗
∃k2.(∃ k1.this@k1 parent() ⊗ l@k2 parent() ⊗
this@½ left(null, 0) ⊸
∃ k.this@k parent() ⊗ l@k2 parent())
    {...}
}
```

The constructor of the class Composite returns half of the permission for the *left* and *right* predicate, and half of a permission to the *parent* predicate.

The method *updateCountRec*() takes in a fraction of $k1$ to the unpacked *parent* predicate and a half fraction to the unpacked *count* predicate of *this*, and it returns the $k1$ fraction to the packed *parent* predicate. This means that after calling this method, the *parent* predicate holds for *this*.

In the same way, the method *updateCount* takes in the unpacked predicate *count* for *this* object and it returns the *count* predicate packed for *this*. Thus, after calling *updateCount*(), the object *this* satisfies its *count* predicate.

The method $setLeft(Composite\ l)$ takes in a fraction to the *parent* predicate of *this*, a fraction to the *parent* predicate of l and the $left$ predicate of *this* with a null argument (saying that the left field of *this* is null and thus a client can attach a new left child here). The post-condition shows that after calling $setLeft$, some of the permission to the *parent* predicate of this has been consumed, while the fraction to the predicate *parent* of l stays the same.

9 Dynamic Semantics and Soundness

The dynamic semantics for our language is given in Figure 15. Below we describe the definitions used for dynamic semantics support.

$$
\begin{array}{rcll}
C(\overline{o}) & \in & \textsc{Objects} & \\
\mu & \in & r \rightsquigarrow \textsc{Objects} & \text{(stores)} \\
\rho & \in & l \rightsquigarrow \textsc{Values} & \text{(environments)} \\
F(\Pi) & ::= & (l \cup r \cup v) \rightsquigarrow \mathrm{R} & \text{(propositions)} \\
\Sigma & ::= & r \rightsquigarrow \textsc{Predicate} & \text{(store types)} \\
r & \in & \textsc{ObjectRefs} & \\
l & \in & \textsc{IndirectRefs} & \\
v & \in & \textsc{Variables} & \\
R & \in & \textsc{Object Propositions} &
\end{array}
$$

$$\frac{}{\mu, \rho, l \ \rightarrow \ \mu, \rho, \rho(l)} \ \textsc{Lookup}$$

$$\frac{o \notin dom(\mu) \qquad \mu' = \mu[o \rightarrow C(\overline{\rho(l)})]}{\mu, \rho, new \ C(\overline{l}) \ \rightarrow \ \mu', \rho, o} \ \textsc{New}$$

$$\frac{\mu, \rho, e_1 \rightarrow \mu', \rho', e'}{\mu, \rho, let \ x = e_1 \ in \ e_2 \ \rightarrow \ \mu', \rho', let \ x = e' \ in \ e_2} \ \textsc{Let-e}$$

$$\frac{l \notin dom(\rho)}{\mu, \rho, let \ x = o \ in \ e_2 \ \rightarrow \ \mu, \rho[l \rightsquigarrow o], [l/x]e_2} \ \textsc{Let-o}$$

$$\frac{v \in \{null, n, true, false\}}{\mu, \rho, let \ x = v \ in \ e_2 \ \rightarrow \ \mu, \rho, [v/x]e_2} \ \textsc{Let-v}$$

$$\frac{\mu(\rho(l_1)) = C(\overline{o}) \qquad fields(C) = \overline{Tf}}{\mu, \rho, l_1.f = l_2 \ \rightarrow \ \mu[\rho(l_1) \rightsquigarrow [\rho(l_2)/o_i]C(\overline{o})], \rho, \rho(l_2)} \ \textsc{Assign}$$

$$\frac{}{\mu, \rho, if \ (true) \ e_1 \ else \ e_2 \ \rightarrow \ \mu, \rho, e_1} \ \textsc{If-true}$$

$$\frac{}{\mu, \rho, if \ (false) \ e_1 \ else \ e_2 \ \rightarrow \ \mu, \rho, e_2} \ \textsc{If-false}$$

$$\frac{\begin{array}{c}\mu(\rho(l_1)) = C(\overline{o}) \\ method(m, C) = T_r \ m(\overline{x})\{return \ e\}\end{array}}{\mu, \rho, l_1.m(\overline{l_2}) \ \rightarrow \ \mu, \rho, [l_1/this, \overline{l_2}/\overline{x}]e} \ \textsc{Invoke}$$

$$\frac{\mu(\rho(l)) = C(\overline{o}) \qquad fields(C) = \overline{Tf}}{\mu, \rho, l.f_i \ \rightarrow \ \mu, \rho, o_i} \ \textsc{Field}$$

$$\frac{}{\mu, \rho, pack \ r@k \ R_1 \ in \ e_1 \ \rightarrow \ \mu, \rho, e_1} \ \textsc{Pack}$$

$$\frac{}{\mu, \rho, unpack \ r@k \ R_1 \ in \ e_1 \ \rightarrow \ \mu, \rho, e_1} \ \textsc{Unpack}$$

Fig. 15. Dynamic Semantics Rules

The semantics is a mostly-standard small-step operational semantics; the rules are complete except for standard rules to reduce binary and logical operators. We define the judgement $\mu, \rho, e \ \rightarrow \ \mu', \rho', e'$.

The main interesting feature is the use of indirect references, a proof technique adapted from [25]. The LET-O rule shows that when the left-hand expression of a let reduces to a reference o, instead of substituting o for x, we allocate a fresh *indirect reference* l, and add a mapping from l to o in an environment ρ. The LOOKUP rule later reduces l to an o, so that execution is isomorphic to a standard substitution-based semantics. However, the use of the ρ enviroment allows us to distinguish references that came from different variables, because they will have different indirect references l. This is useful for preservation, because the original variables may have had different permissions, and we need to preserve those different permissions when the variables are substituted with indirect references. Σ maps a reference to a predicate.Σ will contain actual values for the arguments of the predicates, since Σ is used at runtime.

The $\vdash PR$ represents the judgement that a program PR is well formed, meaning that all methods obey their specification. There are necessary helper judgements for *Program*, *Class* and *Method*. For space reasons, we do not present them here.

We define the consistency of the heap and environment, along with the semantics of predicates using the judgment $\mu, \Sigma, F(\Pi), \rho \vdash o \ \underline{ok}$. We check consistency for each reference o. For space reasons, we do not give the technical details o how we do this.

We do mention that in order to prove that the *invariants of* $\overline{P}$ *hold* (that the heap is in a consistent state), where $\overline{P} = objProps(\mu, \Sigma, F(\Pi), \rho, o)$, the conditions in Figure 16 have to hold. Note that in the third condition of Figure 16 def looks up the definition of the predicate of P_i in the defining class C_i.

- $P_i = unpacked(o@k_1\ Q) \in \overline{P} \Rightarrow \nexists P_{j \neq i} \in \overline{P}$ such that $P_j = unpacked(o@k_2\ Q)$
- $\forall o, Q\ \sum_{o_i = o, Q_i = Q} k_i \leq 1$, where $o_i@k_i\ Q_i$ or $unpacked(o_i@k_i\ Q_i)$
- $\forall i, j$ such that $pred(P_i) \neq pred(P_j)$, $fields(def(pred(P_i), C_i)) \cap fields(def(pred(P_j), C_j)) = \emptyset$, where $fields(o.f \rightarrow o') = f$, $pred(o.f \rightarrow o') = \emptyset$, $pred(o@k\ Q) = Q$ and $pred(unpacked(o@k\ Q)) = Q$

Fig. 16. Heap Invariants

Now we state the main preservation theorem that underlies the soundness of our system:

(Preservation Theorem)

If $\Gamma; \Pi \ \vdash \ e \ : \ \exists x.T; R$ and $\mu, \Sigma, F(\Pi), \rho \ \underline{ok}$ and $\mu, \rho, e \ \rightarrow \ \mu', \rho', e'$ and $\vdash PR$ then there exists Π' , Γ' and Σ' such that $\Gamma'; \Pi' \vdash e' : \exists x.T; R$ and $\mu', \Sigma', F(\Pi'), \rho' \ \underline{ok}$.

The proof uses induction over the derivation of $\mu, \rho, e \rightarrow \mu', \rho', e'$ in the standard way. In the proof, Π and Π' do not contain the $\oplus$ symbol. If the $\oplus$

symbol is added to these contexts, it is straightforward to use induction to prove preservation.

9.1 Proving the Preservation Theorem

Lemma 1. (Substitution) *If* $(\Gamma, y : T_y); (\Pi_1 \otimes R_y) \vdash e : \exists x.T; R$ *and* $\Gamma; \Pi_2 \vdash t : \exists x.T_y; [t/y]R_y$ *then* $\Gamma; ([t/y]\Pi_1 \otimes \Pi_2) \vdash [t/y]e : \exists x.T; [t/y]R$.

Proof of Substitution Lemma

The proof is by induction on the derivation of $(\Gamma, y : T_y); (\Pi_1 \otimes R_y) \vdash e : \exists x.T; R$. Note that there is a clear correspondence between the structure of e and which rule is used to type it. Thus the cases are on the structure of e rather than the rule by which the typing judgement was defined. In the proof, Π_1 and Π_2 do not contain the $\oplus$ symbol. If the $\oplus$ symbol was added to these contexts, it would be straightforward to use induction to prove the lemma using the rule $\oplus$.

1. e is a value v. The values that e can take in this case are $null|true|false|n$. We know $(\Gamma, y : T_y), (\Pi_1 \otimes R_y) \vdash v : \exists x : T.R$. Since v is a value, R will be trivially equal to v. Thus y does not appear in R and $R = [t/y]R$. Because $[e_1/y]e = v$ we trivially obtain that $\Gamma; ([t/y]\Pi_1 \otimes \Pi_2) \vdash v : \exists x : T.[t/y]R$.
2. e is a variable z,$z \neq y$. We know $(\Gamma, y : T_y); (\Pi_1 \otimes R_y) \vdash z : \exists x.T; R$. Since $z \neq y$, we know by inversion that $R \in \Pi_1$, with $R \equiv z@k\ Q(\overline{t_1})$ and $Q(\overline{t_1})$ might contain y as a free variable. Thus $[t/y]R \in [t/y]\Pi_1$ and $[t/y]e = z$. We can deduce that $\Gamma; ([t/y]\Pi_1 \otimes \Pi_2) \vdash z : \exists x.T; [t/y]R$.
3. e is the variable y. Now $[t/y]e = t$ and $T = T_y$ and $R = R_y$. Thus, $\Gamma; ([t/y]\Pi_1 \otimes \Pi_2) \vdash t : \exists x.T; [t/y]R$.
4. e is $r.f_i$. We know that $(\Gamma, y : T_y); (\Pi \otimes R_y) \vdash r.f_i : \exists x.T; R$. We also know by inversion that $\Pi \otimes R_y \vdash r.f_i \rightarrow r_i$ and that $\Gamma; (\Pi \otimes R_y) \vdash [r_i/x]R$. Using the induction hypothesis we have: $([t/y]\Pi \otimes \Pi_2) \vdash [t/y](r.f_i \rightarrow r_i)$ and $\Gamma; ([t/y]\Pi \otimes \Pi_2) \vdash [t/y][r_i/x]R$. Since $r.f_i$ is just the syntactic representation of a field, the substitution will happen in r_i: $([t/y]\Pi \otimes \Pi_2) \vdash (r.f_i \rightarrow [t/y]r_i)$. Also, we can rewrite $[t/y][r_i/x]R$ as $[([t/y]r_i)/x][t/y]R$ and so we have $\Gamma; ([t/y]\Pi \otimes \Pi_2) \vdash [([t/y]r_i)/x][t/y]R$. Using the rule (FIELD), we obtain that $\Gamma; ([t/y]\Pi \otimes \Pi_2) \vdash [t/y]r.f_i : \exists x : T.[t/y]R$, exactly what we wanted.
5. e is `new` $C(\overline{t_1})$. We know $(\Gamma, y : T_y); (\Pi \otimes R_y) \vdash$ `new` $C(\overline{t_1}) : \exists z.C; z.\overline{f} \rightarrow \overline{t_1}$. We also know by inversion that $(\Gamma, y : T_y) \vdash \overline{t_1 : T}$. Using the induction hypothesis we have $\Gamma \vdash \overline{[t/y]t_1 : T}$. Using the rule (NEW), we obtain that $\Gamma; ([t/y]\Pi \otimes \Pi_2) \vdash [t/y]$`new` $C(\overline{t_1})$: $\exists z.C; z.\overline{f} \rightarrow \overline{[t/y]t_1}$, exactly what we wanted.
6. e is `if(`t_1`){`e_1`}else{`e_2`}`. We know $(\Gamma, y : T_y); (\Pi \otimes R_y) \vdash$ `if(`t_1`){`e_1`}else{`e_2`}` : $\exists x.T; R_1 \oplus R_2$. We also know by inversion that $(\Gamma, y : T_y); (\Pi \otimes R_y \otimes t_1 = true) \vdash e_1 : \exists x.T; R_1$ and that $(\Gamma, y : T_y); (\Pi \otimes R_y \otimes t_1 = false) \vdash e_2 : \exists x.T; R_2$. Using the induction hypothesis and knowing that $\Gamma, \Pi_2 \vdash t : \exists x.T_y; [t/y]R_y$, we have $\Gamma; ([t/y]\Pi \otimes \Pi_2 \otimes t_1 = true) \vdash [t/y]e_1 : \exists x.T; [t/y]R_1$

and that $\Gamma; ([t/y]\Pi \otimes \Pi_2 \otimes t_1 = false) \vdash [t/y]e_1 : \exists x.T; [t/y]R_2$. By applying the (IF) rule, we obtain that $\Gamma; ([t/y]\Pi \otimes \Pi_2) \vdash \texttt{if}(t_1)\{e_1\}\texttt{else}\{e_2\} : \exists x.T; [t/y]R_1 \oplus [t/y]R_2$. Since $[t/y]R_1 \oplus [t/y]R_2 = [t/y](R_1 \oplus R_2)$ we obtain that $\Gamma; ([t/y]\Pi \otimes \Pi_2) \vdash \texttt{if}(t_1)\{e_1\}\texttt{else}\{e_2\} : \exists x.T; [t/y](R_1 \oplus R_2)$, exactly what we wanted.

7. e is $\texttt{let}\ x = e_1\ \texttt{in}\ e_2$. We know $(\Gamma, y : T_y); (\Pi \otimes R_y) \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \exists w.T_2; R$. We also know by inversion that $(\Gamma, y : T_y); (\Pi \otimes R_y) \vdash e_1 : \exists x.T_1; R_1 \otimes \Pi_2$. Using the induction hypothesis and knowing that $\Gamma; \Pi_3 \vdash t : \exists x.T_y; [t/y]R_y$, we obtain that $\Gamma; ([t/y]\Pi \otimes \Pi_3) \vdash [t/y]e_1 : \exists x.T_1; [t/y](R_1 \otimes \Pi_2)$. We also know by inversion that $(\Gamma, y : T_y, x : T_1); (R_1 \otimes \Pi_2) \vdash e_2 : \exists w.T_2; R_2$. This means that $(\Gamma, x : T_1); (\Pi_3 \otimes [t/y]R_1 \otimes [t/y]\Pi_2) \vdash [t/y]e_2 : \exists w.T_2; [t/y]R_2$. Now, we can apply the (LET) rule and we obtain that $\Gamma; ([t/y]\Pi \otimes \Pi_3) \vdash [t/y](\texttt{let}\ x = e_1\ \texttt{in}\ e_2) : \exists w.T_2; [t/y]R$, which is exactly what we wanted.
8. e is $\texttt{pack}\ r@k\ Q(\overline{t_1})\ \texttt{in}\ e$, with $0 < k < 1$. We know that $\Gamma; \Pi \vdash \texttt{pack}\ r@k\ Q(\overline{t_1})\ \texttt{in}\ e : \exists x.T; R$. We also know by inversion that $\Gamma; (\Pi_1 \otimes r@k\ Q(\overline{t_1})) \vdash e : \exists x.T; R$. Using the induction hypothesis and knowing that $\Gamma; \Pi_2 \vdash t : \exists x.T_y; [t/y]R_y$, we obtain that
$\Gamma; ([t/y](\Pi_1 \otimes r@k\ Q(\overline{t_1})) \otimes \Pi_2) \vdash [t/y]e : \exists x.T; [t/y]R$. The other two premises of the (PACK2) rule can also be obtained by inversion and they remain the same. From the first premise $\Gamma; \Pi \vdash r : C; [\overline{t_1}/\overline{x}]R_1 \otimes \mathsf{unpacked}(r@k\ Q(\overline{t_1})) \otimes \Pi_1$ we can deduce by induction that $\Gamma; ([t/y]\Pi \otimes \Pi_2) \vdash [t/y]r : C; [t/y]([\overline{t_1}/\overline{x}]R_1 \otimes \mathsf{unpacked}(r@k\ Q(\overline{t_1})) \otimes \Pi_1)$
