Object Propositions

Ligia Nicoleta Nistor

CMU-CS-17-132

January 3 2018

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Prof. Jonathan Aldrich, Chair
Prof. Frank Pfenning
Prof. David Garlan
Dr. Matthew Parkinson

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

This research was sponsored by the National Science Foundation under grant numbers CNS-0931985, CNS-1035800, and CCF-1116907, Pennsylvania State University under grant number 3724CMUNASAF84H, and the "Fundação para a Ciência e Tecnologia" (Portuguese Foundation for Science and Technology) through the Carnegie Mellon Portugal Program and the Information and Communication Technology Institute at Carnegie Mellon University.

Keywords: verification, Java, Oprop, logic, Boogie

For my family.

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.

In this thesis we present 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 have implemented the methodology into our tool Oprop. We have used Oprop to automatically verify a number of examples, thus showing the applicability of this research.

Acknowledgments

I thank my Ph.D. advisor, my thesis committee members, family and friends.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

There are many open problems within the area of object-oriented formal verification. Despite the work of numerous researchers, significant challenges remain in this area. The presence of aliasing (multiple references pointing to the same object) makes modular verification of code difficult. If there are multiple clients depending on the properties of an object, one client may break the property that others depend on. In order to verify whether clients and implementations are compliant with specifications, knowledge of both aliasing and properties about objects is needed.

Statically verifying the correctness of code and catching bugs early on saves time [23], [22]. Bugs are going to be discovered before the program even starts running. Formal verification saves the programmer time by helping him/her find the errors more quickly. By using an off-the-shelf formal verification tool instead of generating as many test cases as possible, the quality assurance process could be made more efficient. In 1987 Boehm stated that 'finding and fixing a software problem after delivery is often 100 times more expensive than finding and fixing it during the requirements and design phase'. In 2001 he revised that statement by saying that 'the cost-escalation factor for small, noncritical software systems is more like 5:1 than 100:1', which is still significant. Their studies show that peer code reviews, analysis tools and testing catch different problems in the development life cycle of a product - meaning that for best results formal verification should be used together with other mechanisms for quality assurance. Boehm states [22] 'All other things being equal, it costs 50 percent more per source instruction to develop high-dependability software products than to develop low-dependability ones. However, the investment is more than worth it if the project involves significant operations and maintenance costs.' This supports the idea that using formal verification in the development cycle is cost effective in the long run.

Creating verification tools for object-oriented programming languages is a worthy endeavor. According to Oracle [6], the Java platform has attracted more than 6.5 million software developers to date. The Java programming language is used in every major industry segment and has a presence in a wide range of devices, computers, and networks. There are 1.1 billion desktops running Java, 930 million Java Runtime Environment downloads each year, 3 billion mobile phones running Java, 100% of all Blu-ray players run Java and 1.4 billion Java Cards are manufactured each year. Java also powers set-top boxes, printers, Web cams, games, car navigation systems, lottery terminals, medical devices, parking payment stations, and more.

We acknowledge that the world is transitioning from single-CPU machines and single-threaded programs to multi-core machines and concurrent programs. Still, in order to apply a verification procedure in a multi-threaded setting, we first have to know how to apply it for a single thread. Moreover, according to Amdahl's law [14], the speedup of a program using multiple processors in parallel computing is limited by the time needed for the sequential fraction of the program. There is a large number of applications that will not benefit from parallelism, such as any algorithm with a large number of serial steps. These algorithms will need to be implemented sequentially and will need a verification procedure designed especially for them. This thesis describes how we can use information about aliasing and properties of objects to implement a robust verification tool for object-oriented code in a single-threaded setting.

The aim of this thesis is to enrich the world with a practical verification tool for object-oriented programs in single-threaded settings. We present a method for modular verification of object-oriented code in the presence of aliasing.

The seminal work of Parnas [62] 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.

When using the object proposition verification system components such as methods and classes are verified independently of each other. Each method only needs to be aware of the specifications of other methods and not of the code inside those methods. Each class only needs to know about the specifications of the public methods in other classes and the predicates of those classes, it does not need to have access to the fields or other private information of different classes. In this way the verification also follows the principle of encapsulation. Moreover, for a particular method, our specifications do not expose the shared data in the data structures of that method. There are methods where the shared data x is used in multiple places inside the method, first as part of a data structure A and second as part of another data structure B. Specifications in separation logic need to expose that x is shared between A and B, while object propositions do not need to do that. In this sense object propositions have information-hiding properties.

Through the use of *object propositions*, we are able to hide the shared data that the two data structures A and B have in common. The implementations of A and B have a shared fractional permission [25] to access the common data x, but this need not be exposed in their external interface. Our solution is therefore more modular in some cases than the state of the art (such as specifications written in separation logic) with respect to hiding shared data, and we have implemented the object propositions methodology into our Oprop tool.

1.1 Thesis Statement

Object propositions, which statically characterize both the aliasing behavior of program references in object-oriented programs and the abstract predicates that hold about the objects, can be successfully used in single-threaded object-oriented programs to write specifications and to automatically verify that those specifications are obeyed by the code.

1.2 Hypotheses

We can break the thesis statement down into more concrete and measurable hypotheses.

1.2.1 Hypothesis:Formalization

We can develop and formalize a system that will guarantee that a single-threaded program including formal specifications obeys those specifications. The programmer could write very simple specifications and in that case it would be trivial to prove that they are verified, but in this thesis we focus on specifications that provide meaningful, strong properties about the programs being verified. If the specifications are not respected, the proof system can be used to signal which part of the specifications is being broken. Our system should have strong modularity properties: the specifications of methods or classes should reveal as little as possible about the data structures used in the programs, methods should rely only on the pre- and post-conditions of other methods.

Validation This hypothesis is validated by the development and formalization of our type system and dynamic semantics based on object propositions. We have proven that the type system is sound with respect to its semantics. The soundness of the proof rules 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. The specifications in our examples hide the sharing of common objects.

1.2.2 Hypothesis:Practicality

Our verification system can be used to verify object oriented programs with practical designs.

Validation In order to validate this hypothesis, we use the Oprop tool that we implemented to verify a number of simple Java programs and also programs implementing the composite, state, proxy and flyweight design patterns, which have been annotated with object propositions. We chose programs that include updates to shared data structures that have multi-object invariants. The complex Java programs represent more complicated implementations, such as the various design patterns that have also been verified with our verification system.