So now we can apply the (PACK2) rule again and we get that
$\Gamma; ([t/y]\Pi \otimes \Pi_2) \vdash [t/y](\texttt{pack}\ r@k\ (Q(\overline{t_1})))\ \texttt{in}\ [t/y]e : \exists x.T; [t/y]R$.
Thus, $\Gamma; ([t/y]\Pi \otimes \Pi_2) \vdash$
$[t/y](\texttt{pack}\ r@k\ Q(\overline{t_1})\ \texttt{in}\ e) : \exists x.T; [t/y]R$, exactly what we wanted.
9. e is $\texttt{pack}\ r@1\ Q_2(\overline{t_2})\ \texttt{in}\ e$. The proof in this case is analogous to the one for the previous case, but the fraction k will be replaced by 1 across the proof.
10. e is $\texttt{unpack}\ r@k\ Q(\overline{t_1})\ \texttt{in}\ e$ for $0 < k < 1$. We know that $\Gamma; \Pi \vdash \texttt{unpack}\ r@k\ Q(\overline{t_1})\ \texttt{in}\ e : \exists x.T; R$. We also know by inversion that $\Gamma; (\Pi_1 \otimes [\overline{t_1}/\overline{x}]R_1 \otimes \mathsf{unpacked}(r@k\ Q(\overline{t_1})) \vdash e : \exists x.T; R$. Using the induction hypothesis and knowing that
$\Gamma; \Pi_2 \vdash t : \exists x.T_y; [t/y]R_y$, we obtain that
$\Gamma; ([t/y](\Pi_1 \otimes [\overline{t_1}/\overline{x}]R_1 \otimes \mathsf{unpacked}(r@k\ Q(\overline{t_1})) \otimes \Pi_2) \vdash [t/y]e : \exists x.T; [t/y]R$. The other premises of the (UNPACK2) rule can also be obtained by inversion and they remain the same. From the first premise $\Gamma; \Pi \vdash r : C; r@k\ Q(\overline{t_1}) \otimes \Pi_1$ we obtain by induction that $\Gamma; ([t/y]\Pi \otimes \Pi_2) \vdash [t/y]r : C; [t/y](r@k\ Q(\overline{t_1}) \otimes \Pi_1)$. Now we can apply the (UNPACK2) rule again and we get that $\Gamma; ([t/y]\Pi \otimes \Pi_2) \vdash \texttt{unpack}\ [t/y]r@k\ [t/y]Q(\overline{t_1})\ \texttt{in}\ [t/y]e : \exists x.T; [t/y]R$. Thus $\Gamma; ([t/y]\Pi \otimes \Pi_2) \vdash [t/y](\texttt{unpack}\ r@k\ in\ Q(\overline{t_1}))\ \texttt{in}\ [t/y]e : \exists x.T; [t/y]R$, exactly what we wanted to prove.
11. e is $\texttt{unpack}\ r@1\ Q(\overline{t_1})\ \texttt{in}\ e$. The proof in this case is analogous to the one for the previous case, but the fraction k will be replaced by 1 across the proof.

12. e is $r_0.m(\overline{t_1})$. We know that $\Gamma; \Pi \vdash r_0.m(\overline{t_1})$:
$\exists\, result.T_r; [r_0/this][\overline{t_1}/\overline{x}]R \otimes \Pi_1$. We know by inversion that $\Gamma; \Pi \vdash [r_0/this][\overline{t_1}/\overline{x}]R_1 \otimes \Pi_1$. Using the induction hypothesis and knowing that
$\Gamma, \Pi_2 \vdash t : \exists x.T_y; [t/y]R_y$, we obtain that
$\Gamma; ([t/y]\Pi \otimes \Pi_2) \vdash [t/y]([r_0/this][\overline{t_1}/\overline{x}]R_1 \otimes \Pi_1)$. This is equivalent to writing
$\Gamma; ([t/y]\Pi \otimes \Pi_2) \vdash [r_0/this][[\overline{t/y}t_1/\overline{x}][t/y]R_1 \otimes [t/y]\Pi_1$.
By inversion we know that $\Gamma \vdash r_0 : C_0$ and $\Gamma \vdash \overline{t_1 : T}$. Using the induction hypothesis we obtain that $\Gamma \vdash [t/y]r_0 : C_0$ and $\Gamma \vdash \overline{[t/y]t_1 : T}$. Also by inversion we know that $mtype(m, C_0) = \forall \overline{x : T}.\exists result.T_r; R_1' \multimap R$ and $R_1 implies R_1'$.
We can infer that $[t/y]R_1 \vdash [t/y]R_1'$ and that
$mtype(m, C_0) = \forall \overline{x : T}.\exists result : T_r.R_1' \multimap R$ will hold for $\overline{[t/y]t_1 : T}$ (because of the $\forall$ quantifier of the (MTYPE) judgement). We can now apply the (CALL) rule again and we obtain that
$\Gamma; ([t/y]\Pi \otimes \Pi_2) \vdash (([t/y]r_0).m(\overline{[t/y]t_1}))$:
$\exists\, result.T_r; [[t/y]r_0/this][\overline{[t/y]t_1}/\overline{x}][t/y]R \otimes [t/y]\Pi_1$.
Since $r_0.m(\overline{[t/y]t_1}) = [t/y](r_0.m(\overline{t_1}))$ and
$[r_0/this][\overline{[t/y]t_1}/\overline{x}][t/y]R = [t/y]([r_0/this][\overline{t_1}/\overline{x}]R)$, we obtain that $\Gamma; ([t/y]\Pi \otimes \Pi_2) \vdash [t/y](r_0.m(\overline{t_1})) : \exists\, result.T_r; [t/y]([r_0/this][\overline{t_1}/\overline{x}]R \otimes [t/y]\Pi_1)$. This is exactly what we wanted to prove.
13. e is $r_1.f_i = t_1$. We know that $\Gamma; \Pi \vdash (r_1.f_i = t_1)$:
$\exists x.T_i; x@k'\ Q'(\overline{t'}) \otimes t_1@k_0\ Q_0(\overline{t_0}) \otimes\ \ r_1.f_i \to t_1 \otimes \Pi_3$. We know by inversion that $\Gamma; \Pi \vdash t_1 : T_i; t_1@k_0\ Q_0(\overline{t_0}) \otimes \Pi_1$. Using the induction hypothesis and knowing that $\Gamma; \Pi_4 \vdash t : \exists x.T_y; [t/y]R_y$, we obtain that $\Gamma; ([t/y]\Pi_1 \otimes \Pi_4) \vdash [t/y]t_1 : T_i; [t/y](t_1@k_0\ Q_0(\overline{t_0}))$. This means that
$[t/y](t_1@k_0\ Q_0(\overline{t_0})) = ([t/y]t_1)@k_0\ [t/y]Q_0(\overline{t_0})$. The other premises of the (ASSIGN) rule can also be obtained by inversion. From the second premise $\Gamma; \Pi_1 \vdash r_1.f_i : T_i; r_i'@k'\ Q'(\overline{t'}) \otimes \Pi_2$ we get by induction that $\Gamma; ([t/y]\Pi_1 \otimes \Pi_4) \vdash r_1.f_i : T_i; [t/y](r_i'@k'\ Q'(\overline{t'}) \otimes \Pi_2)$. From the third premise $\Pi_2 \vdash r_1.f_i \to r_i' \otimes \Pi_3$ we get by induction that $([t/y]\Pi_2 \otimes \Pi_4) \vdash r_1.f_i \to [t/y]r_i' \otimes [t/y]\Pi_3$. So now we can apply the (ASSIGN) rule again and we get that $\Gamma; ([t/y]\Pi \otimes \Pi_4) \vdash r_1.f_i = [t/y]t_1 : \exists x.T_i; [t/y](x@k'\ Q'(\overline{t'})) \otimes [t/y](t@k_0\ Q_0(\overline{t_0})) \otimes\ r_1.f_i \to [t/y]t_1)$. Since $(r_1.f_i = [t/y]t_1) = ([t/y](r_1.f_i = t_1))$ and $[t/y](x@k'\ Q'(\overline{t'}) \otimes [t/y](t_1@k_0\ Q_0(\overline{t_0})) \otimes\ r_1.f_i \to [t/y]t_1 = [t/y](x@k'\ Q'(\overline{t'}) \otimes t@k_0\ Q_0(\overline{t_0}) \otimes\ r_1.f_i \to t_1)$, we finally obtain that
$\Gamma; ([t/y]\Pi_1 \otimes \Pi_4) \vdash [t/y](r_1.f_i = t_1)$:
$\exists x.T_i; [t/y](x@k'\ Q'(\overline{t'}) \otimes t_1@k_0\ Q_0(\overline{t_0}) \otimes\ \ r_1.f_i \to t_1 \otimes \Pi_3)$. This is exactly what we wanted to prove.
We have now gone through all the induction cases and we have completed the proof of the Substitution Lemma.

We also need to define the following lemma:

Lemma 2. (Memory Consistency)

1. *If* $\mu, (\Sigma, l \rightsquigarrow Q), (F(\Pi), l \rightsquigarrow R), \rho\ \underline{ok}$ *then* $\mu, (\Sigma, \rho(l) \rightsquigarrow Q), (F(\Pi), \rho(l) \rightsquigarrow R), \rho\ \underline{ok}$, *where* $R = x@k\ Q$.

2. *If* $\mu, \Sigma, F(\Pi), \rho\ \underline{ok}$ *and* $o \notin dom(\mu)$, *then* $\mu' = \mu[o \rightsquigarrow C(\overline{\rho(l)})], \Sigma' = (\Sigma, o \rightsquigarrow unpacked), F(\Pi') = F(\Pi), \rho\ \underline{ok}$.
3. *If* $\mu, (\Sigma, l \rightsquigarrow Q), (F(\Pi), l \rightsquigarrow R), \rho\ \underline{ok}$ *and* $l' \notin dom(\rho)$ *then* $\mu, (\Sigma, l' \rightsquigarrow Q), (F(\Pi), l' \rightsquigarrow R), \rho[l' \rightsquigarrow \rho(l)]\ \underline{ok}$, *where* $R = x@k\ Q$.
4. *If* $\mu, (\Sigma_1, \Sigma_2), (F(\Pi_1 \otimes \Pi_2)), \rho\ \underline{ok}$ *and* $unpacked(r@k\ Q(\overline{t_1})) \in \Pi_1$, *then* $\mu, (\Sigma_2, r \rightarrow Q(\overline{t_1})), (F(\Pi_2), r \rightsquigarrow r@k\ Q(\overline{t_1})), \rho\ \underline{ok}$.
5. *If* $\mu, (\Sigma_1, \Sigma_2), (F(\Pi_1 \otimes \Pi_2)), \rho\ \underline{ok}$ *and* $[\overline{t_2}/\overline{x}]R_2 \in \Pi_1$, *with* `predicate` $Q_2(\overline{T\ x}) \equiv R_2 \in C$, *then* $\mu, (\Sigma_2, r \rightsquigarrow Q_2(\overline{t_2})), (F(\Pi_2), r \rightsquigarrow x@1\ Q_2(\overline{t_2})), \rho\ \underline{ok}$.
6. *If* $\mu, (\Sigma_0, \Sigma_2), (F(\Pi_0 \otimes \Pi_2)), \rho\ \underline{ok}$ *and* $r@k\ Q(\overline{t_1}) \in \Pi_0$ *and* `predicate` $Q(\overline{T\ x}) \equiv R_1 \in C$ *and* $\forall r', \overline{x} : (\ unpacked(r'@k'\ Q(\overline{x})) \in (\Pi_0 \otimes \Pi_2) \Rightarrow \Pi_0 \otimes \Pi_2 \vdash r \neq r')$ *then* $\mu, \Sigma' = (\Sigma_2, r \rightsquigarrow unpacked)$, $F(\Pi') = (F(\Pi_2), F([\overline{t_1}/\overline{x}]R_1 \otimes r \rightsquigarrow unpacked(x@k\ Q(\overline{t_1}))))$, $\rho\ \underline{ok}$.
7. *If* $\mu, (\Sigma_0, \Sigma_2), (F(\Pi_0 \otimes \Pi_2)), \rho\ \underline{ok}$ *and* $r@k\ Q(\overline{t_1}) \in \Pi_0$ *and* `predicate` $Q(\overline{T\ x}) \equiv R_1 \in C$ *then* $\mu, \Sigma' = (\Sigma_2, r \rightsquigarrow unpacked)$, $F(\Pi') = (F(\Pi_2), F([\overline{t_1}/\overline{x}]R_1))$, $\rho\ \underline{ok}$.
8. *If* $\mu, \Sigma, F(\Pi), \rho\ \underline{ok}$ *and* $\mu(\rho(l)) = C(\overline{o})$ *and* $fields(C) = \overline{Tf}$ *then* $\mu, \Sigma' = (\Sigma, o_i \rightarrow Q), F(\Pi') = (F(\Pi), o_i \rightsquigarrow R), \rho\ \underline{ok}$, *where* $R = x@k\ Q$.
9. *If* $\mu, \Sigma = (\Sigma_0, l_2 \rightsquigarrow Q'(\overline{t'})), F(\Pi) = (F(\Pi_0), l_2 \rightsquigarrow y@k'\ Q'(\overline{t'}) \otimes x@k_0\ Q_0(\overline{t_0}) \otimes\ t_1.f_i \rightarrow x), \rho\ \underline{ok}$ *and* $\rho(l_2) = o_2$, *then* $\mu' = \mu[\rho(l_1) \rightsquigarrow [o_2/o_i]C(\overline{o})], \Sigma' = (\Sigma, o_2 \rightsquigarrow Q'(\overline{t'}))$, $F(\Pi') = (F(\Pi), o_2 \rightsquigarrow y@k'\ Q'(\overline{t'}) \otimes x@k_0\ Q_0(\overline{t_0}) \otimes\ t_1.f_i \rightarrow x), \rho\ \underline{ok}$

Proof of memory consistency lemma

1. Environment map
   Assuming $\mu, (\Sigma, l \rightsquigarrow Q), (F(\Pi), l \rightsquigarrow R), \rho\ \underline{ok}$ we need to show that $\mu, (\Sigma, \rho(l) \rightsquigarrow Q), (F(\Pi), \rho(l) \rightsquigarrow R), \rho\ \underline{ok}$, where $R = x@k\ Q$. Memory does not change. The only object potentially affected is $\rho(l)$, which is equal to o, say. Since $props(\mu, (\Sigma, l \rightsquigarrow Q), (F(\Pi), l \rightsquigarrow R), \rho, o) = props(\mu, (\Sigma, o \rightsquigarrow Q), (F(\Pi), o \rightsquigarrow R), \rho, o)$, we can conclude that $\mu, (\Sigma, \rho(l) \rightsquigarrow Q), (\Pi, \rho(l) \rightsquigarrow R), \rho \vdash o\ \underline{ok}$, and therefore $\mu, (\Sigma, o \rightsquigarrow Q), (F(\Pi), o \rightsquigarrow R), \rho\ \underline{ok}$.
2. New object
   Assuming $\mu, \Sigma, F(\Pi), \rho\ \underline{ok}$ and $o \notin dom(\mu)$, we have to show that $\mu' = \mu[o \rightsquigarrow C(\overline{\rho(l)})], \Sigma' = (\Sigma, o \rightsquigarrow unpacked), F(\Pi') = F(\Pi), \rho\ \underline{ok}$. It must be that $\overline{\rho(l) = o'}$ for some objects o'. The only objects affected are $o, \overline{o'}$. Since $\mu(o') = \mu'(o')$ and $\underline{props(\mu, \Sigma, F(\Pi), \rho, o')} = props(\mu', \Sigma', F(\Pi'), \rho, o')$ we can deduce that $\mu', \Sigma', F(\Pi'), \rho \vdash\ o'\ \underline{ok}$.
   The heap invariants are satisfied and we can deduce that $\mu', \Sigma', F(\Pi'), \rho \vdash o\ \underline{ok}$. Thus, $\mu', \Sigma', F(\Pi'), \rho\ \underline{ok}$.

3. Environment rename
   Assuming $\mu, (\Sigma, l \rightsquigarrow Q), (F(\Pi), l \rightsquigarrow R), \rho$ $\underline{ok}$ and $l' \notin dom(\rho)$, we have to show that $\mu, (\Sigma, l' \rightsquigarrow Q), (F(\Pi), l' \rightsquigarrow R), \rho[l' \rightsquigarrow \rho(l)]$ $\underline{ok}$, where $R = x@k\ Q$. The only object affected can be $\rho(l)$. By the same argument above, that the *props* sets are identical, we can conclude that $\mu, (\Sigma, l' \rightsquigarrow Q), (F(\Pi), l' \rightsquigarrow R), \rho[l' \rightsquigarrow \rho(l)]$ $\underline{ok}$.
4. Pack2
   Assuming $\Omega_1 = [\mu, \Sigma = (\Sigma_1, \Sigma_2), F(\Pi) = (F(\Pi_1 \otimes \Pi_2)), \rho]$ $\underline{ok}$, we have to show that $\Omega_2 = [\mu, \Sigma' = (\Sigma_2, r \rightsquigarrow Q(\overline{t_1})), F(\Pi') = (F(\Pi_2), r \rightsquigarrow r@k\ Q(\overline{t_1})), \rho]$ $\underline{ok}$.
   Let's take an arbitrary o. Since μ and ρ don't change, the only changes in the *objProps* corresponding to Ω_1 and to Ω_2 come from the different $o \rightsquigarrow R$ extracted from Π and from Π'. We have to show that the heap invariants are preserved by the different $o \rightsquigarrow R$ in $F(\Pi')$, knowing that the invariants are preserved by the different $o \rightsquigarrow R$ in $F(\Pi_1 \otimes \Pi_2)$. Knowing this, we deduce that the invariants cannot be broken by the assertions in Π_2. Thus, we only have to see if $r \rightsquigarrow x@k\ Q(\overline{t_1})$ is in contradiction with any assertions about r in Π_2. We also know that $unpacked(r@k\ Q(\overline{t_1}))$ is in Π_1.
   Since Ω_1 $\underline{ok}$, the only object propositions in Π_2 about r have to be of the form $r@k_1\ Q(\overline{t_1})$ such that the sum of k and the k_1 fractions is less than 1. Π_2 could also contain object propositions of the form $unpacked(r@k_1\ Q_i())$, but the fields in the predicates are disjoint, according to the heap invariants. Thus, $(\Pi_2, r \rightsquigarrow r@k\ Q(\overline{t_1}))$ satisfies the heap invariants, Σ' is compatible with Π' and the primitives are preserved, so μ, Σ', Π', ρ $\underline{ok}$.
5. Pack1
   Assuming $\Omega_1 = [\mu, (\Sigma_1, \Sigma_2), F(\Pi_1 \otimes \Pi_2), \rho]$ $\underline{ok}$, we have to show that $\Omega_2 = [\mu, \Sigma' = (\Sigma_2, r \rightsquigarrow Q_2(\overline{t_2})),$
   $F(\Pi') = (F(\Pi_2), r \rightsquigarrow r@1\ Q_2(\overline{t_2})), \rho]$ $\underline{ok}$,
   where $[\overline{t_2}/\overline{x}]R_2 \in \Pi_1$, with `predicate` $Q_2(\overline{T\ x}) \equiv R_2 \in C$. Let's take an arbitrary o. Since μ and ρ don't change, the only changes in the *objProps* corresponding to Ω_1 and to Ω_2 come from the different $o \rightsquigarrow R$ extracted from $F(\Pi_1 \otimes \Pi_2)$ and from $(F(\Pi_2), r \rightsquigarrow r@1\ Q_2(\overline{t_2}))$. We have to show that the heap invariants are preserved by the different $o \rightsquigarrow R$ in $(F(\Pi_2), r \rightsquigarrow r@1\ Q_2(\overline{t_2}))$, knowing that the invariants are preserved by the different $o \rightsquigarrow R$ in $F(\Pi_1 \otimes \Pi_2)$. Thus, we only have to see if $r \rightsquigarrow r@1\ Q_2(\overline{t_2})$ is in contradiction with any assertions about r in $F(\Pi_2)$.
   We know that $[\overline{t_2}/\overline{x}]R_2$ is in Π_1 and that Ω_1 $\underline{ok}$. When the field keys that are present in R_2 are packed to the object proposition Q_2 with a fraction of 1, these field keys cannot be used in any other object propositions. Knowing this, we deduce that the invariants cannot be broken by the assertions in Π_2. It follows that
   $(F(\Pi_2), r \rightsquigarrow r@1\ Q_2(\overline{t_2}))$ satisfies the heap invariants. Since $[\overline{t_2}/\overline{x}]R_2 \in \Pi_1$ and the primitives corresponding to $\Pi_1 \otimes \Pi_2$ are ok, there can be no primitives in Π_2 that contradict $[\overline{t_2}/\overline{x}]R_2$. We know that `predicate` $Q_2(\overline{T\ x}) \equiv R_2 \in C$ and we can deduce that the primitives corresponding to $\mu, \Sigma', F(\Pi'), \rho$ are ok. Thus $\mu, \Sigma', F(\Pi'), \rho$ $\underline{ok}$.
6. Unpack2

Assuming $\Omega_1 = [\mu, \Sigma = (\Sigma_0, \Sigma_2), F(\Pi) = F(\Pi_0 \otimes \Pi_2), \rho]$ $\underline{ok}$, we have to show that
$\Omega_2 = [\mu, \Sigma' = (\Sigma_2, r \rightsquigarrow unpacked)$,
$F(\Pi') = (F(\Pi_2), [\overline{t_1}/\overline{x}]R_1), \rho]$ $\underline{ok}$.
Let's take an arbitrary o. Since μ and ρ don't change, the only changes in the $objProps$ corresponding to Ω_1 and to Ω_2 come from the different $o \rightsquigarrow R$ extracted from $F(\Pi_0 \otimes \Pi_2)$ and from $F(\Pi')$. We have to show that the heap invariants are preserved by the different $o \rightsquigarrow R$ in $F(\Pi')$, knowing that the invariants are preserved by the different $o \rightsquigarrow R$ in $F(\Pi_0 \otimes \Pi_2)$. Knowing this, we deduce that the invariants cannot be broken by the assertions in Π_2. Thus, we only have to see if $r \rightsquigarrow \mathsf{unpacked}(r@k\ Q(\overline{t_1}))$ and $[\overline{t_1}/\overline{x}]R_1$ are in contradiction with any assertions about r in $F(\Pi_2)$.
Since $\forall r', \bar{t} : (\ unpacked(r'@k'\ Q(\bar{t})) \in (\Pi_0 \cup \Pi_2) \Rightarrow \Pi_0, \Pi_2 \vdash r \neq r')$ the heap invariants allow us to infer that Π_2 does not contain any object that is unpacked from the predicate Q and aliases with r. We also know that $r@k\ Q(\overline{t_1}) \in \Pi_0$. Using the heap invariants, we deduce that if there is an object proposition referring to r in Π_2, this object proposition must be $r@k_1\ Q(\overline{t_1})$ with the sum of fractions being less than 1. Π_2 might also contain $r@k_1\ Q_2(\overline{t_2})$ such that the field keys of Q_2 and Q are disjoint.
The formula $[\overline{t_1}/\overline{x}]R_1$ corresponds to r, after it got unpacked. In this formula there might be object propositions referring to r or to other references that appear in Π_2. Since r was packed to Q, using object propositions from Π_0, right before being unpacked and since $Q(\overline{x}) = R_1$, we deduce that $[\overline{t_1}/\overline{x}]R_1$ will only contain object propositions that are already in Π_0. This means that the different $o \rightsquigarrow R$ extracted from $F(\Pi_0 \otimes \Pi_2)$ are compatible with each other and with $r \rightsquigarrow \mathsf{unpacked}(r@k\ Q(\overline{t_1}))$ (same reasoning as in the previous paragraph).
The heap invariants hold of Π' because there is no object that aliases with r that is unpacked from Q in Π', and also because $r \rightsquigarrow \mathsf{unpacked}(r@k\ Q(\overline{t_1}))$, $r@k\ Q(\overline{t_1})$ and $[\overline{t_1}/\overline{x}]R_1$ do not contain object propositions or primitives that are not compatible. Thus, $\mu, \Sigma', F(\Pi'), \rho$ $\underline{ok}$

7. Unpack1
   The proof of this case is very similar to the proof of the previous case Unpack2.
8. Field read
   Assuming $\mu, \Sigma, F(\Pi), \rho$ $\underline{ok}$ and $\mu(\rho(l)) = C(\overline{o})$ and
   $fields(C) = \overline{Tf}$, we have to show that $\mu, \Sigma' = (\Sigma, o_i \rightsquigarrow Q), F(\Pi') = (F(\Pi), o_i \rightsquigarrow R), \rho$ $\underline{ok}$, where $R = x@k\ Q$. The only object affected is o_i. Because of the way $fieldProps(\mu, \Sigma')$ is defined, any object proposition about o_i will be extracted from the object propositions referring to $\mu(\rho(l))$, which are already in Π. This means that $props(\mu, \Sigma, F(\Pi), \rho, o_i) = props(\mu, \Sigma', F(\Pi'), \rho, o_i)$ and $\mu, \Sigma', F(\Pi'), \rho \vdash o_i$ $\underline{ok}$. Thus $\mu, \Sigma', F(\Pi'), \rho$ $\underline{ok}$.
9. Assignment
   Assuming $\mu, \Sigma = (\Sigma_0, l_2 \rightsquigarrow Q'(\overline{t'})), F(\Pi) = (F(\Pi_0), l_2 \rightsquigarrow l_2@k'\ Q'(\overline{t'})), \rho$ $\underline{ok}$ and $\rho(l_2) = o_2$, we have to prove that $\mu' = \mu[\rho(l_1) \rightsquigarrow [o_2/o_i]C(\overline{o})], \Sigma' =$

$(\Sigma, o_2 \rightsquigarrow Q'(\overline{t'})), F(\Pi') = (F(\Pi), o_2 \rightsquigarrow o_2@k'\ Q'(\overline{t'})), \rho\ \underline{ok}$. The only object that changes is o_i.
Since $props(\mu, \Sigma, F(\Pi), \rho, o_i) = props(\mu', \Sigma', F(\Pi'), \rho, o_i)$ and
$\mu, \Sigma, F(\Pi), \rho \vdash o_i\ \underline{ok}$, we can conclude that
$\mu', \Sigma', F(\Pi'), \rho \vdash o_i\ \underline{ok}$ and thus $\mu', \Sigma', F(\Pi'), \rho\ \underline{ok}$.

The proof for the Preservation Theorem is done by induction on the dynamic semantics rules.

Proof of the Preservation Theorem
Case (LOOKUP)

1. By assumption
   (a) $\Gamma, \Pi\ \vdash\ l : \exists\ x.T; R$
   (b) $\mu, \Sigma, F(\Pi), \rho\ \underline{ok}$
   (c) $\mu, \rho, l\ \rightarrow\ \mu, \rho, \rho(l)$
2. By inversion on 1a
   (a) $\Gamma = (\Gamma_1,\ l : T)$
   (b) $F(\Pi) = (F(\Pi_1), l \rightsquigarrow\ R)$, where $R = x@k\ Q$
   (c) $\Sigma = (\Sigma_1, l \rightsquigarrow Q)$, where Σ_1 is the store type corresponding to Π_1.
3. $\mu, (\Gamma_1, l : T), (F(\Pi_1), l \rightsquigarrow R), (\Sigma_1, l \rightsquigarrow Q)\ \underline{ok}$ -by 2
4. $\rho(l) = o$, for some o - by Object Proposition Consistency
5. $(\Gamma, o : T), (\Pi_1, o \rightsquigarrow R) \vdash o : \exists x.T; R$ -by (TERM)
6. Let $\Gamma' = (\Gamma, \rho(l) : T)$, $F(\Pi') = (F(\Pi_1), \rho(l) \rightsquigarrow R)$ and $\Sigma' = (\Sigma_1, \rho(l) \rightsquigarrow Q)$
7. $\Gamma', \Pi' \vdash \rho(l) : \exists x.T; R$ -by 6,5
8. $\mu, (\Sigma_1, \rho(l)\ \rightsquigarrow\ Q), (F(\Pi_1), \rho(l)\ \rightsquigarrow\ R), \rho\ \underline{ok}$ -by 3,4, memory consistency lemma
9. $\mu, \Sigma', F(\Pi'), \rho\ \underline{ok}$ -by 6, 8
10. q.e.d -by 7, 9

Case (NEW)

1. By assumption
   (a) $\Gamma, \Pi \vdash new\ C(\bar{l})\ : \exists y.T; R$
   (b) $\mu, \Sigma, F(\Pi), \rho\ \underline{ok}$
   (c) $\mu, \rho, new\ C(\bar{l}) \rightarrow \mu', \rho, o$
   (d) $o \notin dom(\mu)$
   (e) $\mu' = \mu[o \rightarrow C(\overline{\rho(l)})]$
2. By inversion on 1a
   (a) $\exists y.T; R = \exists z.C; z.\overline{f} \rightarrow \overline{t} \otimes \Pi_1$
   (b) $\Gamma = (\Gamma_1, \overline{l : T})$
   (c) $fields(C) = \overline{T\ f}$
3. Let $\Gamma' = (\Gamma, o : C), F(\Pi') = (F(\Pi_1), o \rightsquigarrow (o.\overline{f} \rightarrow \overline{t}))$
4. Let $\Sigma' = (\Sigma, o \rightsquigarrow \mathsf{unpacked})$
5. $\Gamma', \Pi' \vdash o : \exists z.C; z.\overline{f} \rightarrow \overline{t} \otimes \Pi_1$ -by (TERM)
6. $\mu[o \rightsquigarrow C(\overline{\rho(l)})], (\Sigma, o \rightsquigarrow \mathsf{unpacked}), (F(\Pi_1), o \rightsquigarrow (o.\overline{f} \rightarrow \overline{t})), \rho\ \underline{ok}$ -by memory consistency lemma
7. q.e.d. -by 5, 6

Case (LET-O)

1. By assumption
   (a) $\Gamma, \Pi \vdash let\ x = o\ in\ e_2 : \exists\ w.T_2; R_3$
   (b) $\mu, \Sigma, F(\Pi), \rho\ \underline{ok}$
   (c) $\mu, \rho, let\ x = o\ in\ e_2 \rightarrow \mu, \rho[l \rightsquigarrow o], [l/x]e_2$
   (d) $l \notin dom(\rho)$
2. By inversion on 1a
   (a) $\Gamma; \Pi \vdash o : \exists x.T_1; R_1 \otimes \Pi_2$
   (b) $(\Gamma, x : T_1); (\Pi_2 \otimes R_1) \vdash e_2 : \exists w.T_2; R_3$
3. $\Gamma = (\Gamma_1, o : T_1)$, $F(\Pi) = (F(\Pi_2), o \rightsquigarrow R_1)$ -by inversion on 2a
4. Also, $\Sigma = (\Sigma_1, o \rightsquigarrow Q_1)$, where $R_1 = x@k\ Q_1$
5. Let $\Gamma' = (\Gamma, l : T_1)$, $F(\Pi') = (F(\Pi_2), l \rightsquigarrow R_1)$, $\Sigma' = (\Sigma_2, l \rightsquigarrow Q_1)$, where Σ_2 corresponds to Π_2
6. $(\Gamma, l : T_1); (F(\Pi_2), l \rightsquigarrow R_1) \vdash [l/x]e_2 : \exists w.T_2; [l/x]R_3$ -by 1d, 2b, Substitution Lemma
7. $\Gamma', \Pi' \vdash e_2 : \exists\ w.T_2; R_3$ -by 6
8. $\mu, (\Sigma_2, o \rightsquigarrow Q_1), (F(\Pi_2), o \rightsquigarrow R_1), \rho\ \underline{ok}$ -by 2a
9. $\mu, (\Sigma_2, l \rightsquigarrow Q_1), (F(\Pi_2), l \rightsquigarrow R_1), \rho[l \rightsquigarrow o]\ \underline{ok}$ -by memory consistency lemma
10. $\mu, \Sigma', F(\Pi'), \rho[l \rightsquigarrow o]\ \underline{ok}$
11. q.e.d. -by 10, 7

Case (LET-E)

1. By assumption
   (a) $\Gamma, \Pi \vdash let\ x = e_1\ in\ e_2 : \exists w.T_2; R_3$
   (b) $\mu, \Sigma, F(\Pi), \rho\ \underline{ok}$
   (c) $\mu, \rho, let\ x = e_1\ in\ e_2 \rightarrow \mu', \rho', let\ x = e_1'\ in\ e_2$
   (d) $\mu, \rho, e_1 \rightarrow \mu', \rho', e'$
2. By inversion on 1a
   (a) $\Gamma, \Pi \vdash e_1 : \exists x.T_1; R_1 \otimes \Pi_2$
   (b) $\Gamma; (R_1 \otimes \Pi_2) \vdash e_2 : \exists w.T_2; R_3$
3. By induction on 1b, 1d, 2a
   (a) $\exists \Gamma_0; \Pi'$ such that $\Gamma_0, \Pi' \vdash e' : \exists x.T_1; R_1$
   (b) $\exists\ \Sigma'$ such that $\mu', \Sigma', F(\Pi'), \rho'\ \underline{ok}$
4. Let $\Gamma' = \Gamma \cup \Gamma_0$
5. $\Gamma'; \Pi' \vdash let\ x = e_1'\ in\ e_2 : \exists w.T_2; R_3$ -by 3a,2b, (LET)
6. q.e.d. -by 3b,5

Case (LET-V) is very similar to Case (LET-O) and we have not included it here.

Case (PACK)

Subcase: the static semantics rule corresponding to (PACK) is (PACK2).

1. By assumption
   (a) $\Gamma, \Pi \vdash pack\ r@k\ Q(\overline{t_1})\ in\ e_1 : \exists x.T; R$
   (b) $\mu, \Sigma, F(\Pi), \rho\ \underline{ok}$
   (c) $\mu, \rho, pack\ r@k\ Q(\overline{t_1})\ in\ e_1 \rightarrow \mu, \rho, e_1$

2. By inversion on (PACK2)
   (a) $\Gamma; \Pi \vdash r : C.[\overline{t_1}/\overline{x}]R_1 \otimes$
   $\mathsf{unpacked}(r@k\ Q(\overline{t_1})) \otimes \Pi_1$
   (b) $\Gamma; (\Pi_1 \otimes r@k\ Q(\overline{t_1})) \vdash e : \exists x.T; R$
   (c) $\mathtt{predicate}\ Q(\overline{x}) \equiv R_1$
   (d) $0 < k < 1$
3. Let $F(\Pi') = (F(\Pi_1), r \rightsquigarrow r@k\ Q(\overline{t_1}))$, $\Sigma' = (\Sigma_1, r \rightsquigarrow Q(\overline{t_1}))$, $\Gamma' = \Gamma$
4. $\Gamma', \Pi' \vdash e : \exists x.T; R$ -by 2b, 3
5. $\mu, (\Sigma_1, r \rightsquigarrow Q(\overline{t_1})), (F(\Pi_1), r \rightsquigarrow r@k\ Q(\overline{t_1})), \rho\ \underline{ok}$ -by memory consistency lemma
6. q.e.d. -by 4, 5

Subcase: the static semantics rule corresponding to (PACK) is (PACK1).

1. By assumption
   (a) $\Gamma, \Pi \vdash pack\ r@1\ Q_2(\overline{t_2})\ in\ e : \exists x.T; R$
   (b) $\mu, \Sigma, F(\Pi), \rho\ \underline{ok}$
   (c) $\mu, \rho, pack\ r@1\ Q_2(\overline{t_2})\ in\ e \rightarrow \mu, \rho, e$
2. By inversion on (PACK1)
   (a) $\Gamma; \Pi \vdash r : C; [\overline{t_2}/\overline{x}]R_2 \otimes \Pi_1$
   (b) $\Gamma; (\Pi_1 \otimes r@1\ Q_2(\overline{t_2})) \vdash e : \exists x.T; R$
3. Let $F(\Pi') = (F(\Pi_1), r \rightsquigarrow r@1\ Q_2(\overline{t_2}))$, $\Sigma' = (\Sigma_1, r \rightsquigarrow Q_2(\overline{t_2}))$, $\Gamma' = \Gamma$
4. $\Gamma', \Pi' \vdash e : \exists x.T; R$ -by 2b, 3
5. $\mu, (\Sigma_1, r \rightsquigarrow Q_2(\overline{t_2})), (F(\Pi_1), r \rightsquigarrow r@1\ Q_2(\overline{t_2})), \rho\ \underline{ok}$ -by memory consistency lemma
6. q.e.d. -by 4, 5

Case (UNPACK)
Subcase: the static semantics rule corresponding to (UNPACK) is (UNPACK2).

1. By assumption
   (a) $\Gamma, \Pi \vdash unpack\ r@k\ Q(\overline{t_1})\ in\ e : \exists x.T; R$
   (b) $\mu, \Sigma, F(\Pi), \rho\ \underline{ok}$
   (c) $\mu, \rho, unpack\ r@k\ Q(\overline{t_1})\ in\ e \rightarrow \mu, \rho, e$
2. By inversion on (UNPACK2)
   (a) $\Gamma; \Pi \vdash r : C; r@k\ Q(\overline{t_1}) \otimes \Pi_1$
   (b) $\Gamma; (\Pi_1 \otimes [\overline{t_1}/\overline{x}]R_1 \otimes$
   $\mathsf{unpacked}(r@k\ Q(\overline{t_1}))) \vdash e : \exists x.T; R$