All programs are written such that their syntax matches the syntax of programs that can be verified using Oprop. We have incorporated as many Java features as possible in the syntax of the object proposition specification language, but of course we have not been to be able to incorporate all Java features.

The small programs that we have verified are written by us, but they implement scientifically significant design patterns. We have verified instances of the state, proxy and flyweight design patterns. Design patterns have been verified in prior work, but there were challenges, a few of which we describe below.

Krishnaswami *et al.* [45] used Idealized ML, a higher-order imperative language and a specification language based on separation logic for it. They verified the iterator, composite, decorator patterns, flyweight, factory and subject-observer patterns and tried to use the Coq [17] interactive theorem prover together with its Ynot extension (the extension was used for separation logic support) to create a machine readable proof. Oprop is a static analysis tool and thus it is very different from an interactive proof assistant like Coq. When using Oprop the programmer needs to write the specifications once and our tool will perform the verification without needing constant input from the programmer. Moreover, when verifying the iterator pattern, Krishnaswami *et al.* were forced to prove 'some preciseness properties because of the use of binary post-conditions'. They mention that 'It is unclear whether the preciseness problem encountered is a limitation of binary post-conditions in general or the current Ynot implementation; we think it is the latter. We did not finish the proof for next in this case, either with or without nextvc (without nextvc the proof became too long for us to finish by hand). New versions of Ynot should provide better tactic support for such examples.' We have not encountered such problems with Oprop, that we built and for which we could extend the implementation if we found that a feature is not supported.

Blewitt *et al.* [21] are using *Hedgehog*, a proof system which attempts to prove that a pattern instance is correctly implemented by a given class or set of classes. Specifying the patterns as structural and static semantic constraints allows the pattern to be compared directly with the source code. The main problem with their approach is that it suffers when a pattern's implementation does not match that of a known specification. In the case of Oprop, the implementation of a pattern can deviate from a known specification and we would be able to verify it. Nonetheless, we acknowledge that when using Oprop the programmer tries to prove that certain properties that he is interested in hold, while the problem for Blewitt *et al.* is different as they try to prove that an implementation correctly implements a design pattern.

Our programs implementing the state, proxy and flyweight design patterns show a few very common patterns of using object oriented languages. This generality makes them valuable, as the model of how they have been verified using object propositions can be studied and the ideas can be applied to entire families of similar programs.

1.3 Contributions of the thesis

The main contributions of this thesis are the following:

- A verification system that unifies substructural logic-based reasoning with invariant-based reasoning. Full permissions permit reasoning similar to separation logic, while fractional permissions introduce non-linear invariant-based reasoning. Unlike prior work [25], 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 static and dynamic semantics that represent the theory of object propositions.
- Another main contribution is our approach to implementing the object propositions system, which resulted in our Oprop tool, available online as a web application at the *lowcost-env.ynzf2j4byc.us-west-2.elasticbeanstalk.com* address. We have used the Oprop tool to automatically verify an instance of the composite pattern, which was a challenging endeavor. The approach we took to implementing our tool - translating our Oprop language into the intermediate verification language Boogie [16] and using the Boogie verifier as a backend - is one of our scientific contributions. Our tool is critical to validating the hypothesis that our methodology is practical.

- Validation of the approach by specifying and proving partial correctness of the composite design pattern, demonstrating benefits in modularity and abstraction compared to other solutions with the same code structure. We have verified our instance of the composite pattern using our tool Oprop.
- We have annotated and verified instances of the state, proxy and flyweight design patterns, thus showing that the object proposition verification system can be used for many kinds of programs. We have manually translated these three desing patterns into the Boogie verification language and we have successfully verified the resulting translations using the Boogie verification tool. We have not automatically verified these examples automatically using our Oprop tool because it does not support some of the needed features - such as *interfaces*. We leave the implementation of such features into Oprop as future work.

A big contribution of our work is related to modularity. Our work is specific to a class: in each class we define predicates that objects of other classes can rely on. We get the modularity advantages while also supporting a high degree of expressiveness by allowing the modification of multiple objects simultaneously. While 'simultaneously' usually means 'concurrently', in our case it means that we can temporarily break (and then restore) the invariants of multiple objects. While prior work on this issue [68] allows invariants that are declared 'broken' in a method specification to not hold before calls to the corresponding methods, the invariants are expected to be fixed by those methods.

Below we highlight the main differences in modularity compared to existing approaches:

- like separation logic and permissions, but unlike conventional object invariant and ownership-based work (including [54] and [55]), our system allows "ownership transfer" by passing unique permissions around (permissions with a fraction of 1). This gives more flexibility to our system because it allows us to change the predicates that hold for objects, reflecting the state changes that happen in the program for those objects.
- unlike separation logic and permission systems, but like object invariant work and its extensions (for example, the work of Summers and Drossopoulou [68]), we can modify objects without owning them. More broadly, unlike either ownership or separation logic systems, in our system object A can depend on a property of object B even when B is not owned by A, and when A is not "visible" from B. This has information-hiding and system-structuring benefits.
- part of the innovation is combining the two mechanisms above so that we can choose between one or the other for each object, and even switch between them for a given object. This allows us to verify more programs than either of the two mechanisms can verify on their own.

1.4 Object Propositions in a Nutshell

Our methodology uses abstract predicates [61] to characterize the state of an object. Parkinson and Bierman [61] describe abstract predicates in the following way: 'Intuitively, the predicates are used like abstract data types. Abstract data types have a name, a scope and a concrete representation. Within this scope the name and the representation can be freely exchanged, but outside only the name can be used. Similarly abstract predicates have a name and a formula. The formula is scoped: inside the scope the name and the body can be exchanged, and outside the predicate must be treated atomically.'

We embed those predicates in a logic and specify sharing using *fractions* [25]. A fraction can be equal to 1 or it can be less than 1. If for a particular object in the system there is only one reference to it, that reference has a fraction of 1 to the object and thus full modifying control over the fields of the object. If there are multiple references to an object, each reference has a fraction less than 1 to the object and each can modify the object as long as that modification does not break a predefined invariant (expressed as a predicate). In case the modification is not an atomic action (and instead is composed of several steps), the invariant might be broken in the course of the modification, but it must be restored at the end of the modification.