   (c) $\forall r', \overline{t} : (\ unpacked(r'@k'\ Q(\overline{t})) \in \Pi \Rightarrow \Pi \vdash r \neq r')$
   (d) $\mathtt{predicate}\ Q(\overline{x}) \equiv R_1 \in C$
   (e) $0 < k < 1$
3. Let $F(\Pi') = (F(\Pi_2 \otimes [\overline{t_1}/\overline{x}]R_1)$,
   $r \rightsquigarrow \mathsf{unpacked}(r@k\ Q(\overline{t_1})))$,
   $\Sigma' = (\Sigma_2, r \rightsquigarrow unpacked)$, $\Gamma' = \Gamma$
4. $\Gamma', \Pi' \vdash e : \exists x.T; R$ -by 2b, 3
5. $\mu, (\Sigma_2, r \rightsquigarrow unpacked), (F(\Pi_2 \otimes [\overline{t_1}/\overline{x}]R_1)$,
   $r \rightsquigarrow \mathsf{unpacked}(r@k\ Q(\overline{t_1}))), \rho\ \underline{ok}$ -by memory consistency lemma
6. q.e.d. -by 4, 5

Subcase: the static semantics rule corresponding to (UNPACK) is (UNPACK1).

1. By assumption
   (a) $\Gamma, \Pi \vdash unpack\ r@1\ Q(\overline{t_1})\ in\ e : \exists x.T; R$
   (b) $\mu, \Sigma, F(\Pi), \rho\ \underline{ok}$
   (c) $\mu, \rho, unpack\ r@1\ Q(\overline{t_1})\ in\ e \rightarrow \mu, \rho, e$
2. By inversion on (UNPACK1)
   (a) $\Gamma; \Pi \vdash r : T_1; r@1\ Q(\overline{t_1}) \otimes \Pi_1$
   (b) $\Gamma; (\Pi_1, [\overline{t_1}/\overline{x}]R_1) \vdash e : \exists x.T; R$
   (c) `predicate` $Q(\overline{x}) \equiv R_1 \in C$
3. Let $F(\Pi') = F(\Pi_1 \otimes [\overline{t_1}/\overline{x}]R_1)$,
   $\Sigma' = (\Sigma_1, r \rightsquigarrow unpacked)$, $\Gamma' = \Gamma$
4. $\Gamma', \Pi' \vdash e : \exists x.T; R$ -by 2b, 3
5. $\mu, (\Sigma_1, r \rightsquigarrow unpacked), F(\Pi_1 \otimes [\overline{t_1}/\overline{x}]R_1), \rho\ \underline{ok}$ -by memory consistency lemma
6. q.e.d. -by 4, 5

Case (IF-TRUE)

1. By assumption
   (a) $\Gamma, \Pi \vdash\ if(true)\{e_1\}else\{e_2\} : \exists x.T; R$
   (b) $\mu, \Sigma, F(\Pi), \rho\ \underline{ok}$
   (c) $\mu, \rho, if(true)\{e_1\}\{e_2\} \rightarrow \mu, \rho, e_1$
   (d) $\exists x.T; R = \exists x.T; R_1 \oplus R_2$
2. By inversion on the static semantics rule (IF): $\Gamma, \Pi \vdash e_1 : \exists x.T; R_1$
3. Let $\Gamma' = \Gamma, \Pi' = \Pi, \Sigma' = \Sigma$
4. $R_1 \oplus R_2$ is true if R_1 is true or if R_2 is true
5. $\Gamma', \Pi' \vdash e_1 : \exists x.T; R_1 \oplus R_2$ -by 2,3,4
6. $\mu, \Sigma', F(\Pi'), \rho\ \underline{ok}$ -by 3,1b
7. q.e.d. -by 5,6

Case (IF-FALSE)

1. By assumption
   (a) $\Gamma, \Pi \vdash\ if(false)\{e_1\}else\{e_2\} : \exists x.T; R$
   (b) $\mu, \Sigma, F(\Pi), \rho\ \underline{ok}$
   (c) $\mu, \rho, if(false)\{e_1\}else\{e_2\} \rightarrow \mu, \rho, e_2$
   (d) $\exists x.T; R = \exists x.T; R_1 \oplus R_2$
2. By inversion on the static semantics rule (IF): $\Gamma, \Pi \vdash e_2 : \exists x.T; R_2$
3. Let $\Gamma' = \Gamma, \Pi' = \Pi, \Sigma' = \Sigma$
4. $R_1 \oplus R_2$ is true if R_1 is true or if R_2 is true
5. $\Gamma', \Pi' \vdash e_2 : \exists x.T; R_1 \oplus R_2$ -by 2,3,4
6. $\mu, \Sigma', F(\Pi'), \rho\ \underline{ok}$ -by 3,1b
7. q.e.d. -by 5,6

Case (FIELD)

1. By assumption
   (a) $\Gamma, \Pi \vdash l.f_i : \exists x.T_r; R$
   (b) $\mu, \Sigma, F(\Pi), \rho\ \underline{ok}$

(c) $\mu, \rho, l.f_i \ \rightarrow \ \mu', \rho, o_i$
(d) $\mu(\rho(l)) = C(\overline{o})$
(e) $fields(C) = \overline{Tf}$
(f) $T_r = T_i$
2. By inversion on the static semantics rule (FIELD)
(a) $l.f_i : T_r$ *is a field of* C
(b) $\Gamma; \Pi \vdash [l.f_i/x]R$
3. Let $\Gamma' = (\Gamma, o_i : T_i), F(\Pi') = (F(\Pi), o_i \rightsquigarrow R)$
4. Let $\Sigma' = (\Sigma, o_i \rightsquigarrow Q)$, where $R = x@k\ Q$
5. $\Gamma', \Pi' \vdash o_i : \exists x.T_i; R$ -by (TERM)
6. $\mu, \Sigma' = (\Sigma, o_i \rightsquigarrow Q), F(\Pi') = (F(\Pi), o_i \rightsquigarrow R)), \rho\ \underline{ok}$ -by memory consistency lemma
7. q.e.d. -by 5,6

Case (ASSIGN)

1. By assumption
(a) $\Gamma, \Pi \vdash (l_1.f = l_2) : \exists x.T; R$
(b) $\mu, \Sigma, \Pi, \rho\ \underline{ok}$
(c) $\mu, \rho, (l_1.f = l_2) \ \rightarrow$
$\mu[\rho(l_1) \rightsquigarrow [\rho(l_2)/o_i]C(\overline{o})], \rho, \rho(l_2)$
(d) $\mu(\rho(l_1)) = C(\overline{o})$
(e) $fields(C) = \overline{Tf}$
2. By inversion on the static semantics rule (ASSIGN)
(a) $\Gamma; \Pi \vdash l_2 : T_i; l_2@k_0\ Q_0(\overline{t_0}) \otimes \Pi_1$, thus $T = T_i$
(b) $\Gamma; \Pi_1 \vdash l_1.f : T_i; r_i@k'\ Q'(\overline{t'}) \otimes \Pi_2$
(c) $\Pi_2 \vdash l_1.f \rightarrow r_i \otimes \Pi_3$
(d) $\exists x.T; R = \exists x.T_i; x@k'\ Q'(\overline{t'}) \otimes l_2@k_0\ Q_0(\overline{t_0}) \otimes \ \ l_1.f \rightarrow l_2 \otimes \Pi_3$
3. $\exists\ o_2$ such that $\rho(l_2) = o_2$.
4. Let $\Gamma' = (G, o_2 : T_i)$, $F(\Pi') = (F(\Pi \otimes l_2@k_0\ Q_0(\overline{t_0}) \otimes \ \ t_1.f_i \rightarrow l_2), o_2 \rightsquigarrow x@k'\ Q'(\overline{t'}))$, $\Sigma' = (\Sigma, o_2 \rightsquigarrow Q'(\overline{t'}))$.
5. $\Gamma', \Pi' \vdash o_2 : \exists x.T_i; R$ -by (TERM)
6. $\mu' = \mu[\rho(l_1) \rightsquigarrow [\rho(l_2)/o_i]C(\overline{o})], \Sigma', F(\Pi'), \rho\ \underline{ok}$ -by memory consistency lemma
7. q.e.d. -by 5, 6
Case (INVOKE)
(a) By assumption
i. $\Gamma, \Pi \vdash l_1.m(\overline{l_2}) : \exists x.T; R'$
ii. $\mu, \Sigma, F(\Pi), \rho\ \underline{ok}$
iii. $\mu, \rho, l_1.m(\overline{l_2}) \ \rightarrow \ \mu, \rho, [l_1/this, \overline{l_2}/\overline{x}]e$
iv. $\vdash PR$
v. $\mu(\rho(l_1)) = C(\overline{o})$
vi. $method(m, C) = T_r\ m(\overline{x})\{return\ e\}$
(b) By inversion on the static semantics rule (CALL)
i. $\Gamma \vdash l_1 : C$ and $\Gamma \vdash \overline{l_2 : T}$
ii. $\Gamma; \Pi \vdash [l_1/this][\overline{l_2}/\overline{x}]R_1 \otimes \Pi_1$
iii. $mtype(m, C) = \forall x : \overline{T}.\exists result.T_r; R_1' \multimap R$

iv. $R_1 \vdash R_1'$
v. $\exists x.T; R' = \exists\ result.T_r; [l_1/this][\overline{l_2}/\overline{x}]R$

(c) From 7(a)iv we know that the body $\{return\ e\}$ of the method m implements its specification, so the result will be of the type $\exists x.T_r; R'$, given the arguments of the right type.

(d) By the substitution Lemma, we know that $[l_1/this, \overline{l_2}/\overline{x}]e$ will be of the type $\exists x.T_r; R'$. Since $\mu, \Sigma, F(\Pi), \rho$ do not change, $\mu, \Sigma, F(\Pi), \rho\ \underline{ok}$.

(e) q.e.d., by 7d, 7(a)iv.

The cases LET-V, BINOP, AND, OR, NOT are trivial and they preserve soundness. After having proved the Preservation Theorem, we should prove the Progress Theorem for our soundness proof to be complete. Since our system is similar to Featherweight Java and we did not add new features to the language, the Progress Theorem automatically holds. This concludes out soundness proof.

$$\frac{bindFields(R, R, \mu) = \bigwedge v\ binop\ v \quad \vdash v\ binop\ v}{primitives_{ok}(R, \mu)}\ primitives_{ok}$$

$$\frac{bindFields(R_i, R, \mu) = v_i \quad i = 1, 2 \quad \vdash v_1\ binop\ v_2}{bindFields(R_1\ binop\ R_2, R, \mu) = R_1'\ binop\ R_2'}\ binop_{ok}$$

$$\frac{bindFields(R_i, R, \mu) = v_i \quad i = 1, 2 \quad \vdash v_1\ \wedge\ v_2}{bindFields(R_1\ \oplus\ R_2, R, \mu) = R_1' \oplus R_2'}\ and_{ok}$$

$$\frac{\mu[o, f] = v}{bindFields(o.f \to x, R, \mu) = v}\ field_{ok}$$

$$\frac{\mu[o, f] = v \qquad f \to x \in R}{bindFields(x, R, \mu) = v}\ var_{ok}$$

Fig. 17. Rules for $primitives_{ok}$

Our system inherits a Progress property from related object calculi such as Featherweith Java.

10 Related Work

There are two main lines of research that give partial solutions for the verification of object-oriented code in the presence of aliasing: the permission-based work and the separation logic approaches.

Bierhoff and Aldrich [3] developed access permissions, an abstraction that combines typestate and object aliasing information. Developers use access permissions to express the design intent of their protocols in annotations on methods and classes. Our work is a generalization of their work, as we use object propositions to modularly check that implementations follow their design intent. The typestate [7] formulation has certain limits of expressiveness: it is only suited to finite state abstractions. This makes it unsuitable for describing fields that contain integers and can take an infinite number of values and can satisfy various arithmetical properties. Our object propositions have the advantage that they can express predicates over an infinite domain, such as the integers.

Access permissions allow predicate changes even if objects are aliased in unknown ways. States and fractions [5] capture alias types, borrowing, adoption, and focus with a single mechanism. In Boyland's work, a fractional permission means immutability (instead of sharing) to ensure non-interference of permissions. We use fractions to keep object propositions consistent but track, split, and join fractions in the same way as Boyland.

Boogie [1] is a modular reusable verifier for Spec# programs. It provides design-time feedback and generates verification conditions to be passed to an automatic theorem prover. While Boogie allows a client to depend on properties of objects that it owns, we allow a client to depend on properties of objects that it doesn't own, too.

Krishnaswami et al. [17] show how to modularly verify programs written using dynamically-generated bidirectional dependency information. They introduce a ramification operator in higher-order separation logic that explains how local changes alter the knowledge of the rest of the heap. Their solution is application specific, as they need to find a version of the frame rule specifically for their library. Our methodology is a general one that can potentially be used for verifying any object-oriented program.

Nanevski et al. [20] developed Hoare Type Theory (HTT), which combines a dependently typed, higher-order language with stateful computations. While HTT offers a semantic framework for elaborating more practical external languages, our work targets Java-like languages and does not have the complexity overhead of higher-order logic.

Summers and Drossopoulou [24] introduce Considerate Reasoning, an invariant-based verification technique adopting a relaxed visible-state semantics. Considerate Reasoning allows distinguished invariants to be broken in the initial states of method executions, provided that the methods re-establish the invariant in the final state. The authors demonstrate Considerate Reasoning based on the Composite pattern and provide the encoding of their technique in the Boogie intermediate verification language [1], facilitating the automatic verification of the Composite pattern specification. Despite the fundamental differences in underlying methodology (visble-state invariants vs. abstract predicates) and logic between Considerate Reasoning and our approach, there are interesting analogies in the specification of the Composite pattern. For instance, the method that triggers the bottom-up traversal of the Composite to update a composite's count

field in the Considerate Reasoning specification does not expect the composite invariant in the method's initial state. This is similar to our method updateCountRec() which requires the predicates parent and count to be unpacked.

Cohen et al. [6] use locally checked invariants to verify concurrent C programs. In their approach, each object has an invariant, a unique owner and they use handles (read permissions) to accommodate shared objects. The disadvantage is their high annotation overhead and the need to introduce ghost fields. We do not have to change the code in order to verify our specifications.

Our work uses abstract predicates, similar to the work of Parkinson and Bierman [21] and Dinsdale-Young et al. [8]. The abstraction makes it easy to change the internal representation of a predicate without modifying the client's external view of it. The main mechanism is still separation logic, with its shortcomings. Unlike separation logic, we permit sharing of predicates with an invariant-based methodology. This avoids non-local characterizations of the heap structure, as required (for example) in Bart Jacob's Composition pattern solution [15].

There exist a set of verification methodologies for object-oriented programs in a concurrent setting: [8, 14, 19, 16]. These approaches can express externally imposed invariants on shared objects, but only for invariants that are associated with the lock protecting that object. In many cases, it may be inappropriate to associate such an invariant with the lock: for example, in a singlethreaded setting, there is no such lock. Even in multithreaded settings, a high level lock may protect a data structure with internal sharing, in which case specifying that sharing in the lock would break the modularity of the data structure. Thus, these systems do not provide an adequate solution to the modular verification problem we consider.

11 Implementation

11.1 Verification of Range example in Boogie

The Boogie encoding of the queues of integers example from section 4 is given below. The code below is the implementation of our methodology for the queues of integers example and it is verified by Boogie.

```
//type Ref is intended to represent object references
type Ref;
type PredicateTypes;
type FractionType = [Ref, PredicateTypes] int;
type UnpackedType = [Ref, PredicateTypes] bool;

const null:Ref;
const unique RangeP:PredicateTypes;

var val: [Ref]int;
var next: [Ref]Ref;
var frac: FractionType;
var unpacked: UnpackedType;
```

```
function Range(this: Ref, val:[Ref]int, next:[Ref]Ref,
         frac: FractionType, x:int, y:int) returns (bool);

//frame theorem
axiom (forall this:Ref, x:int, y:int, val: [Ref]int,
       val1: [Ref]int, next: [Ref]Ref , frac: FractionType ::
 (Range(next[this],val,next,frac,x,y) &&
 ((val1[this]!=val[this]) || (val1[this]==val[this])) &&
 (forall r:Ref :: r!=this ==> val1[r]==val[r]) )
     ==> Range(next[this],val1,next,frac,x,y)
);

//these axioms are for packing the Range predicate
axiom (forall this:Ref, x:int, y:int, val: [Ref]int,
       next: [Ref]Ref , frac: FractionType ::
{Range(this,val,next,frac,x,y)}
 ((this != null) &&  (next[this] == null) &&
 (val[this] >= x) && (val[this] <= y) ==>
     Range(this,val,next,frac,x,y)
));

axiom (forall this:Ref, x:int, y:int, val: [Ref]int,
      next: [Ref]Ref, frac: FractionType ::
{Range(this,val,next,frac,x,y)}
 ((this != null) && (next[this] != null) && (val[this] >= x) &&
 (val[this] <= y) && Range(next[this],val,next,frac,x,y)
       ==> Range(this,val,next,frac,x,y) )
);

axiom (forall this:Ref, x:int, y:int, val: [Ref]int,
      next: [Ref]Ref, frac: FractionType ::
{Range(this,val,next,frac,x,y)}
     (this==null) ==> Range(this,val,next,frac,x,y)
);

//this axiom is used for unpacking
axiom (forall this:Ref, x:int, y:int, val: [Ref]int,
      next: [Ref]Ref, frac: FractionType ::
{Range(this,val,next,frac,x,y)}
 ( Range(this,val,next,frac,x,y) ==>
     (this != null) && (val[this] >= x) && (val[this] <= y) &&
  Range(next[this],val,next,frac,x,y) && (frac[this, RangeP]>=50)
  )
);

//modulo is not implemented in Boogie
//so its properties have to be described
function modulo(x:int, y:int) returns (int);
  axiom (forall x:int, y:int :: {modulo(x,y)}
```

```
    ((0<=x) &&(0<y) ==> (0<= modulo(x,y) ) && (modulo(x,y)<y) )
    &&
    ((0<=x) &&(y<0) ==> (0<= modulo(x,y) ) && (modulo(x,y)<-y) )
    &&
    ((x<=0) &&(0<y) ==> (-y<= modulo(x,y) ) && (modulo(x,y)<=0) )
    &&
    ((x<=0) &&(y<0) ==> (y<= modulo(x,y) ) && (modulo(x,y)<=0) )
   );

procedure addModulo11(this: Ref, x:int)
modifies val, unpacked;
//need to put this in for modulo
requires this!=null;
requires val[this]>=0 && x>=0;
//this could be 100 here
requires frac[this, RangeP]>=50;
requires (forall r:Ref :: unpacked[r, RangeP]==false);
ensures  Range(this,val,next,frac,0,10);
ensures frac[this, RangeP]>=50;
{
 assert unpacked[this, RangeP]==false;
 assume Range(this,val,next,frac,0,10);
 assert (this != null) && (val[this] >= 0) && (val[this] <= 10)
        && Range(next[this],val,next,frac,0,10)   ;
 unpacked[this, RangeP]:=true;

 val[this] := modulo((val[this]+x),11);
 assert (this != null);
 assert (val[this] >= 0) && (val[this] <= 10);
 assert Range(next[this],val,next,frac,0,10);
 unpacked[this, RangeP]:=false;

  if (next[this] != null )
  {
    call addModulo11(next[this], x);
  }
  assert Range(next[this], val, next,frac, 0, 10);
  //val[this] was not modified
  assume old(val[this])==val[this];
}
```

11.2 Verification of Composite pattern in Boogie

The Boogie encoding of our instance of the composite pattern from section 8 is given below. The code below is the implementation of our methodology for the composite example and it is verified by Boogie.

```
//type Ref represents object references
type Ref;
```

```
type PredicateTypes;
type FractionType = [Ref, PredicateTypes] int;
type UnpackedType = [Ref, PredicateTypes] bool;

const null: Ref;
const unique leftP: PredicateTypes;
const unique rightP: PredicateTypes;
const unique parentP: PredicateTypes;
const unique countP: PredicateTypes;

var left: [Ref]Ref;
var right: [Ref]Ref;
var parent: [Ref]Ref;
var count: [Ref]int;
var unpacked: UnpackedType;
var frac: FractionType;


//axiom that shows there are no cycles in the tree, locally
//this axiom describes the data structure, does not depend on whether
//a predicate holds or not
axiom (forall this: Ref, l:Ref, left: [Ref]Ref,
      right: [Ref]Ref, parent:[Ref]Ref::
    (this!=right[this]) && (this!=left[this]) && (this!=parent[this]));

//axiom stating that this is a binary tree
axiom (forall this: Ref, left: [Ref]Ref, right: [Ref]Ref, parent:[Ref]Ref::
    (this!=null) && (parent[this]!=null) ==>
    ((this==right[parent[this]]) || (this==left[parent[this]]))
);

function leftPred(this: Ref, left: [Ref]Ref, right: [Ref]Ref,
         count: [Ref]int, frac: FractionType, lc: int) returns (bool);
axiom (forall this: Ref, left: [Ref]Ref, right: [Ref]Ref,
      count: [Ref]int, frac: FractionType,  lc: int::
  ((  (left[this] != null) &&
      countPred(left[this],left, right, count, frac, lc) &&
      (frac[left[this], countP] >= 50) )
       ==> (leftPred(this, left, right, count, frac, lc) ) )
  &&
  ( ( (left[this] == null) &&
   (lc==0) )
    ==> (leftPred(this, left, right, count, frac, 0) ) )
);

//this axiom is for unpacking of left predicate when left[this]!= null
axiom (forall this: Ref, left: [Ref]Ref, right: [Ref]Ref,
      count: [Ref]int, frac: FractionType,  lc: int::
  ( (leftPred(this, left, right, count, frac, lc) &&
    (left[this] != null) )==>
```

```
  ( countPred(left[this],left, right, count, frac, lc) &&
    (frac[left[this], countP] >= 50) ) ) ) ;

//this axiom is for unpacking of left predicate when left[this]== null
axiom (forall this: Ref, left: [Ref]Ref, right: [Ref]Ref,
      count: [Ref]int, frac: FractionType::
  {leftPred(this, left, right, count, frac, 0)}
   (leftPred(this, left, right,count, frac, 0)
    ==> (left[this] == null) ) );

function rightPred(this: Ref, left: [Ref]Ref, right: [Ref]Ref,
         count: [Ref]int, frac: FractionType,  rc: int) returns (bool);
axiom (forall this: Ref, left: [Ref]Ref, right: [Ref]Ref,
      count: [Ref]int, frac: FractionType,  rc: int::
  ((  (right[this] != null) &&
    countPred(right[this],left, right, count, frac, rc) &&
    (frac[right[this], countP] >= 50) )
     ==> rightPred(this, left, right, count, frac, rc)  )
  &&
  ((  (right[this] == null) &&
  (rc==0) )
  ==> rightPred(this, left, right, count, frac, 0) )
);

//this axiom is for unpacking of right predicate when right[this]!=null
axiom (forall this: Ref, left: [Ref]Ref, right: [Ref]Ref,
      count: [Ref]int, frac: FractionType,  rc: int::
  (rightPred(this, left, right, count, frac, rc) &&
  (right[this] != null) ==>
  (countPred(right[this],left, right, count, frac, rc) &&
  (frac[right[this], countP] >= 50) )));

//this axiom is for unpacking of right predicate when right[this]==null
axiom (forall this: Ref, left: [Ref]Ref, right: [Ref]Ref,
      count: [Ref]int, frac: FractionType::
   (rightPred(this, left, right, count, frac, 0)
   ==> (right[this] == null) ) );

function countPred(this: Ref, left: [Ref]Ref, right: [Ref]Ref,
         count: [Ref]int, frac: FractionType, c: int) returns (bool);
axiom (forall this: Ref, left: [Ref]Ref, right: [Ref]Ref,
      count: [Ref]int, frac: FractionType, c:int ::
 ( ( (this!=null) && (
  exists lc: int, rc:int ::
   (leftPred(this, left, right, count, frac, lc) &&
   (frac[this, leftP] >= 50) &&
   rightPred(this, left, right, count, frac, rc) &&
   (frac[this, rightP] >= 50) &&
   (count[this] == lc+rc+1) &&
   ( count[this]== c) )
```

```
) ) ==> countPred(this, left, right, count, frac, c) )
);

//this axiom is used for unpacking of count predicate
axiom (forall this: Ref, left: [Ref]Ref, right: [Ref]Ref,
      count: [Ref]int, frac: FractionType, c:int ::
 ( countPred(this, left, right, count, frac, c) ==>
    ( (this!=null) &&
  (exists lc: int, rc:int ::
  (leftPred(this, left, right, count, frac, lc) &&
  (frac[this, leftP] >= 50) &&
  rightPred(this, left, right, count, frac, rc) &&
  (frac[this, rightP] >= 50) &&
  (count[this] == lc+rc+1) &&
  (count[this]== c))
) ) )
);

axiom (forall this:Ref,count: [Ref]int ::
      (this==null) ==> (count[this]==0)  );

function parentPred(this: Ref, left: [Ref]Ref, right: [Ref]Ref, parent: [Ref]Ref,
         count: [Ref]int, frac: FractionType) returns (bool);
axiom ( forall this: Ref, left: [Ref]Ref, right: [Ref]Ref, parent: [Ref]Ref,
      count: [Ref]int, frac: FractionType ::
  (
    (parent[this] != this)
    &&
    countPred(this, left, right, count, frac, count[this])
    &&
    (frac[this, countP]>=50)
    &&
    (parent[this] != null)
    &&
    parentPred(parent[this], left, right, parent, count, frac)
    &&
    (
    (leftPred(parent[this], left, right, count, frac, count[this]) &&
     (frac[parent[this], leftP] >= 50))
    ||
     (rightPred(parent[this], left, right, count, frac, count[this]) &&
     (frac[parent[this], rightP] >= 50))
    )
  )
  ==>
  parentPred(this, left, right, parent, count, frac)
);

axiom ( forall this: Ref, left: [Ref]Ref, right: [Ref]Ref, parent: [Ref]Ref,
      count: [Ref]int, frac: FractionType ::
```

```
  (
  (parent[this] != this)
   &&
   countPred(this, left, right, count, frac, count[this])
   &&
   (frac[this, countP]>=100)
   &&
   (parent[this]==null)
  )
  ==>
  parentPred(this, left, right, parent, count, frac)
);

//this axiom is used for unpacking of parentPred
axiom ( forall this: Ref, left: [Ref]Ref, right: [Ref]Ref, parent: [Ref]Ref,
      count: [Ref]int, frac: FractionType ::
  {parentPred(this, left, right, parent, count, frac)}
  ( parentPred(this, left, right, parent, count, frac) ==>
  ( (parent[this] != this)
       &&
   ( exists c:int::
     ((countPred(this, left, right, count, frac, c)
     &&
     (frac[this, countP]>=50)
     &&
     ((parent[this] != null) ==>
       (parentPred(parent[this], left, right, parent, count, frac)
       &&
       ((leftPred(parent[this], left, right, count, frac, c) &&
       (frac[parent[this], leftP] >= 50))
       ||
       (rightPred(parent[this], left, right, count, frac, c) &&
       (frac[parent[this], rightP] >= 50))
       ) ) )
     &&
     ((parent[this]==null) ==>
      countPred(this, left, right, count, frac, c) &&
      (frac[this,countP]>=50) )  ) ))
  )  ) );

//axioms about the existance of a variable and instantiation
axiom (forall this: Ref, left: [Ref]Ref, right: [Ref]Ref,
      count: [Ref]int, frac: FractionType::
    (exists lc:int :: leftPred(this, left, right, count, frac, lc)
     ==> leftPred(this, left, right, count, frac, count[left[this]]))
);

axiom (forall this: Ref, left: [Ref]Ref, right: [Ref]Ref,
      count: [Ref]int, frac: FractionType::
    (exists rc:int :: rightPred(this, left, right, count, frac, rc)
```

```
     ==> rightPred(this, left, right, count, frac, count[right[this]]))
);

axiom (forall this: Ref, left: [Ref]Ref, right: [Ref]Ref,
      count: [Ref]int, frac: FractionType::
    (exists c:int :: countPred(this, left, right, count, frac, c)
    ==> (countPred(this, left, right, count, frac, count[this])) )
);

//frame axiom for count and leftPred
axiom (forall this: Ref, left: [Ref]Ref, right: [Ref]Ref,
      count: [Ref]int,  count1: [Ref]int, frac: FractionType,
lc: int::
 (leftPred(this, left, right, count, frac, lc) &&
 (forall r:Ref:: r!=this ==> count[r]==count1[r])
  ==> leftPred(this, left, right, count1, frac, lc))
 );

//frame axiom for when frac is modified, for leftPred
axiom (forall this: Ref, left: [Ref]Ref, right: [Ref]Ref,
      count: [Ref]int,  frac: FractionType, frac1: FractionType, lc:int::
 (leftPred(this, left, right, count, frac, lc) &&
 (forall r:Ref:: r!=this==>  (frac1[r,countP]==frac[r,countP]) )
  ==> (leftPred(this, left, right, count, frac1, lc)))
 );


//frame axiom for countPred
axiom (forall this: Ref, left: [Ref]Ref, right: [Ref]Ref, parent: [Ref]Ref,
      count: [Ref]int,  left1: [Ref]Ref, frac: FractionType,  c: int::
 (countPred(this, left, right, count, frac, c) &&
 (left1[parent[this]]==this)
 ==> countPred(this, left1, right, count, frac, c))
 );


procedure Composite(this:Ref)
modifies left, right, parent, count, frac, unpacked;
requires this != null;
ensures parentPred(this, left, right, parent, count, frac);
ensures leftPred(this, left, right, count, frac, 0);
ensures rightPred(this, left, right, count, frac, 0);
ensures (frac[this, parentP] >= 50) &&
        (frac[this, leftP] >= 50) &&
        (frac[this, rightP] >= 50) ;
ensures unpacked[this, parentP]==false;
{
  count[this] := 1;
  left[this] := null;
  right[this] := null;
```

```
  parent[this] := null;
  assert leftPred(this, left, right, count, frac, 0);
  frac[this, leftP] := 100;
  assert rightPred(this, left, right, count, frac, 0);
  frac[this, rightP] :=100;
  assert leftPred(this, left, right, count, frac, 0) &&
         rightPred(this, left, right, count, frac, 0) ;
  assert countPred(this, left, right, count, frac, 1);
  frac[this, countP] := 100;
  assert leftPred(this, left, right, count, frac, 0) &&
         rightPred(this, left, right, count, frac, 0) ;
  assert countPred(this, left, right, count, frac, 1);
  assert parentPred(this, left, right, parent, count, frac);
  frac[this, parentP] := 100;
  assert leftPred(this, left, right, count, frac, 0) &&
         rightPred(this, left, right,count, frac, 0) ;
  assert countPred(this, left, right, count, frac, 1);
  assert parentPred(this, left, right, parent, count, frac);
  unpacked[this, parentP]:=false;
}

procedure updateCountRec(this: Ref)
modifies count, frac, unpacked;
requires this!=null;
requires unpacked[this, parentP];
requires parent[this]!=null
         ==> (frac[parent[this], parentP]>0);
requires (parent[this]!=null) &&
         (this==right[parent[this]])
         ==> (frac[parent[this], rightP]>=50);
requires (parent[this]!=null) &&
         (this==left[parent[this]])
         ==> (frac[parent[this], leftP]>=50);
requires parent[this]==null
         ==> (frac[this, countP] >= 50);
requires (forall r:Ref:: (r!=this)
         ==> (unpacked[r, countP]==false));
requires unpacked[this, leftP]==false;
requires unpacked[this, rightP]==false;
requires (frac[this,leftP] >=50) &&
         (frac[this,rightP] >=50);
requires (frac[this, countP] >= 50);

ensures (parentPred(this, left, right, parent, count, frac) );
ensures unpacked[this, parentP]==false;

{
var fracLocal: FractionType;

assume (exists c1:int ::
```

```
       leftPred(this, left, right, count, frac, c1));
assume (exists c2:int ::
      rightPred(this, left, right, count, frac, c2));

  if (parent[this] != null)
 {
//we assert what is true when unpacked[this, parentP]
assert frac[this, countP]>=50;
fracLocal[this, countP]:=frac[this, countP];

//this asserts that we have a fraction to parentP
assert (frac[parent[this], parentP]>0);
//we unpack parent[this] from parentP
unpacked[parent[this], parentP]:=true;
assert (forall r:Ref:: (r!=this) ==>
       (unpacked[r, countP]==false));

//we can assume this because we just unpacked
//parent[this] from parentP
assume frac[parent[this], countP]>=50;
fracLocal[parent[this], countP]:=frac[parent[this], countP];

//we unpack parent[this] from countP
unpacked[parent[this], countP]:=true;

//we can assume this because we just unpacked parent[this] from countP
assume (frac[this,leftP]>=50) && (frac[this,rightP]>=50);
fracLocal[parent[this], rightP]:=frac[parent[this], rightP];


//we assume this is the right child of parent[this]
//the other case is analogous
assume this==right[parent[this]];
assert (parent[this]!=null) &&
       (this==right[parent[this]])
       ==> frac[parent[this],rightP]>=50;
fracLocal[parent[this],rightP]:=fracLocal[parent[this], rightP]
                                + frac[parent[this],rightP];

assert (fracLocal[parent[this],rightP]>=100);
//unpack parent[this] from rightP
unpacked[parent[this], rightP]:=true;
assert unpacked[parent[this],rightP]
       ==> frac[right[parent[this]],countP]>=50;
fracLocal[this, countP]:=fracLocal[this, countP]+frac[this, countP];

assert (frac[this,leftP] >=50) &&   (frac[this,rightP] >=50);
assert unpacked[this, parentP];
frac[this, countP]:=fracLocal[this, countP];
assert (frac[this, countP] >= 100);
```

```
assert leftPred(this, left, right, count, frac, count[left[this]]);
assert (exists c1:int :: leftPred(this, left, right, count, frac, c1));
unpacked[this, leftP]:=false;
assert rightPred(this, left, right, count, frac, count[right[this]]);
assert (exists c2:int :: rightPred(this, left, right, count, frac, c2));
unpacked[this, rightP]:=false;
      call updateCount(this);
assert( exists c:int ::
          countPred(this, left, right, count, frac, c)
        );
assert countPred(this, left, right, count, frac, count[this]);
assert unpacked[this, countP]==false;
assert (frac[this, countP]>=100)  ;