We introduce the novel concept of *object propositions*. To express that the object q in Figure 1.1 has full modifying control of a queue of integers greater or equal to 0 and less than or equal to 10, we use the object proposition $q@1\ Range(0, 10)$. This states that there is a unique reference q pointing to a queue of integers in the 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. Following Fähndrich and DeLine [33], our verification system maintains a key [12] for each field of the receiver object, which is used to track the current values of those fields through the method. A key $o.f \to x$ represents read/write access to field f of object o holding a value represented by the concrete value x. This information is useful when reading the example in Section 1.4.1, which is presented to give the reader a flavor of what the verification of a method looks like.

As an illustrative example, we consider two linked queues q and r that share a common tail p, in Figure 1.1. In prior work on separation logic and dynamic frames, the specification of any method has to describe the entire footprint of the method, i.e., all 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 $q@1\ Range(0, 10)$ in the specification of a method accessing q. The fact that p is shared between the two aliases is hidden by the abstract predicate $Range(0, 10)$.

1.4.1 Verification of client code

In this section we present the verification of code that creates objects of type *Link* and predicates *Range*. We present the steps that we go through in order to verify the client code below so that the reader has a sense, early in the thesis, of how object propositions are used in the verification of code.

```
Link la = new (Link(3, null) , Range(0,10));
Link p = new (Link(6, la) , Range(0,10));
Link q = new (Link(1, p) , Range(0,10));
Link r = new (Link(8, p) , Range(0,10));
```