//assert the preconditions of updateCountRec
assert parent[this]!=null;
unpacked[parent[this], parentP]:=true;

//we can assume these because we have just
//unpacked parent[this] from parentP
assume parent[parent[this]]!=null
       ==> (frac[parent[parent[this]], parentP]>0);
assume parent[parent[this]]==null
       ==> (frac[parent[this], countP] >= 50);
assume (parent[parent[this]]!=null) &&
       (this==right[parent[parent[this]]])
       ==> (frac[parent[parent[this]], rightP]>=50);
assume (parent[parent[this]]!=null) &&
       (this==left[parent[parent[this]]])
       ==> (frac[parent[parent[this]], leftP]>=50);

assert rightPred(parent[this], left, right, count, frac, count[this]);
assert (exists c2:int :: rightPred(parent[this], left, right, count, frac, c2));
unpacked[parent[this],rightP]:=false;
assert (forall r:Ref:: (r!=parent[this])
       ==> (unpacked[r, countP]==false));

//assertions for proving leftPred(parent[this]...
//we assume this, the other case is trivial
assume left[parent[this]] != null;
assert unpacked[left[parent[this]], countP]==false;
//we can assume this because
//countPred is packed for left[parent[this]]
assume countPred(left[parent[this]],left,
                 right, count, frac, count[this]);

assert leftPred(parent[this], left, right, count,
                frac, count[left[parent[this]]]);
assert (exists c1:int ::
       leftPred(parent[this], left, right, count, frac, c1));
```

```
unpacked[parent[this],leftP]:=false;
      call updateCountRec(parent[this]);
assert parentPred(parent[this], left, right, parent, count, frac);
assert unpacked[parent[this], parentP]==false;
//this is the final assertion needed
assert parentPred(this, left, right, parent, count, frac);
unpacked[this, parentP]:=false;
      }
    else
  {
assert (frac[this, countP] >= 50);
fracLocal[this, countP]:=frac[this, countP];
assert  parent[this]==null ==> (frac[this, countP] >= 50);
fracLocal[this, countP]:=fracLocal[this, countP] + frac[this, countP];
frac[this, countP]:=fracLocal[this, countP];
assert (frac[this, countP] >= 100);
        call updateCount(this);
assert parentPred(this, left, right, parent, count, frac);
unpacked[this, parentP]:=false;
  }
 }

procedure updateCount(this: Ref)
modifies count, frac, unpacked;
requires this != null;
requires unpacked[this, leftP]==false;
requires unpacked[this, rightP]==false;
requires (frac[this,leftP] >=50) &&
         (frac[this,rightP] >=50) &&
         (frac[this, countP] >= 100);

ensures ( exists c:int ::
         countPred(this, left, right, count, frac, c)
        );
ensures unpacked[this, countP]==false;
ensures (frac[this, countP]>=100);
ensures (forall r:Ref:: (r!=this)
       ==> (unpacked[r, countP]==old(unpacked[r, countP])));
{
var newc:int;
newc := 1;

//we can assume this because leftP and rightP are unpacked
assume (exists c1:int :: leftPred(this, left, right, count, frac, c1));
assume (exists c2:int :: rightPred(this, left, right, count, frac, c2));

assert leftPred(this, left, right, count, frac, count[left[this]]);
assert rightPred(this, left, right, count, frac, count[right[this]]);

    if (left[this] != null)
```

```
    {
    newc := newc + count[left[this]];
    }

    if (right[this] != null)
    {
    newc := newc + count[right[this]];
    }

    count[this] := newc;
assert leftPred(this, left, right, count,
       frac, count[left[this]]);
assert rightPred(this, left, right, count,
       frac, count[right[this]]);
assert (newc==1+count[left[this]]+ count[right[this]]);
assert (count[this]==1+count[left[this]]+ count[right[this]]);
assert (frac[this, leftP] >= 50) ;
assert (frac[this, rightP] >= 50);
assert count[this]==newc;
assert this!=null;
assert countPred(this,left, right, count, frac, newc);
unpacked[this, countP]:=false;
}

procedure setLeft(this: Ref, l:Ref)
modifies parent, left, count, frac, unpacked;
requires this!=null;
requires this!=l;
requires l!=null;
requires unpacked[this, parentP]==false;
requires unpacked[l, parentP]==false;
requires (forall r:Ref:: (unpacked[r, countP]==false));
ensures parentPred(this, left, right, parent, count, frac);
 {
//we can assume these becase parentPred is true for this and l
assume parentPred(this, left, right, parent, count, frac);
assume parentPred(l,left, right, parent, count, frac);

unpacked[this, parentP]:=true;
//we can assume the following because we have
//just unpacked this from parentP
assume parent[this]!=null
       ==> (frac[parent[this], parentP]>0);
assume (parent[this]!=null) &&
       (this==right[parent[this]])
       ==> (frac[parent[this], rightP]>=50);
assume (parent[this]!=null) &&
       (this==left[parent[this]])
       ==> (frac[parent[this], leftP]>=50);
assume parent[this]==null
```

```
       ==> (frac[this, countP] >= 50);

assert (exists c: int ::
       rightPred(this, left, right, count, frac, c) );
assert rightPred(this, left, right, count,
       frac, count[right[this]]) ;
assert (exists c: int ::
       countPred(l, left, right, count, frac, c) );
assert countPred(l, left, right, count, frac, count[l]);

   parent[l]:=this;
   left[this]:=l;
assert this==parent[l];
//proving the assertions for updateCountRec
assert this!=null;
assert  unpacked[this, parentP];

assert left[this] != null;
assert countPred(left[this], left, right,
       count, frac, count[left[this]]);
assert frac[left[this], countP] >= 50;
assert leftPred(this, left, right, count,
       frac, count[left[this]]);
unpacked[this, leftP]:=false;

assert rightPred(this, left, right, count,
       frac, count[right[this]]) ;
assert (exists c2:int ::
       rightPred(this, left, right, count, frac, c2));
unpacked[this, rightP]:=false;

assert (frac[this,leftP] >=50) &&
       (frac[this,rightP] >=50);
assert (frac[this, countP] >= 50);
unpacked[this, countP]:=true;
assert (forall r:Ref:: (r!=this)
       ==> (unpacked[r, countP]==false));

assert this!=null;
assert unpacked[this, parentP];

   call updateCountRec(this);

}

}
```

11.3 Manual verification of Composite pattern

Below we display the manual verification of our instance of the composite pattern, using the object propositions methodology.

```
class Composite {
  private Composite left, right, parent;
  private int count;

  public Composite
  ⊸ this@½ parent() ⊗ this@½ left(null, 0) ⊗ this@½ right(null, 0)
   {
    this.count = 1;
    { this.count → 1 }
    this.left = null;
    { this.left → null ⊗ this.count → 1 }
    this.right = null;
    { this.right → null ⊗ this.left → null ⊗ this.count → 1 }
    this.parent = null;
    { this.parent → null ⊗ this.right → null ⊗ this.left → null ⊗
this.count → 1 }
    pack this@1 right(null, 0)
    { this.parent → null ⊗ this.left → null ⊗ this.count → 1 ⊗
this@1 right(null, 0) }
    pack this@1 left(null, 0)
    { this.parent → null ⊗ this.count → 1 ⊗
this@1 right(null, 0) ⊗ this@1 left(null, 0) }
    split 1 into two halfs for left, right
    pack this@1 count(0)
    { this.parent → null ⊗
this@½ right(null, 0) ⊗ this@½ left(null, 0) ⊗ this@1 count(0) }
    split 1 into two halfs for count
    pack this@1 parent()
    { this@½ right(null, 0) ⊗ this@½ left(null, 0) ⊗ this@1 parent() }
    split 1 into halfs
    { this@½ right(null, 0) ⊗ this@½ left(null, 0) ⊗
this@½ parent() ⊗ this@½ parent() }
    we only need one half of parent in the post-condition
    { this@½ right(null, 0) ⊗ this@½ left(null, 0) ⊗ this@½ parent() }
    { QED }
   }