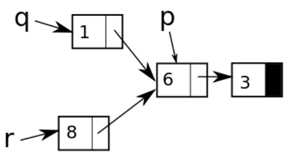

Figure 1.1: Linked queues sharing the tail

```
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]

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

Figure 1.2: Link class and Range predicate

`r.addModulo11(9);`

$\Pi = \cdot$

`Link la = new (Link(3, null) , Range(0,10));`

We pack `la` to the predicate `Range(0,10)` and obtain:

$\Pi = la@1Range(0, 10)$

We go on to the next line of client code:

`Link p = new (Link(6, la) , Range(0,10));`

In order to pack p to the predicate $Range(0, 10)$, we need a k permission to la. This means that we need to split the permission of 1 to la into two half permissions, and we consume one of those halves when packing the predicate $Range(0, 10)$ for p. We obtain:

$\Pi = la@\frac{1}{2})\ Range(0, 10) \otimes p@1\ Range(0, 10)$

We apply the same reasoning for q and r: we split a permission of 1 into two permissions only when needed. After the creation of the four objects la, p, q, r we have the following Π:

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

We go on to the next line of code:

`r.addModulo11(9);`

The specification of $addModulo11()$ mentions that we need some permission k to r. Since we have a permission of 1 to it, after returning from the call to $addModulo11()$ we have:

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

In this way we have verified a small example where the programmer wanted to verify that however the queue is modified, the integers in the queue always are in the range [0,10]. In Section 2.5.2 we discuss this example in more detail.

1.5 Structure of Dissertation

In the following chapter, Chapter 2, we go over the existing approaches of verifying object oriented code and then give a detailed description of the object propositions methodology. In the same chapter we describe a few simple examples written in our simplified version of Java and their Oprop annotations.

Chapter 3 presents the syntax and semantics of the Oprop language, together with the dynamic semantics of the language and a proof of soundness for the proof rules. In Chapter 4 we show the strategy that we used in the implementation of our Oprop tool, how we encoded our Oprop language into first order logic and why this encoding is sound.

In Chapter 5 we describe the examples that we verified using object propositions: instances of the state, proxy and flyweight design patterns. We have written the translation of these files into Boogie by hand and we formally verified them in the Boogie tool. Moreover, we present the verification of an instance of the composite pattern, that we automatically verified using the Oprop tool. We then showcase a number of simple examples that are useful for illustrating the basic features of both our methodology and of our tool.

In the following chapters 6, 8 and 9 we present a detailed comparison between object propositions and other verification approaches such as considerate reasoning and concurrent abstract predicates, we suggest possible avenues for future work and then present our conclusions.

Chapter 2

Existing Approaches and Proposed Approach

2.1 Current Approaches

The verification of object-oriented code can be achieved using the classical invariant-based technique [15]. Object invariants were proposed by Tony Hoare in his paper describing monitors [37] and were crystallized for the object-oriented paradigm by Bertrand Meyer [52]. 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. Summers *et al.* [69] present the advantages of using object invariants: invariant-based reasoning aids and improves software design, the way software is designed can be leveraged by invariant-based reasoning, and object invariants allow local reasoning about global properties.

At the same time, Matthew Parkinson [60] argues that there are two limitations in scaling class invariants to real programs: invariants need to depend on multiple objects, thus limiting the specifications that can be written using the proof language, and invariants need to be temporarily broken and thus the verification that can be performed is limited. These limitations show themselves 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. This is a problem because there might be situations where one would like some methods to have one predicates as post-conditions and other methods to have other predicates as post-conditions. Objects of the same class could start off by satisfying different invariants and continue to satisfy them throughout the program, depending on which method is called on those objects. In the class invariants verification system, this scenario is not possible. 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 [49] 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 class invariant holds. Thus the classic technique for checking object invariants ensures that objects remain well-formed, but it needs the help of pre- and post-conditions in order to describe how objects change over time.

Separation logic approaches [61], [31], [28], etc. bypass the limitations of invariant-based verification techniques by requiring that each method describe all heap locations that are being touched through reading or writing in the body of the method and the predicates that should hold in the pre- and post-conditions of that method. 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 these objects are completely encapsulated. But if they are shared between two structures, that sharing must be revealed in the specifications of methods. This is not desirable from an information hiding point of view.

On the other hand, permission-based work, first proposed by Boyland [25] and then adopted by other works [19], [30] 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*, used by Bierhoff and Aldrich [19] and also adopted by Jan Smans in implicit dynamic frames [67], to express the design intent of their protocols in annotations on methods and classes. Bierhoff and Aldrich [19] explain that 'Access permissions systematically capture different patterns of aliasing. An access permission tracks how a reference is allowed to read and/or modify the referenced object, how the object might be accessed through other references, and what is currently known about the object's typestate. In particular, the full and pure permissions capture the situation where one reference has exclusive write access to an object (a full permission) while other references are only allowed to read from the same object (using pure permissions).'

This thesis uses fractional permissions [25] 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 [30] 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. In some cases our work can be more modular than separation logic approaches because it can more effectively hide the exact aliasing relationships, such as the simulator of queues of jobs from Section 2.5.1. The examples in the sections 2.5.1 and 2.3 represent instances where object propositions are better at hiding shared data than separation logic.

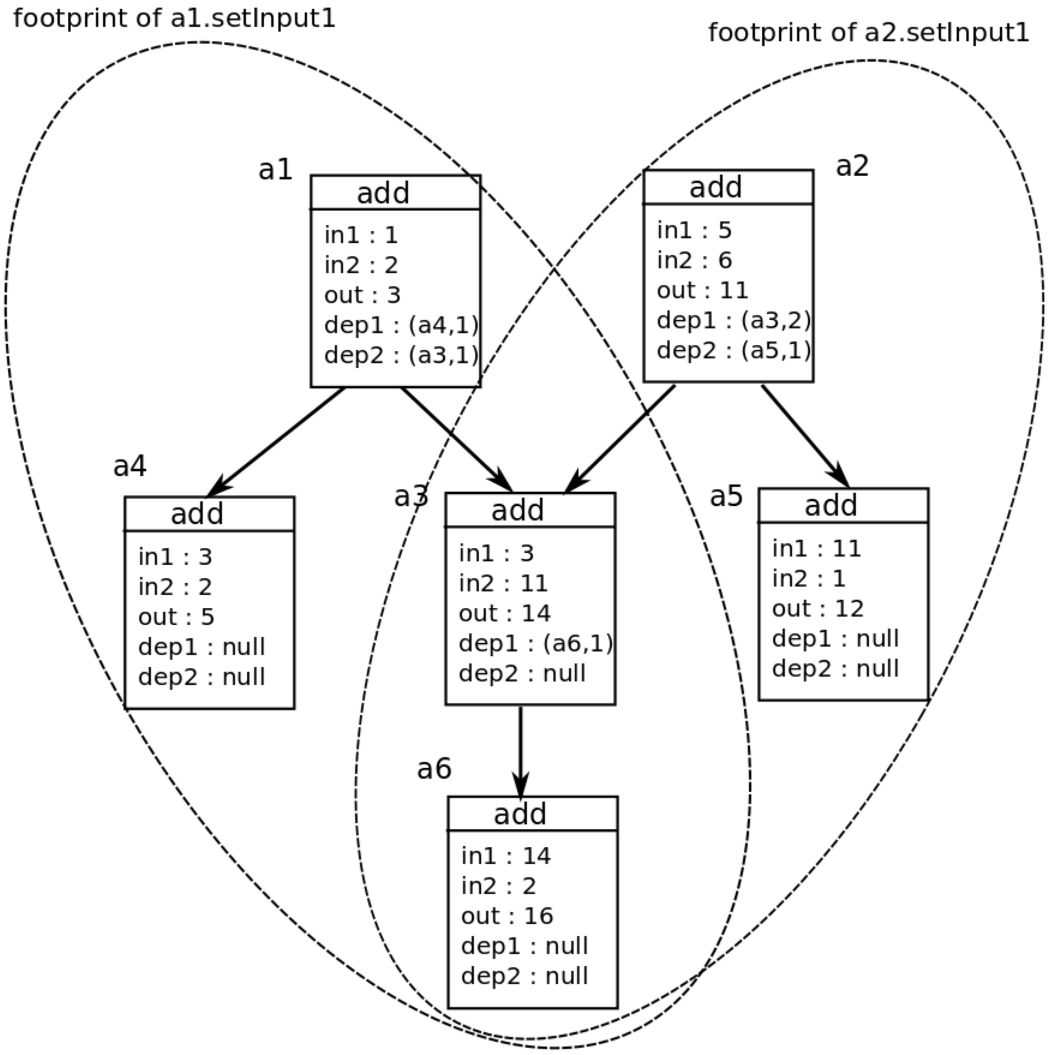

Figure 2.1: Add cells in spreadsheet

2.2 Example: Cells in a spreadsheet

We consider the example of a spreadsheet, as described in [44]. In our spreadsheet each cell contains an add formula that adds two integer inputs and produces an output sum. Each cell may refer to two other cells, by feeding the output sum as an input to each of those cells. The general case would be for each cell to have a dependency list of cells, but since our grammar does not support arrays yet, we are 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 2.1. In separation logic, the specification of any method has to describe all the 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 2.2, we 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, we define the predicate below. In the definition below, $\wedge$ and $\vee$ represent the standard boolean connectives *and* and *or*. Additionally, there are the connectives specific to separation logic: the binary operator $\rightarrow$ takes an address and a value and asserts that the heap is defined at exactly one location, mapping the given address to the given value. The binary operator $\star$ (separating conjunction) asserts that the heap can be split into two disjoint parts where its two arguments hold, respectively.

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

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 2.1) 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 2.1). If the dependencies between the cells form a cycle, as in Figure 2.3, 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 we 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,

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

and

```
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);
    }

 }
```

Figure 2.2: Cell class

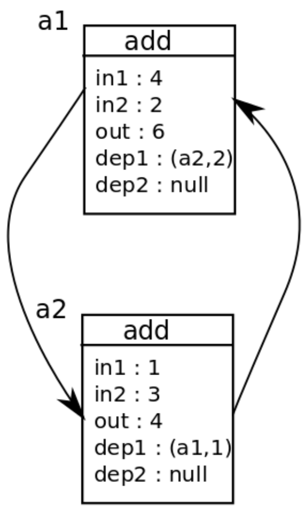

Figure 2.3: Cells in a cycle

$$\begin{aligned}SepOK(a2) \equiv{} & a2.in1 \to z1 \star a2.in2 \to z2 \star a2.out \to p \star z1 + z2 = p \star \\ & (SepOK(a3) \wedge a3.in2 = p) \star (SepOK(a5) \wedge a5.in1 = p)\end{aligned}$$

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. 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 [40]. 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 $SepOK(this) -\!\!* \; SepOK(this)$. The binary operator $-\!\!*$ (separating implication) asserts that extending the heap with a disjoint part that satisfies its first argument results in a heap that satisfies its second argument.

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, the structure of the data on which the method will be called.

Another way of addressing this problem in separation logic is by collecting all the object structure into a mathematical structure and then asserting all the necessary properties using the separation logic star operator. This approach is taken by Jacobs *et al.* [40] for the specification and verification of the composite pattern. Below we show how this approach would apply to our cells in a spreasheet example:

```
struct node;
typedef struct node⋆node;
inductive tree := nil | tree(node, tree, tree);
inductive context :=
   |root
   |leftcontext(context, node, tree)
   |rightcontext(context, node, tree);
predicate tree(node node, context c, tree subtree);

void setInput1(int x);
requires tree(node, ?c, ?t);
ensures tree(node, c, t) ⋆ node → in1 → x

struct dep {
 struct node ⋆ n;
 int i;
};

struct node {
 struct node ⋆left;
 struct node ⋆right;
 int in1, in2, out;
 struct dep ⋆dep1, ⋆dep2;
};

predicate subtree(node node, tree t) ≡
 switch(t) {
 case nil:  node=0;
 case tree(node0, leftNodes, rightNodes):
   node=node0⋆node≠0 ⋆
   node→left→?left⋆
   node→right→?right⋆
   malloc_block_node(node)⋆
   subtree(left, leftNodes)⋆
   subtree(right, rightNodes);
};

predicate context(node p, context c) ≡
switch(c) {
 case root:  p =0;
 case left_context(pns, p0, r):
   p=p0 ⋆ p≠0⋆
   p→left→?left ⋆
   p→right→?right ⋆
   p→in1→?pin1 ⋆
   p→in2→?pin2 ⋆
   p→out→?pout ⋆
   p→dep1→?pdep1 ⋆
   p→dep2→?pdep2 ⋆
   malloc_block_node(p) ⋆
   context(p, gp, pns) ⋆
   subtree(right, r) ⋆
   pout=pin1+pin2 ⋆
   ((pdep1→n→in1 = pin1 ⋆ pdep1→i=1 ) ∨
   (pdep1→n→in2 = pin2 ⋆ pdep1→i=2 )) ⋆
   ((pdep2→n→in1 = pin1 ⋆ pdep2→i=1 ) ∨
   (pdep2→n→in2 = pin2 ⋆ pdep2→i=2 )) ⋆
 case right_context(pns, p0, l):
   ...analogous...
};

predicate tree(node node, context c, tree subtree)≡
  context(node, c) ⋆
  subtree(node, subtree);
```

The above version of separation logic specification is cumbersome because of the extra information that has to be kept around, such as the focus node's subtree and the focus node's context.

Separation logic approaches will thus have a difficult time trying to verify this kind of code. This is because in object-oriented design, the natural abstraction is that each cell updates its dependents, while they are hidden from the outside. The cells in the spreadsheet example is an instance of the subject-observer pattern, as described in [41], which implements this abstraction.

2.2.1 Verifying Cells in a Spreadsheet Example using Ramified Frame Properties

Krishnaswami *et al.* [44] describe the specification and verification of demand-driven notification networks. The example of a spreadsheet of cells can be considered as such a notification network. The implementation of the cells in the spreadsheet using ramified frames is given in Figure 2.4.

Krishnaswami *et al.* use the idea of stream transducers to model the cells in the spreadsheet. The spreadsheet is seen as an interactive system that is modeled as a function that takes a stream of inputs $\mathsf{stream}(A)$ and yields a stream of outputs $\mathsf{stream}(B)$. The semantics of stream tranducers is given in Figure 2.5.

The implementation of the spreadsheet using stream transducers is given in Figure 2.6.

The specifications in this section demonstrate how complicated it is to verify programs that exhibit sharing of state, such as the cells in a spreadsheet, using ramified frame properties. Although this technique is theoretically beautiful, no verification tools that are based on it have been implemented.

2.3 Modularity Example: Simulator for Queues of Jobs

The formal verification of modules should ideally follow the following principle: the specification and verification of one method should not depend on details that are private to the implementation of another method that is called by the specified method. 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.

As shown earlier in this chapter, some of the most popular reasoning techniques available today — principally those based on separation logic [65] — cannot hide sharing, because the specification of a method must mention the entire memory footprint that the method accesses. We back this claim by giving in this section the full specification in separation logic for a simulator of queues of jobs. Abstract predicates are not sufficient to hide the details because, as we have seen in Section 2.2 for the separation logic specification of the cells in a spreadsheet example, there are cases when the different abstract predicates in the pre-condition have to reveal the exact structure of the cells.

This gives rise to non-modular, and therefore fragile, specifications and proofs. There exist versions of higher-order separation logics that can hide the presence of sharing to some extent [44], but their higher-order nature makes them considerably more complicated both for specification and verification.

To illustrate the modularity issues, we present here a relatively realistic example. Figure 2.7 depicts 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

```
code : $\star \to \star$
code $\alpha = \bigcirc(\alpha \times$ cellset)

cell : $\star \to \star$
cell $\alpha = \{code$ : ref code $\alpha$;
           $value$ : ref option $\alpha$;
           $reads$ : ref cellset;
           $obs$ : ref cellset;
           $unique : \mathbb{N}\}$

$ecell = \exists\alpha : \star.$ cell $\alpha$

return  : $\forall\alpha : \star.\ \alpha \to$ code $\alpha$
return  $\alpha\ x = [\langle x,$ emptyset$\rangle]$

bind : $\forall\alpha, \beta : \star.$ code $\alpha \to (\alpha \to$ code $\beta) \to$ code $\beta$
bind $\alpha\ \beta\ e\ f = $ [letv $(v, r_1) = e$ in
                 letv $(v', r_2) = f\ v$ in
                 $\langle v',$ union $r_1\ r_2\rangle]$

read : $\forall\alpha : \star.$ cell $\alpha \to$ code $\alpha$
read $\alpha\ a = $ [letv $o = [!a.value]$ in
             run case($o$,
                 Some $v \to [\langle v,$ singleton $a\rangle]$,
                 None$\to$
                     [letv $exp = [!a.code]$ in
                     letv $(v, r) = exp$ in
                     letv _ $= [a.value :=$ Some$(v)]$ in
                     letv _ $= [a.reads := r]$ in
                     letv _ $=$ iterset (add_observer pack$(\alpha, a))\ r$ in
                      $\langle v,$ singleton $a\rangle])$