  private void updateCountRec ()
  ∃ k1, opp, lcc, k, ol, lc, or, rc.
(unpacked(this@ k1 parent()) ⊗
this.parent → opp ⊗ opp ≠ this ⊗
```

$\Big($(opp $\neq$ null $\multimap$ opp@k parent() $\otimes$
(opp@$\frac{1}{2}$ left(this, lcc) $\oplus$ opp@$\frac{1}{2}$ right(this, lcc))) $\oplus$
(opp = null $\multimap$ this@$\frac{1}{2}$ count(lcc)) $\Big)$ $\otimes$

unpacked(this@$\frac{1}{2}$ count(lcc)) $\otimes$
this@$\frac{1}{2}$ left(ol, lc) $\otimes$ this@$\frac{1}{2}$ right(or, rc)

$\multimap$ $\exists$ k1.this@$k1$ parent())
{
if (this.parent != null)
{ $\exists$ k1,lcc,opp, k.
unpacked(this@k1 parent()) $\otimes$
this.parent $\rightarrow$ opp $\otimes$ opp $\neq$ this $\otimes$
opp@k parent() $\otimes$
(opp@$\frac{1}{2}$ left(this, lcc) $\oplus$ opp@$\frac{1}{2}$ right(this, lcc)) $\otimes$

unpacked(this@$\frac{1}{2}$ count(lcc)) $\otimes$
$\exists$ ol, lc, or, rc. $\otimes$
this@$\frac{1}{2}$ left(ol, lc) $\otimes$ this@$\frac{1}{2}$ right(or, rc) }

split the fraction k of opp in parent and unpack opp from parent()
{ $\exists$ k1,lcc,opp, k.
unpacked(this@k1 parent()) $\otimes$
this.parent $\rightarrow$ opp $\otimes$ opp $\neq$ this $\otimes$
unpacked(opp@$\frac{k}{2}$ parent()) $\otimes$ $\exists$ oppp, lccc, kk. opp.parent $\rightarrow$ oppp $\otimes$ opp $\neq$ oppp
$\otimes$ opp@$\frac{1}{2}$ count(lccc) $\otimes$
$\Big($(oppp $\neq$ null $\multimap$ oppp@kk parent() $\otimes$ (oppp@$\frac{1}{2}$ left(opp, lccc) $\oplus$ oppp@$\frac{1}{2}$
right(opp, lccc))) $\oplus$
(oppp = null $\multimap$ opp@$\frac{1}{2}$ count(lccc))$\Big)$ $\otimes$

(opp@$\frac{1}{2}$ left(this, lcc) $\oplus$ opp@$\frac{1}{2}$ right(this, lcc)) $\otimes$
$\otimes$ opp@$\frac{k}{2}$ parent() $\otimes$

unpacked(this@$\frac{1}{2}$ count(lcc)) $\otimes$
$\exists$ ol, lc, or rc. $\otimes$
this@$\frac{1}{2}$ left(ol, lc) $\otimes$ this@$\frac{1}{2}$ right(or, rc)

unpack opp from $\frac{1}{2}$ count(lccc)
{$\exists$ k1, opp, lcc, k.
unpacked(this@k1 parent()) $\otimes$
this.parent $\rightarrow$ opp $\otimes$ opp $\neq$ this $\otimes$

unpacked(opp@$\frac{k}{2}$ parent()) $\otimes$ $\exists$ oppp, lccc, kk. opp.parent $\rightarrow$ oppp $\otimes$ opp $\neq$ oppp
$\otimes$

unpacked(opp@$\frac{1}{2}$ count(lccc)) $\otimes$ $\exists$ oll, orr, llc, rrc. $\otimes$ opp@$\frac{1}{2}$ left(oll, llc) $\otimes$
opp@$\frac{1}{2}$ right(orr, rrc) $\otimes$

$\Big($(oppp $\neq$ null $\multimap$ oppp@kk parent() $\otimes$ (oppp@$\frac{1}{2}$ left(opp, lccc) $\oplus$ oppp@$\frac{1}{2}$
right(opp, lccc))) $\oplus$
(oppp = null $\multimap$ opp@$\frac{1}{2}$ count(lccc))$\Big)$ $\otimes$
(opp@$\frac{1}{2}$ left(this, lcc) $\oplus$ opp@$\frac{1}{2}$ right(this, lcc)) $\otimes$
$\otimes$ opp@$\frac{k}{2}$ parent() $\otimes$

unpacked(this@$\frac{1}{2}$ count(lcc)) $\otimes$
$\exists$ ol, lc, or, rc.
this.count $\rightarrow$ lcc $\otimes$
lcc = lc + rc + 1 $\otimes$
this@$\frac{1}{2}$ right(or, rc) $\otimes$this@$\frac{1}{2}$ left(ol, lc) }

this is either the right or left child of opp. We analyze both cases.
1.In the first case we assume it's the right child.
instantiate orr = this, rrc = lcc; merge both $\frac{1}{2}$ to opp in right
{$\exists$ k1, opp, lcc, k.
unpacked(this@k1 parent()) $\otimes$
this.parent $\rightarrow$ opp $\otimes$ opp $\neq$ this $\otimes$
unpacked(opp@$\frac{k}{2}$ parent()) $\otimes$ $\exists$ oppp, lccc, kk. opp.parent $\rightarrow$ oppp $\otimes$ opp $\neq$ oppp
$\otimes$

unpacked(opp@$\frac{1}{2}$ count(lccc)) $\otimes$ $\exists$ oll, llc. opp@$\frac{1}{2}$ left(oll, llc) $\otimes$

$\Big($(oppp $\neq$ null $\multimap$ oppp@kk parent() $\otimes$ (oppp@$\frac{1}{2}$ left(opp, lccc) $\oplus$ oppp@$\frac{1}{2}$
right(opp, lccc))) $\oplus$
(oppp = null $\multimap$ opp@$\frac{1}{2}$ count(lccc))$\Big)$ $\otimes$
$\otimes$ opp@$\frac{k}{2}$ parent() $\otimes$

opp@1 right(this, lcc) $\otimes$

unpacked(this@$\frac{1}{2}$ count(lcc)) $\otimes$
$\exists$ ol, lc, or, rc.
this@$\frac{1}{2}$ right(or, rc) $\otimes$this@$\frac{1}{2}$ left(ol, lc) }

unpack opp from right(this, lcc)

{∃ k1, opp, lcc, k.
unpacked(this@k1 parent()) ⊗
this.parent → opp ⊗ opp ≠ this ⊗
unpacked(opp@$\frac{k}{2}$ parent()) ⊗ ∃ oppp, lccc, kk. opp.parent → oppp ⊗ opp ≠ oppp
⊗

unpacked(opp@$\frac{1}{2}$ count(lccc)) ⊗ ∃ oll, llc. opp@$\frac{1}{2}$ left(oll, llc) ⊗

$\Big($(oppp ≠ null ⊸ oppp@kk parent() ⊗ (oppp@$\frac{1}{2}$ left(opp, lccc) ⊕ oppp@$\frac{1}{2}$
right(opp, lccc))) ⊕
(oppp = null ⊸ opp@$\frac{1}{2}$ count(lccc))$\Big)$ ⊗
⊗ opp@$\frac{k}{2}$ parent() ⊗

unpacked(opp@1 right(this, lcc)) ⊗
opp.right → this ⊗ this@$\frac{1}{2}$ count(lcc) ⊗

unpacked(this@$\frac{1}{2}$ count(lcc)) ⊗
∃ ol, lc, or, rc.
this@$\frac{1}{2}$ right(or, rc) ⊗this@$\frac{1}{2}$ left(ol, lc) }

pack this @ $\frac{1}{2}$ count(lcc), add it to the other half
then unpack the count predicate
{∃ k1, opp, lcc, k.
unpacked(this@k1 parent()) ⊗
this.parent → opp ⊗ opp ≠ this ⊗
unpacked(opp@$\frac{k}{2}$ parent()) ⊗ ∃ oppp, lccc, kk. opp.parent → oppp ⊗ opp ≠ oppp
⊗

unpacked(opp@$\frac{1}{2}$ count(lccc)) ⊗ ∃ oll, llc. opp.count → lccc ⊗ lccc = llc + lcc + 1
⊗ opp@$\frac{1}{2}$ left(oll, llc) ⊗

$\Big($(oppp ≠ null ⊸ oppp@kk parent() ⊗ (oppp@$\frac{1}{2}$ left(opp, lccc) ⊕ oppp@$\frac{1}{2}$
right(opp, lccc))) ⊕
(oppp = null ⊸ opp@$\frac{1}{2}$ count(lccc))$\Big)$ ⊗
⊗ opp@$\frac{k}{2}$ parent() ⊗

unpacked(opp@1 right(this, lcc)) ⊗
opp.right → this ⊗

unpacked(this@1 count(lcc)) ⊗
∃ ol, lc, or, rc.
this@$\frac{1}{2}$ right(or, rc) ⊗this@$\frac{1}{2}$ left(ol, lc) }

this.updateCount();
{∃ k1, opp, lcc, k.
unpacked(this@k1 parent()) ⊗
this.parent → opp ⊗ opp ≠ this ⊗
unpacked(opp@$\frac{k}{2}$ parent()) ⊗ ∃ oppp, lccc, kk. opp.parent → oppp ⊗ opp ≠ oppp
⊗

unpacked(opp@$\frac{1}{2}$ count(lccc)) ⊗ ∃ oll, llc. opp.count → lccc ⊗ lccc = llc + lcc + 1
⊗ opp@$\frac{1}{2}$ left(oll, llc) ⊗

$\Big($(oppp ≠ null ⊸ oppp@kk parent() ⊗ (oppp@$\frac{1}{2}$ left(opp, lccc) ⊕ oppp@$\frac{1}{2}$
right(opp, lccc))) ⊕
(oppp = null ⊸ opp@$\frac{1}{2}$ count(lccc))$\Big)$ ⊗
⊗ opp@$\frac{k}{2}$ parent() ⊗

unpacked(opp@1 right(this, lcc)) ⊗
opp.right → this ⊗

this@1 count(lcc) }

pack opp in right(this, lcc)
{∃ k1, opp, lcc, k.
unpacked(this@k1 parent()) ⊗
this.parent → opp ⊗ opp ≠ this ⊗
unpacked(opp@$\frac{k}{2}$ parent()) ⊗ ∃ oppp, lccc, kk. opp.parent → oppp ⊗ opp ≠ oppp
⊗

unpacked(opp@$\frac{1}{2}$ count(lccc)) ⊗ ∃ oll, llc. opp.count → lccc ⊗ lccc = llc + lcc + 1
⊗ opp@$\frac{1}{2}$ left(oll, llc) ⊗

$\Big($(oppp ≠ null ⊸ oppp@kk parent() ⊗ (oppp@$\frac{1}{2}$ left(opp, lccc) ⊕ oppp@$\frac{1}{2}$
right(opp, lccc))) ⊕
(oppp = null ⊸ opp@$\frac{1}{2}$ count(lccc))$\Big)$ ⊗
⊗ opp@$\frac{k}{2}$ parent() ⊗

opp@1 right(this, lcc) ⊗

this@$\frac{1}{2}$ count(lcc) }

this.parent.updateCountRec();
{∃ k1, opp, lcc, k, k3.
unpacked(this@k1 parent()) ⊗
this.parent → opp ⊗ opp ≠ this ⊗
unpacked(opp@$\frac{k}{2}$ parent()) ⊗ ∃ oppp, lccc, kk. opp.parent → oppp ⊗ opp ≠ oppp
⊗

$\Big(\big($opp ≠ null ⊸ oppp@kk parent() ⊗ (oppp@$\frac{1}{2}$ left(opp, lccc) ⊕ oppp@$\frac{1}{2}$
right(opp, lccc))$\big)$ ⊕
(opp = null ⊸ opp@$\frac{1}{2}$ count(lccc))$\Big)$ ⊗

opp@$\frac{1}{2}$ right(this, lcc) ⊗

this@$\frac{1}{2}$ count(lcc) ⊗
opp@k3 parent() }

pack opp in parent()
{∃ k1, opp, lcc, k, k3.
unpacked(this@k1 parent()) ⊗
this.parent → opp ⊗ opp ≠ this ⊗
unpacked(opp@$\frac{k}{2}$ parent()) ⊗ ∃ oppp, lccc, kk. opp.parent → oppp ⊗ opp ≠ oppp
⊗

$\Big(\big($opp ≠ null ⊸ oppp@kk parent() ⊗ (oppp@$\frac{1}{2}$ left(opp, lccc) ⊕ oppp@$\frac{1}{2}$
right(opp, lccc))$\big)$ ⊕
(opp = null ⊸ opp@$\frac{1}{2}$ count(lccc))$\Big)$ ⊗

opp@$\frac{1}{2}$ right(this, lcc) ⊗

this@$\frac{1}{2}$ count(lcc) ⊗
opp@k3 parent() }

pack this in parent(), assuming opp is not null.
It's not since we just called updateCountRec on it, we are on that branch.
{∃ k1. this@k1 parent()}
QED

2.In the second case we assume it's the left child.
instantiate oll = this, llc = lcc; merge both $\frac{1}{2}$ to opp in right
{∃ k1, opp, lcc, k.
unpacked(this@k1 parent()) ⊗
this.parent → opp ⊗ opp ≠ this ⊗

$\text{unpacked}(\text{opp}@\frac{k}{2}\ \text{parent}()) \otimes \exists\ \mathsf{oppp},\ \mathsf{lccc},\ \mathsf{kk}.\ \mathsf{opp.parent} \rightarrow \mathsf{oppp} \otimes \mathsf{opp} \neq \mathsf{oppp}$
$\otimes$

$\text{unpacked}(\text{opp}@\frac{1}{2}\ \text{count}(\text{lccc})) \otimes \exists\ \text{orr}, \text{rrc}.\ \mathsf{opp.count} \rightarrow \mathsf{lccc} \otimes \mathsf{lccc} = \mathsf{lcc} + \mathsf{rrc} + 1$
$\otimes\ \text{opp}@\frac{1}{2}\ \text{right}(\text{orr}, \text{rrc}) \otimes$

$\Big(\big(\mathsf{oppp} \neq \texttt{null} \multimap \text{oppp@kk parent}() \otimes (\text{oppp}@\frac{1}{2}\ \text{left}(\text{opp}, \text{lccc}) \oplus \text{oppp}@\frac{1}{2}$
$\text{right}(\text{opp}, \text{lccc}))\big) \oplus$
$(\mathsf{oppp} = \mathsf{null} \multimap \text{opp}@\frac{1}{2}\ \text{count}(\text{lccc}))\Big) \otimes$
$\otimes\ \ \text{opp}@\frac{k}{2}\ \text{parent}() \otimes$

$\text{opp}@1\ \text{left}(\mathsf{this}, \mathsf{lcc}) \otimes$

$\text{unpacked}(\text{this}@\frac{1}{2}\ \text{count}(\text{lcc})) \otimes$
$\exists\ \mathsf{ol}, \mathsf{lc}, \mathsf{or}, \mathsf{rc}.$
$\mathsf{this.count} \rightarrow \mathsf{lcc} \otimes$
$\mathsf{lcc} = \mathsf{lc} + \mathsf{rc} + 1 \otimes$
$\text{this}@\frac{1}{2}\ \text{right}(\text{or}, \text{rc}) \otimes \text{this}@\frac{1}{2}\ \text{left}(\mathsf{ol}, \mathsf{lc})\ \}$

unpack opp from left(this, lcc)
{$\exists$ k1, opp, lcc, k.
$\text{unpacked}(\text{this}@\text{k1 parent}()) \otimes$
$\mathsf{this.parent} \rightarrow \mathsf{opp} \otimes \mathsf{opp} \neq \mathsf{this} \otimes$
$\text{unpacked}(\text{opp}@\frac{k}{2}\ \text{parent}()) \otimes \exists\ \mathsf{oppp},\ \mathsf{lccc},\ \mathsf{kk}.\ \mathsf{opp.parent} \rightarrow \mathsf{oppp} \otimes \mathsf{opp} \neq \mathsf{oppp}$
$\otimes$

$\text{unpacked}(\text{opp}@\frac{1}{2}\ \text{count}(\text{lccc})) \otimes \exists\ \text{orr}, \text{rrc}.\ \mathsf{opp.count} \rightarrow \mathsf{lccc} \otimes \mathsf{lccc} = \mathsf{rrc} + \mathsf{lcc} + 1$
$\otimes\ \text{opp}@\frac{1}{2}\ \text{right}(\text{orr}, \text{rrc}) \otimes$

$\Big(\big(\mathsf{oppp} \neq \texttt{null} \multimap \text{oppp@kk parent}() \otimes (\text{oppp}@\frac{1}{2}\ \text{left}(\text{opp}, \text{lccc}) \oplus \text{oppp}@\frac{1}{2}$
$\text{right}(\text{opp}, \text{lccc}))\big) \oplus$
$(\mathsf{oppp} = \mathsf{null} \multimap \text{opp}@\frac{1}{2}\ \text{count}(\text{lccc}))\Big) \otimes$
$\otimes\ \ \text{opp}@\frac{k}{2}\ \text{parent}() \otimes$

$\text{unpacked}(\text{opp}@1\ \text{left}(\text{this}, \text{lcc})) \otimes$
$\mathsf{opp.left} \rightarrow \mathsf{this} \otimes \text{this}@\frac{1}{2}\ \text{count}(\text{lcc}) \otimes$

$\text{unpacked}(\text{this}@\frac{1}{2}\ \text{count}(\text{lcc})) \otimes$
$\exists\ \mathsf{ol}, \mathsf{lc}, \mathsf{or}, \mathsf{rc}.$
$\mathsf{this.count} \rightarrow \mathsf{lcc} \otimes$
$\mathsf{lcc} = \mathsf{lc} + \mathsf{rc} + 1 \otimes$
$\text{this}@\frac{1}{2}\ \text{right}(\text{or}, \text{rc}) \otimes \text{this}@\frac{1}{2}\ \text{left}(\mathsf{ol}, \mathsf{lc})\ \}$

pack this @ $\frac{1}{2}$ count(lcc), add it to the other half
then unpack the count predicate
{$\exists$ k1, opp, lcc, k.
unpacked(this@k1 parent()) $\otimes$
this.parent $\rightarrow$ opp $\otimes$ opp $\neq$ this $\otimes$
unpacked(opp@$\frac{k}{2}$ parent()) $\otimes$ $\exists$ oppp, lccc, kk. opp.parent $\rightarrow$ oppp $\otimes$ opp $\neq$ oppp
$\otimes$

unpacked(opp@$\frac{1}{2}$ count(lccc)) $\otimes$ $\exists$ orr, rrc. opp.count $\rightarrow$ lccc $\otimes$ lccc = rrc + lcc + 1
$\otimes$ opp@$\frac{1}{2}$ right(orr, rrc) $\otimes$

$\Big(\big($oppp $\neq$ null $\multimap$ oppp@kk parent() $\otimes$ (oppp@$\frac{1}{2}$ left(opp, lccc) $\oplus$ oppp@$\frac{1}{2}$ right(opp, lccc))$\big)$ $\oplus$
(oppp = null $\multimap$ opp@$\frac{1}{2}$ count(lccc))$\Big)$ $\otimes$
$\otimes$ opp@$\frac{k}{2}$ parent() $\otimes$

unpacked(opp@1 left(this, lcc)) $\otimes$
opp.left $\rightarrow$ this $\otimes$

unpacked(this@1 count(lcc)) $\otimes$
$\exists$ ol, lc, or, rc.
this.count $\rightarrow$ lcc $\otimes$
lcc = lc + rc + 1 $\otimes$
this@$\frac{1}{2}$ right(or, rc) $\otimes$this@$\frac{1}{2}$ left(ol, lc) }

this.updateCount();
{$\exists$ k1, opp, lcc, k.
unpacked(this@k1 parent()) $\otimes$
this.parent $\rightarrow$ opp $\otimes$ opp $\neq$ this $\otimes$
unpacked(opp@$\frac{k}{2}$ parent()) $\otimes$ $\exists$ oppp, lccc, kk. opp.parent $\rightarrow$ oppp $\otimes$ opp $\neq$ oppp
$\otimes$

unpacked(opp@$\frac{1}{2}$ count(lccc)) $\otimes$ $\exists$ orr, rrc. opp.count $\rightarrow$ lccc $\otimes$ lccc = rrc + lcc + 1
$\otimes$ opp@$\frac{1}{2}$ right(orr, rrc) $\otimes$

$\Big(\big($oppp $\neq$ null $\multimap$ oppp@kk parent() $\otimes$ (oppp@$\frac{1}{2}$ left(opp, lccc) $\oplus$ oppp@$\frac{1}{2}$ right(opp, lccc))$\big)$ $\oplus$
(oppp = null $\multimap$ opp@$\frac{1}{2}$ count(lccc))$\Big)$ $\otimes$
$\otimes$ opp@$\frac{k}{2}$ parent() $\otimes$

unpacked(opp@1 left(this, lcc)) $\otimes$
opp.left $\rightarrow$ this $\otimes$

this@1 count(lcc) }

pack opp in left(this, lcc)
{$\exists$ k1, opp, lcc, k.
unpacked(this@k1 parent()) $\otimes$
this.parent $\rightarrow$ opp $\otimes$ opp $\neq$ this $\otimes$
unpacked(opp@$\frac{k}{2}$ parent()) $\otimes$ $\exists$ oppp, lccc, kk. opp.parent $\rightarrow$ oppp $\otimes$ opp $\neq$ oppp
$\otimes$

unpacked(opp@$\frac{1}{2}$ count(lccc)) $\otimes$ $\exists$ orr, rrc. opp.count $\rightarrow$ lccc $\otimes$ lccc = rrc + lcc + 1
$\otimes$ opp@$\frac{1}{2}$ right(orr, rrc) $\otimes$

$\Big($(oppp $\neq$ null $\multimap$ oppp@kk parent() $\otimes$ (oppp@$\frac{1}{2}$ left(opp, lccc) $\oplus$ oppp@$\frac{1}{2}$
right(opp, lccc))) $\oplus$
(oppp = null $\multimap$ opp@$\frac{1}{2}$ count(lccc))$\Big)$ $\otimes$
$\otimes$ opp@$\frac{k}{2}$ parent() $\otimes$

opp@1 left(this, lcc) $\otimes$

this@$\frac{1}{2}$ count(lcc) }

this.parent.updateCountRec();

{$\exists$ k1, opp, lcc, k, k3.
unpacked(this@k1 parent()) $\otimes$
this.parent $\rightarrow$ opp $\otimes$ opp $\neq$ this $\otimes$
unpacked(opp@$\frac{k}{2}$ parent()) $\otimes$ $\exists$ oppp, lccc, kk. opp.parent $\rightarrow$ oppp $\otimes$ opp $\neq$ oppp
$\otimes$

$\Big($(oppp $\neq$ null $\multimap$ oppp@kk parent() $\otimes$ (oppp@$\frac{1}{2}$ left(opp, lccc) $\oplus$ oppp@$\frac{1}{2}$
right(opp, lccc))) $\oplus$
(oppp = null $\multimap$ opp@$\frac{1}{2}$ count(lccc))$\Big)$ $\otimes$

opp@$\frac{1}{2}$ left(this, lcc) $\otimes$

this@$\frac{1}{2}$ count(lcc) $\otimes$
opp@k3 parent()}
pack this in parent(), assuming opp is not null.

```
It's not since we just called updateCountRec on it, we are on that branch.
        {∃ k1. this@k1 parent()}
          QED
       else


          { unpacked(this@k1 parent()) ⊗ ∃ opp, lcc. unpacked(this@½ count(lcc))
⊗
∃ ol, lc. this@½ left(l, lc) ⊗
∃ or, rc, lc1. this.count → lcc ⊗
lcc = lc1 + rc + 1 ⊗
this@½ right(or, rc) ⊗
this.parent → opp ⊗
opp ≠ this ⊗ opp = null ⊗ this@½ count(lcc) }


          merge this, ½ in count(lcc) from packed and unpacked
          { unpacked(this@k1 parent()) ⊗ ∃ opp, lcc. unpacked(this@1 count(lcc))
⊗
∃ ol, lc. this@½ left(l, lc) ⊗
∃ or, rc, lc1. this.count → lcc ⊗
lcc = lc1 + rc + 1 ⊗
this@½ right(or, rc) ⊗
this.parent → opp ⊗
opp ≠ this ⊗ opp = null }


          this.updateCount();


{ unpacked(this@k1 parent()) ⊗ ∃ opp, lcc. this@1 count(lcc) ⊗
this.parent → opp ⊗
opp ≠ this ⊗ opp = null }

          split count in half and pack this in parent
          {∃ k1. this@k1 parent() }
          QED
   }