getref  : $\forall\alpha :$ ref $\alpha \to$ code $\alpha$
getref $\alpha\ r = $ [letv $v = [!r]$ in $\langle v,$ emptyset$\rangle]$

setref  : $\forall\alpha :$ ref $\alpha \to \alpha \to$ code $\mathbf{1}$
setref $\alpha\ r\ v = $ [letv _ $= [r := v]$ in $\langle\langle\rangle,$ emptyset$\rangle]$

newcell  : $\forall\alpha : \star.$ code $\alpha \to \bigcirc$cell $\alpha$
newcell  $\alpha\ code = $ [letv $unique = !counter$ in
                   letv _ $= [counter := unique + 1]$ in
                   letv $code = $ new$_{\text{code } \alpha}(code)$ in
                   letv $value = $ new$_{\text{option } \alpha}$(None) in
                   letv $reads = $ new$_{\text{cellset}}$(emptyset) in
                   letv $obs = $ new$_{\text{cellset}}$(emptyset) in
                    $(code, value, reads, obs, unique)]$

update : $\forall\alpha : \star.$ code $\alpha \to$ cell $\alpha \to \bigcirc\mathbf{1}$
update $\alpha\ exp\ a = $ [letv _ $=$ mark_unready pack$(\alpha, a)$ in
                   $a.code := exp]$

mark_unready : $ecell \to \bigcirc\mathbf{1}$
mark_unready $cell = $ unpack$(\alpha, a) = cell$ in
                     [letv $os = [!a.obs]$ in
                      letv $rs = [!a.reads]$ in
                      letv _ $=$ iterset mark_unready $os$ in
                      letv _ $=$ iterset (remove_obs $cell)\ rs$ in
                      letv _ $= [a.value :=$ None] in
                      letv _ $= [a.reads :=$ emptyset] in
                      $a.obs :=$ emptyset]

add_observer : $ecell \to ecell \to \bigcirc 1$
add_observer $a$ pack$(\beta, b) = $ [letv $os = [!b.obs]$ in
                             $b.obs :=$ addset $os\ a]$

remove_obs : $ecell \to ecell \to \bigcirc 1$
remove_obs $a$ pack$(\beta, b) = $ [letv $os = [!b.obs]$ in
                           $b.obs :=$ removeset $os\ a]$
```

Figure 2.4: Implementation of Notification Networks

$$\mathsf{ST}(A,B) = \{f \in \mathsf{stream}(A) \to \mathsf{stream}(B) \mid causal(f)\}$$

$$\begin{aligned}
&\mathit{lift} : (A \to B) \to \mathsf{ST}(A,B)\\
&\mathit{lift}\ f\ as = map\ f\ as
\end{aligned}$$

$$\begin{aligned}
&seq : \mathsf{ST}(A,B) \to \mathsf{ST}(B,C) \to \mathsf{ST}(A,C)\\
&seq\ p\ q = q \circ p
\end{aligned}$$

$$\begin{aligned}
&par : \mathsf{ST}(A,B) \to \mathsf{ST}(C,D) \to \mathsf{ST}(A \times C, B \times D)\\
&par\ p\ q\ abs = zip\ (p\ (map\ \pi_1\ abs))\ (q\ (map\ \pi_2\ abs))
\end{aligned}$$

$$\begin{aligned}
&switch : \mathbb{N} \to \mathsf{ST}(A,B) \to \mathsf{ST}(A,B) \to \mathsf{ST}(A,B)\\
&switch\ k\ p\ q = \lambda as.\ (take\ k\ (p\ as)) \cdot (q\ (drop\ k\ as))
\end{aligned}$$

$$\begin{aligned}
&loop : A \to \mathsf{ST}(A \times B, A \times C) \to \mathsf{ST}(B,C)\\
&loop\ a_0\ p = (map\ \pi_2) \circ (cycle\ a_0\ p)
\end{aligned}$$

$$\begin{aligned}
&cycle : A \to \mathsf{ST}(A \times B, A \times C) \to \mathsf{ST}(B, A \times C)\\
&cycle\ a_0\ p = \lambda bs.\ \lambda n.\ last(gen\ a_0\ p\ bs\ n)
\end{aligned}$$

$$\begin{aligned}
&gen : A \to \mathsf{ST}(A \times B, A \times C) \to \mathsf{stream}(B)\\
&\qquad\quad \to \mathbb{N} \to \mathsf{list}\ (A \times C)\\
&gen\ a_0\ p\ bs\ 0 \quad = \hat{p}\ [(a_0, bs_0)]\\
&gen\ a_0\ p\ bs\ (n+1) =\\
&\ \ \hat{p}\ (zip(a_0 :: (map\ \pi_1\ (gen\ a_0\ p\ bs\ n)))\ (take\ (n+2)\ bs))
\end{aligned}$$

Figure 2.5: Semantics of Stream Transducers

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.

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 2.7. 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 this thesis that our methodology does not modify the specification of the control object, thus allowing one to make changes locally without influencing other code, while (first-order) separation logic approaches [32] will modify the specification of the controller.

The code in Figures 2.8, 2.9 and 2.10 represents the example from Figure 2.7.

Now, let's imagine changing the code to reflect the modifications in the right image of Figure 2.7. The current separation logic approaches break the information hiding principle. Distefano and Parkinson [32] 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 cases, they reveal which references point to the same shared data, and this violates the information hiding principle by unnecessarily exposing the structure of the data.

```
ST : $\star \to \star \to \star$
ST$(\alpha, \beta) \equiv$ cell $\alpha \to \bigcirc$cell $\beta$

lift  : $\forall \alpha, \beta : \star.\ (\alpha \to \beta) \to$ ST$(\alpha, \beta)$
lift  $\alpha\ \beta\ f\ input =$
  newcell  (bind (read $input$) ($\lambda x : \alpha.$ return $(f\ x)$))

seq  : $\forall \alpha, \beta : \star.$ ST$(\alpha, \beta) \to$ ST$(\beta, \gamma) \to$ ST$(\alpha, \gamma)$
seq $\alpha\ \beta\ p\ q\ input =$ [letv $middle = p\ input$ in
                         letv $output = q\ middle$ in
                          $output$]

par  : $\forall \alpha, \beta, \gamma, \delta : \star.$
          ST$(\alpha, \beta) \to$ ST$(\gamma, \delta) \to$ ST$(\alpha \times \gamma, \beta \times \delta)$
par $\alpha\ \beta\ \gamma\ \delta\ p\ q\ input =$
 [letv $a =$ newcell (bind (read  $input$)
                          ($\lambda x : \alpha \times \beta.$ return (fst $x$))) in
  letv $b = p\ a$ in
  letv $c =$ newcell (bind (read  $input$)
                          ($\lambda x : \alpha \times \beta.$ return (snd $x$))) in
  letv $d = q\ c$ in
  letv $output =$ newcell (bind (read $b$) ($\lambda b : \beta.$
                          bind (read $d$) ($\lambda d : \delta.$
                            return  $\langle b, d \rangle$))))] in
   $output$]

switch  : $\forall \alpha, \beta : \star.\ \mathbb{N} \to$ ST$(\alpha, \beta) \to$ ST$(\alpha, \beta) \to$ ST$(\alpha, \beta)$
switch $\alpha\ \beta\ k\ p\ q\ input =$
 [letv $r =$ new$_{\mathbb{N}}(0)$ in
  letv $a = p\ input$ in
  letv $b = q\ input$ in
  letv $out =$ newcell  (bind (getref $r$) ($\lambda i : \mathbb{N}.$
                       bind (setref $r$ $(i + 1)$) ($\lambda q : \mathbf{1}.$
                         if($i < k$, read $a$, read $b$)))) in
   $out$]

loop  : $\forall \alpha, \beta, \gamma : \star.\ \alpha \to$ ST$(\alpha \times \beta, \alpha \times \gamma) \to$ ST$(\beta, \gamma)$
loop $\alpha\ \beta\ \gamma\ a_0\ p\ input =$
 [letv $r =$ new$_{\alpha}(a_0)$ in
  letv $ab =$ newcell  (bind (read $input$) ($\lambda b : \beta.$
                      bind (getref $r$) ($\lambda a : \alpha.$
                        return  $\langle a, b \rangle$)))) in
  letv $ac = p\ ab$ in
  letv $c =$ newcell  (bind (read $ac$) ($\lambda v : \alpha \times \gamma.$
                     bind (setref $r$ (fst $v$)) ($\lambda q : \mathbf{1}.$
                       return (snd $v$)))) in
   $c$]
```

Figure 2.6: Imperative Stream Transducers

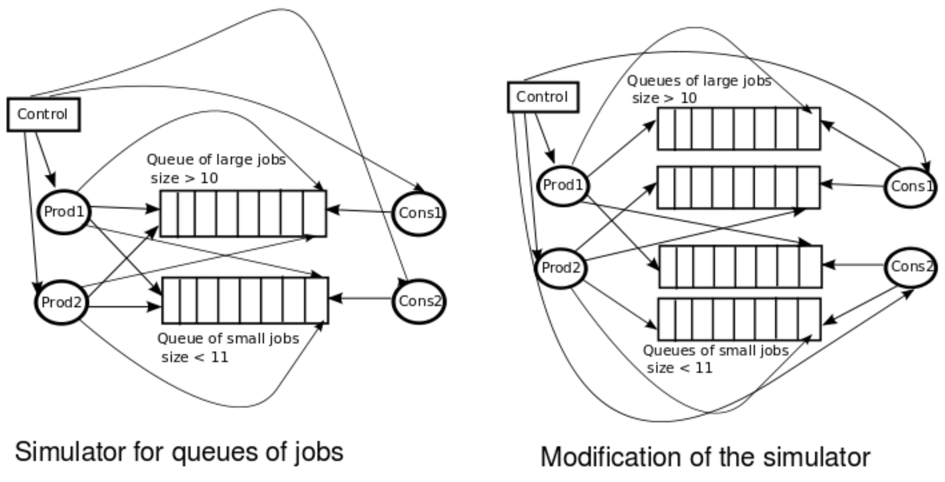

Figure 2.7:

Below we present two ways of giving the specifications needed to verify the code in Figure 2.7 using separation logic. In the first specification we put the keys representing the actual values of the fields in the parameters of the predicates. In the second specification the parameters of the predicates are existentially quantified.

2.3.1 First Specification

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 pre-condition for the `produce()` method is:

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

Note that one can think of the definition $Prod(p)$ below

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

to be more modular than $Prod(this, ss, es, sl, el)$ because the predicate exposes fewer parameters, but then the pre-condition for the `produce()` method would become:

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

This would remove the connection between the predicates $Prod$ and $Listseg$ in the pre-condition and would make the pre-condition much less meaningful because the information that ss corresponds to the $startSmallJobs$ field of p and that sl corresponds to the $startLargeJobs$ field of p would be lost.

The predicate for the Consumer class is

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

The pre-condition for the `consume()` method is:

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

The predicate for the Control class is :

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

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

  public void produce()
   { 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;}
   }
  }
}
```

Figure 2.8: Producer class

```
public class Consumer {

      Link startJobs;

  public Consumer(Link s) {
        startJobs = s;

  public void consume()
    { if (startJobs != null)
       {System.out.println(startJobs.val);
       startJobs = startJobs.next;}
  }
```