   private void updateCount()
   ∃ c, c1, c2, ol, or. unpacked(this@1 count(c)) ⊗
this@½ left(ol, c1) ⊗this@½ right(or, c2)
—∘ ∃ c. this@1 count(c)
    {
      int newc = 1;
      unpack this @½left(ol,c1)
```

$\{ \mathsf{newc} = 1 \otimes$
$\mathrm{unpacked}(\mathrm{this}@1\ \mathrm{count}(c)) \otimes$
$\mathrm{unpacked}(\mathrm{this}@\frac{1}{2}\ \mathrm{left}(\mathrm{ol}, c1)) \otimes$
$\mathsf{this.left} \rightarrow \mathsf{ol} \otimes (\mathsf{ol} = \mathsf{null} \multimap c1 = 0) \otimes$
$(\mathsf{ol} \neq \mathsf{null} \multimap \mathrm{ol}@\frac{1}{2}\ \mathrm{count}(c1)) \otimes$
$\mathrm{this}@\frac{1}{2}\ \mathrm{right}(\mathrm{or}, c2)\ \}$
`if (this.left != null)`
$\{ \mathsf{newc} = 1 \otimes$
$\mathrm{unpacked}(\mathrm{this}@1\ \mathrm{count}(c)) \otimes$
$\mathrm{unpacked}(\mathrm{this}@\frac{1}{2}\ \mathrm{left}(\mathrm{ol}, c1)) \otimes$
$\mathsf{this.left} \rightarrow \mathsf{ol} \otimes$
$\mathrm{ol}@\frac{1}{2}\ \mathrm{count}(c1) \otimes$
$\mathrm{this}@\frac{1}{2}\ \mathrm{right}(\mathrm{or}, c2)\ \}$
$\mathsf{unpack\ ol\ in\ } \frac{1}{2}\ \mathsf{count(c1)}$
$\{ \mathsf{newc} = 1 \otimes$
$\mathrm{unpacked}(\mathrm{this}@1\ \mathrm{count}(c)) \otimes$
$\mathrm{unpacked}(\mathrm{this}@\frac{1}{2}\ \mathrm{left}(\mathrm{ol}, c1)) \otimes$
$\mathsf{this.left} \rightarrow \mathsf{ol} \otimes$
$\mathrm{unpacked}(\mathrm{ol}@\frac{1}{2}\ \mathrm{count}(c1)) \otimes$
$\exists\ \mathsf{lol,lor,llc,lrc.\ ol.count} \rightarrow c1 \otimes$
$c1 = \mathrm{llc} + \mathrm{lrc} + 1 \otimes$
$\mathrm{ol}@\frac{1}{2}\ \mathrm{left}(\mathrm{lol}, \mathrm{llc}) \otimes$
$\mathrm{ol}@\frac{1}{2}\ \mathrm{right}(\mathrm{lor}, \mathrm{lrc}) \otimes$
$\mathrm{this}@\frac{1}{2}\ \mathrm{right}(\mathrm{or}, c2) \otimes\}$
`newc = newc + left.count;`
$\{ \mathsf{newc} = 1 + c1 \otimes$
$\mathrm{unpacked}(\mathrm{this}@1\ \mathrm{count}(c)) \otimes$
$\mathrm{unpacked}(\mathrm{this}@\frac{1}{2}\ \mathrm{left}(\mathrm{ol}, c1)) \otimes$
$\mathsf{this.left} \rightarrow \mathsf{ol} \otimes$
$\mathrm{unpacked}(\mathrm{ol}@\frac{1}{2}\ \mathrm{count}(c1)) \otimes$
$\exists\ \mathsf{lol,lor,llc,lrc.\ ol.count} \rightarrow c1 \otimes$
$c1 = \mathrm{llc} + \mathrm{lrc} + 1 \otimes$
$\mathrm{ol}@\frac{1}{2}\ \mathrm{left}(\mathrm{lol}, \mathrm{llc}) \otimes\ \mathrm{ol}@\frac{1}{2}\ \mathrm{right}(\mathrm{lor}, \mathrm{lrc}) \otimes$
$\mathrm{this}@\frac{1}{2}\ \mathrm{right}(\mathrm{or}, c2)\ \}$
$\mathsf{pack\ ol\ in\ count(c1)}$
$\{ \mathsf{newc} = 1 + \mathrm{lc} \otimes$
$\mathrm{unpacked}(\mathrm{this}@1\ \mathrm{count}(c)) \otimes$
$\mathrm{unpacked}(\mathrm{this}@\frac{1}{2}\ \mathrm{left}(\mathrm{ol}, c1)) \otimes$
$\mathsf{this.left} \rightarrow \mathsf{ol} \otimes$
$\mathrm{ol}@\frac{1}{2}\ \mathrm{count}(c1) \otimes$
$\mathrm{this}@\frac{1}{2}\ \mathrm{right}(\mathrm{or}, c2)\}$
$\mathsf{pack\ this\ in\ left(ol,\ c1)}$
$\{ \mathsf{newc} = 1 + c1 \otimes$
$\mathrm{unpacked}(\mathrm{this}@1\ \mathrm{count}(c)) \otimes$

this@$\frac{1}{2}$ left(ol, c1) $\otimes$
this@$\frac{1}{2}$ right(or, c2) }
unpack this in $\frac{1}{2}$ right(or, c2)
{ newc = 1 + c1 $\otimes$
unpacked(this@1 count(c)) $\otimes$
this@$\frac{1}{2}$ left(ol, c1) $\otimes$
unpacked(this@$\frac{1}{2}$ right(or, c2)) $\otimes$
this.right $\rightarrow$ or $\otimes$
((or $\neq$ null $\multimap$ or@$\frac{1}{2}$ count(c2)) $\oplus$
(or = null $\multimap$ c2 = 0)) }
if (this.right != null)
{ newc = 1 + c1 $\otimes$
unpacked(this@1 count(c)) $\otimes$
this@$\frac{1}{2}$ left(ol, c1) $\otimes$
unpacked(this@$\frac{1}{2}$ right(or, c2)) $\otimes$
this.right $\rightarrow$ or $\otimes$
or@$\frac{1}{2}$ count(c2) }
unpack or in $\frac{1}{2}$ count(c2)
{ newc = 1 + c1 $\otimes$
unpacked(this@1 count(c)) $\otimes$
this@$\frac{1}{2}$ left(ol, c1) $\otimes$
unpacked(this@$\frac{1}{2}$ right(or, c2)) $\otimes$
this.right $\rightarrow$ or $\otimes$
unpacked(or@$\frac{1}{2}$ count(c2)) $\otimes$
$\exists$ rol,ror,rlc,rrc. or.count $\rightarrow$ c2 $\otimes$
c2 = rlc + rrc + 1 $\otimes$
or@$\frac{1}{2}$ left(rol, rlc) $\otimes$
or@$\frac{1}{2}$ right(ror, rrc) }
newc = newc + right.count;
{ newc = 1 + c1 + c2 $\otimes$
unpacked(this@1 count(c)) $\otimes$
this@$\frac{1}{2}$ left(ol, c1) $\otimes$
unpacked(this@$\frac{1}{2}$ right(or, c2)) $\otimes$
this.right $\rightarrow$ or $\otimes$
unpacked(or@$\frac{1}{2}$ count(c2)) $\otimes$
$\exists$ rol,ror,rlc,rrc. or.count $\rightarrow$ c2 $\otimes$
c2 = rlc + rrc + 1 $\otimes$
or@$\frac{1}{2}$ left(rol, rlc) $\otimes$
or@$\frac{1}{2}$ right(ror, rrc) }
pack or in count
{ newc = 1 + c1 + c2 $\otimes$
unpacked(this@ 1 count(c)) $\otimes$
this@$\frac{1}{2}$ left(ol, c1) $\otimes$
unpacked(this@$\frac{1}{2}$ right(or, c2)) $\otimes$

this.right → or ⊗
or@$\frac{1}{2}$ count(c2) }
pack this in right
{ newc = 1 + c1 + c2 ⊗
unpacked(this@1 count(c)) ⊗
this@$\frac{1}{2}$ left(ol, c1) ⊗
this@$\frac{1}{2}$ right(or, c2) }
this.count = newc;
{ newc = 1 + c1 + c2 ⊗
unpacked(this@1 count(c)) ⊗
this.count → c ⊗ c = newc ⊗
this@$\frac{1}{2}$ left(ol, c1) ⊗
this@$\frac{1}{2}$ right(or, c2) }
pack this in count(newc)
{ this@1 count(newc) }
QED

}

public void setLeft (Composite l)
this ≠ l ⊗
∃$k1, k2$.(this@k1 parent() ⊗ l@k2 parent() ⊸
∃ k.this@k parent()
{
unpack l from parent
{ unpacked(l@k2 parent()) ⊗ ∃ op, lc, k,k1,k3. l.parent → op ⊗
op ≠ l ⊗ l@$\frac{1}{2}$ count(lc) ⊗
$\Big($(op ≠ null ⊸
op@k3 parent() ⊗
(op@$\frac{1}{2}$ left(l, lc) ⊕
op@$\frac{1}{2}$ right(l, lc))) ⊕
(op = null ⊸ l@$\frac{1}{2}$ count(lc)) $\Big)$ ⊗ this ≠ l ⊗
this@k1 parent() }
{ unpacked(l@k2 parent()) ⊗ ∃ lc, k1. l.parent → null ⊗
null ≠ l ⊗ l@$\frac{1}{2}$ count(lc) ⊗
l@$\frac{1}{2}$ count(lc) ⊗ this ≠ l ⊗
this@k1 parent() }
l.parent = this;
assignment rule
{ unpacked(l@k2 parent()) ⊗ ∃ lc. l.parent → this ⊗
null ≠ l ⊗ l@$\frac{1}{2}$ count(lc) ⊗
l@$\frac{1}{2}$ count(lc) ⊗ this ≠ l ⊗
this@k1 parent() }
unpack this from parent

{ unpacked(l@k2 parent()) ⊗ unpacked(this@k1 parent()) ⊗ ∃ opp, lcc, k,k4.
this.parent → opp ⊗
opp ≠ this ⊗ this@$\frac{1}{2}$ count(lcc) ⊗
$\Big($(opp ≠ null ⊸
opp@k4 parent() ⊗
(opp@$\frac{1}{2}$ left(this, lcc) ⊕
opp@$\frac{1}{2}$ right(this, lcc))) ⊕
(opp = null ⊸ this@$\frac{1}{2}$ count(lcc)) $\Big)$ ⊗
∃ lc. l.parent → this ⊗
null ≠ l ⊗ l@$\frac{1}{2}$ count(lc) ⊗
l@$\frac{1}{2}$ count(lc) ⊗ this ≠ l }
unpack this from $\frac{1}{2}$ count(lcc)
{ unpacked(l@k2 parent()) ⊗ unpacked(this@k1 parent()) ⊗ ∃ opp, lcc, k.
unpacked(this@$\frac{1}{2}$ count(lcc)) ⊗ ∃ ol, llc, or, rc. this.count → lcc ⊗
lcc = llc + rc + 1 ⊗
this@$\frac{1}{2}$ left(ol, llc) ⊗
this@$\frac{1}{2}$ right(or, rc) ⊗

this.parent → opp ⊗
opp ≠ this ⊗
$\Big($(opp ≠ null ⊸
opp@$\frac{k}{2}$ parent() ⊗
(opp@$\frac{1}{2}$ left(this, lcc) ⊕
opp@$\frac{1}{2}$ right(this, lcc))) ⊕
(opp = null ⊸ this@$\frac{1}{2}$ count(lcc)) $\Big)$ ⊗

∃ lc. l.parent → this ⊗
null ≠ l ⊗ l@$\frac{1}{2}$ count(lc) ⊗
l@$\frac{1}{2}$ count(lc) ⊗ this ≠ l ⊗
this@$\frac{1}{2}$ left(null, 0) }
existentialize ol with null and llc with 0 (to unify left permissions)
{ unpacked(l@k2 parent()) ⊗ unpacked(this@k1 parent()) ⊗ ∃ opp, lcc, k.
unpacked(this@$\frac{1}{2}$ count(lcc)) ⊗ ∃ or, rc. this.count → lcc ⊗
lcc = 0 + rc + 1 ⊗
this@$\frac{1}{2}$ right(or, rc) ⊗
this.parent → opp ⊗

opp ≠ this ⊗
$\Big($(opp ≠ null ⊸
opp@k4 parent() ⊗
(opp@$\frac{1}{2}$ left(this, lcc) ⊕
opp@$\frac{1}{2}$ right(this, lcc))) ⊕

$(\mathsf{opp} = \mathsf{null} \multimap \mathrm{this}@\frac{1}{2}\ \mathrm{count}(\mathsf{lcc}))\ \Big) \otimes$

$\exists$ lc. l.parent $\rightarrow$ this $\otimes$
null $\neq$ l $\otimes$ l@$\frac{1}{2}$ count(lc) $\otimes$
l@$\frac{1}{2}$ count(lc) $\otimes$ this $\neq$ l }
merge the half fractions to left
{ unpacked(l@k2 parent()) $\otimes$ unpacked(this@k1 parent()) $\otimes$ $\exists$ opp, lcc, k.
unpacked(this@$\frac{1}{2}$ count(lcc)) $\otimes$
this.left $\rightarrow$ null $\otimes$ $\exists$ or, rc. this.count $\rightarrow$ lcc $\otimes$
lcc = 0 + rc + 1 $\otimes$
this@$\frac{1}{2}$ right(or, rc) $\otimes$
this.parent $\rightarrow$ opp $\otimes$
opp $\neq$ this $\otimes$
$\Big($(opp $\neq$ null $\multimap$
opp@k4 parent() $\otimes$
(opp@$\frac{1}{2}$ left(this, lcc) $\oplus$
opp@$\frac{1}{2}$ right(this, lcc))) $\oplus$
$(\mathsf{opp} = \mathsf{null} \multimap \mathrm{this}@\frac{1}{2}\ \mathrm{count}(\mathsf{lcc}))\ \Big) \otimes$
$\exists$ lc. l.parent $\rightarrow$ this $\otimes$
null $\neq$ l $\otimes$ l@$\frac{1}{2}$ count(lc) $\otimes$
l@$\frac{1}{2}$ count(lc) $\otimes$ this $\neq$ l }
this.left = l;
assignment
{ unpacked(l@k2 parent()) $\otimes$ unpacked(this@k1 parent()) $\otimes$ $\exists$ opp, lcc, k.
unpacked(this@$\frac{1}{2}$ count(lcc)) $\otimes$
this.left $\rightarrow$ l $\otimes$ $\exists$ or, rc. this.count $\rightarrow$ lcc $\otimes$
lcc = lc + rc + 1 $\otimes$
this@$\frac{1}{2}$ right(or, rc) $\otimes$
this.parent $\rightarrow$ opp $\otimes$
opp $\neq$ this $\otimes$
$\Big($(opp $\neq$ null $\multimap$
opp@k4 parent() $\otimes$
(opp@$\frac{1}{2}$ left(this, lcc) $\oplus$
opp@$\frac{1}{2}$ right(this, lcc))) $\oplus$
$(\mathsf{opp} = \mathsf{null} \multimap \mathrm{this}@\frac{1}{2}\ \mathrm{count}(\mathsf{lcc}))\ \Big) \otimes$
$\exists$ lc. l.parent $\rightarrow$ this $\otimes$
null $\neq$ l $\otimes$ l@$\frac{1}{2}$ count(lc) $\otimes$
l@$\frac{1}{2}$ count(lc) $\otimes$ this $\neq$ l }
pack this in left(l, lc)
{ unpacked(l@k2 parent()) $\otimes$ unpacked(this@k1 parent()) $\otimes$ $\exists$ opp, lcc, k.
unpacked(this@$\frac{1}{2}$ count(lcc)) $\otimes$
$\exists$ lc. this@1 left(l, lc) $\otimes$
$\exists$ or, rc. this.count $\rightarrow$ lcc $\otimes$

```
lcc = lc + rc + 1 ⊗
this@½ right(or, rc) ⊗
this.parent → opp ⊗
opp ≠ this ⊗
(((opp ≠ null —∘
opp@k4 parent() ⊗
(opp@½ left(this, lcc) ⊕
opp@½ right(this, lcc))) ⊕
(opp = null —∘ this@½ count(lcc)) ) ⊗
l.parent → this ⊗
null ≠ l ⊗
l@½ count(lc) ⊗ this ≠ l }
      this.updateCountRec();
{ ∃ k1, k2, lc. unpacked(l@k2 parent()) ⊗
this@½ left(l, lc) ⊗
l.parent → this ⊗
null ≠ l ⊗
this@k1 parent()
l@½ count(lc) ⊗ this ≠ l }
    pack l in parent()
      {∃ k1, k2. l@k2 parent() ⊗ this@k1 parent()}
      QED
  }
}
```

12 Conclusion

We have introduced the novel abstraction *object proposition*, which uses abstract predicates to describe properties of objects, and fractions to describe the aliasing between objects. We used object propositions to write the specification and formally prove the correctness of an instance of the Composite pattern. We proved our system to be sound and highlighted the ways in which it improves the state of the art in the verification of object-oriented code.

References

1. Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *FMCO*, pages 364–387. Springer, 2005.
2. Mike Barnett, Robert DeLine, Manuel Fähndrich, K. Rustan M. Leino, and Wolfram Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology Special Issue: ECOOP 2003 workshop on Formal Techniques for Java-like Programs*, 3(6):27–56, June 2004.

3. Kevin Bierhoff and Jonathan Aldrich. Modular typestate checking of aliased objects. In *OOPSLA*, pages 301–320, 2007.
4. Kevin Bierhoff and Jonathan Aldrich. Permissions to specify the composite design pattern. In *Proc of SAVCBS 2008*, 2008.
5. John Boyland. Checking interference with fractional permissions. In *Static Analysis Symposium*, pages 55–72, 2003.
6. Ernie Cohen, Michal Moskal, Wolfram Schulte, and Stephan Tobies. Local verification of global invariants in concurrent programs. In *CAV*, pages 480–494, 2010.
7. Robert DeLine and Manuel Fähndrich. Typestates for objects. In *ECOOP*, pages 465–490, 2004.
8. Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, Matthew Parkinson, and Viktor Vafeiadis. Concurrent abstract predicates. In *ECOOP*, pages 504–528, 2010.
9. Dino Distefano and Matthew J. Parkinson. jStar: Towards Practical Verification for Java. In *OOPSLA*, pages 213–226, 2008.
10. Manuel Fähndrich and Robert DeLine. Adoption and focus: practical linear types for imperative programming. In *PLDI*, pages 13–24, 2002.
11. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.
12. Jean-Yves Girard. Linear logic. *Theor. Comput. Sci.*, 50(1):1–102, 1987.
13. Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: a minimal core calculus for Java and GJ. pages 132–146, 2001.
14. Bart Jacobs and Frank Piessens. Expressive modular fine-grained concurrency specification. In *POPL*, pages 271–282, 2011.
15. Bart Jacobs, Jan Smans, and Frank Piessens. Verifying the composite pattern using separation logic. In *Proc of SAVCBS 2008*, 2008.
16. J.Smans, B.Jacobs, and F.Piessens. Vericool: An automatic verifier for a concurrent object-oriented language. In *FMOODS*, pages 220–239, 2008.
17. Neel R. Krishnaswami, Lars Birkedal, and Jonathan Aldrich. Verifying event-driven programs using ramified frame properties. In *TLDI '10*, pages 63–76, 2010.
18. Gary T. Leavens, K. Rustan M. Leino, and Peter Müller. Specification and verification challenges for sequential object-oriented programs. *Form. Asp. Comput.*, 19:159–189, June 2007.
19. K. Rustan Leino and Peter Muller. A basis for verifying multi-threaded programs. In *ESOP*, pages 378–393, 2009.
20. Aleksandar Nanevski, Amal Ahmed, Greg Morrisett, and Lars Birkedal. Abstract Predicates and Mutable ADTs in Hoare Type Theory. In *ESOP, volume 4421 of LNCS*, pages 189–204, 2007.
21. Matthew Parkinson and Gavin Bierman. Separation logic and abstraction. In *POPL*, pages 247–258, 2005.
22. D.L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15:1053–1058, 1972.
23. K. Rustan, M. Leino, and Peter Müller. Object invariants in dynamic contexts. In *In ECOOP*, 2004.
24. Alexander J. Summers and Sophia Drossopoulou. Considerate reasoning and the composite design patterns. In *VMCAI*, volume 5944 of Lecture Notes in Computer Science, pages 328–344, 2010.
25. Roger Wolff, Ronald Garcia, Éric Tanter, and Jonathan Aldrich. Gradual typestate. In *ECOOP*, pages 459–483, 2011.