Figure 2.9: Consumer class

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

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

  public void makeActive( int i)
    { 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);}
        }
    }
```

Figure 2.10: Control class

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

The pre-condition 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 2.7.

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 pre-condition 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 pre-condition 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 is an indicator of loss of modularity.

2.3.2 Second Specification

The predicate for the Producer class is $Prod(this)$, where :
$Prod(p) \equiv \exists ss : Link, es : Link, sl : Link, el : Link.$
$p.startSmallJobs \rightarrow ss \star p.endSmallJobs \rightarrow es \star$
$p.startLargeJobs \rightarrow sl \star p.endLargeJobs \rightarrow el.$

The pre-condition for the `produce()` method would then be:
$Prod(p) \star \exists ss : Link, sl : Link.(Listseg(ss, null, 0, 10) \star Listseg(sl, null, 11, 100)).$

The definition of the predicate $Prod(p)$ exposes fewer parameters and could seem more modular, but the precondition given above for the `produce()` method becomes more difficult to prove. Since this precondition uses existentially quantified variables, it will be very difficult for an automated theorem prover such as Z3 to prove it.

Moreover, the connection between the predicates $Prod$ and $Listseg$ from the precondition is removed, thus making the precondition much less meaningful.

The predicate for the Consumer class is
$Cons(c) \equiv \exists c : Link.c \rightarrow s.$

The pre-condition for the `consume()` method would be:
$Cons(c) \star \exists s : Link.Listseg(s, null, 0, 10).$

The predicate for the Control class is :
$Ctrl(ct) \equiv \exists p1 : Producer, p2 : Producer, c1 : Consumer, c2 : Consumer.$
$ct.prod1 \rightarrow p1 \star ct.prod2 \rightarrow p2 \star ct.cons1 \rightarrow c1 \star ct.cons2 \rightarrow c2.$

The pre-condition for `makeActive()` would be:
$Ctrl(this) \star Prod(p1) \star Prod(p2) \star Cons(c1) \star Cons(c2)$

$\star \exists ss : Link, sl : Link.(Listseg(ss, null, 0, 10) \ \star \ Listseg(sl, null, 11, 100))$.
The precondition for the `consume()` method becomes:
$Cons(c) \ \star \ \exists s1 : Link, s2 : Link.(Listseg(s1, null, 0, 10) \ \star \ Listseg(s2, null, 0, 10))$.
The precondition for `consume()` reveals that there are two distinct queues each with its elements in the range [0,10], but one does not know how they relate to $Cons(c)$.
In this version of the specification we do not experience a lack of modularity, but instead the pre- and postconditions of methods become meaningless, as the connection between the queues and the $Listseg$ predicate has been removed.
The pre-condition for `makeActive()` in class Control becomes:
$Ctrl(this) \ \star \ Prod(p1) \ \star \ Prod(p2) \ \star$
$\exists ss1 : Link, ss2 : Link, sl1 : Link, sl2 : Link.$
$Cons(c1) \ \star \ Cons(c2) \ \star \ Listseg(ss1, null, 0, 10) \ \star$
$Listseg(ss2, null, 0, 10) \ \star \ Listseg(sl1, null, 11, 100) \ \star \ Listseg(sl2, null, 11, 100)$
The existentially quantified specifications do not change very much although the pointers to the job queues have been modified, but they do not convey much information either about the requirements on the queues. We cannot include the $Listseg$ predicate in the definition of the $Prod$, $Cons$ or $Ctrl$ predicates because the predicates $Prod$ and $Cons$ have to co-exist and they refer to the same queue (same memory), say ss. This means that the predicate $List(ss, null, 0, 10)$ would appear twice when we have $Prod(p1) \star Cons(c1)$, i.e., we would have $List(ss, null, 0, 10) \star List(ss, null, 0, 10)$. This is not possible in separation logic, unless it is augmented with fractions.

2.4 Detailed Description of Proposed Approach

Our methodology uses abstract predicates [61] to characterize the state of an object, embeds those predicates in a logical framework, and specifies sharing using fractional permissions [25].

Our main technical contribution is the novel abstraction called *object proposition* that combines predicates with aliasing information about objects. 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.

Our checking approach is modular and verifies 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 [33], 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* [30] the keys back into an object proposition and check that object proposition against the method post-condition. A crucial part of our approach is that 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. The intuition behind this is that when we hold a full fraction, equal to 1, to an object, we have total control over it and we can change the prdicate that holds for it.

2.5 Examples with Object Propositions

2.5.1 Cells in a spreadsheet

In Figure 2.12, we present the Java class from Figure 2.2 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 are consumed upon usage, the object propositions are consumed upon usage and their duplication is forbidden. Therefore, we use a fragment of linear logic [35] to write the specifications. 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). We do not use all the linear logic connectives, such as the alternative conjunction &, exponential modality **!** or the multiplicative unit **1**. This means that a linear theory is at the center of our system and not a linear logic. From now on in this thesis we will use the terminology *linear theory*.

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 3.1). 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. The critical property of an invariant is that it cannot have parameters that represent the fields of the current object. This is because invariants are supposed to hold at the boundaries of methods and if the parameters are changed inside a method, the invariant is broken and other aliases to the same object cannot rely on the invariant anymore. Thus, the invariant for the cell class is $OK()$, while the predicates $In1(int\ x1), In2(int\ x2), OKdep(int\ o)$ cannot be invariants because they have parameters that refer to the fields of the current object and can change.

A critical part of our work is allowing clients to depend on a property of a shared object. Other methodologies such as Boogie [16] 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. Summers and Drossopoulou's work [68] accomplishes the same thing, as they do not have the concept of owner. At the same time, they use the classical invariant technique and we have previously discussed in Section 2.1 how our work is different from that line of research.

```
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);
    }

 }
```

Figure 2.11: Cell class

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

   $predicate\ \ OKdep(int\ \ o)\ \ \equiv \exists c, k, i.this.ce \rightarrow c \otimes this.input \rightarrow i \otimes$
       $((i = 1 \otimes c@\frac{1}{2}\ \ In1(o) \otimes c@k\ \ OK()) \oplus (i = 2 \otimes c@\frac{1}{2}\ \ In2(o) \otimes c@k\ \ OK()))$
  }
class Cell {
      int in1, in2, out;
      Dependency dep1, dep2;

   $predicate\ \ In1(int\ \ x1) \equiv this.in1 \rightarrow x1$

   $predicate\ \ In2(int\ \ x2) \equiv this.in2 \rightarrow x2$

   $predicate\ \ OK()\ \ \equiv \exists x1, x2, o, d1, d2.this@\frac{1}{2}\ \ In1(x1) \otimes\ \ this@\frac{1}{2}\ \ In2(x2) \otimes$
        $x1 + x2 = o\ \ \otimes this.out \rightarrow o \otimes this.dep1 \rightarrow d1 \otimes this.dep2 \rightarrow d2 \otimes$
       $d1@1\ \ OKdep(o) \otimes d2@1\ \ OKdep(o)$

   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)
   $\forall k.(this@k\ \ OK() \multimap this@k\ \ OK())$
   { this.in1 = x;
     this.out = this.in1 + this.in2;
     this.setInputDep(out);
   }

   void setInput2(int x)
   $\forall k.(this@k\ \ OK() \multimap this@k\ \ OK())$
   { this.in2 = x;
     this.out = this.in1 + this.in2;
     this.setInputDep(out);
   }

  }
```

Figure 2.12: Cell class and OK predicate

To gain read or write access to the fields of an object, we have to $unpack$ it [30]. 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 3.3 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 [30] 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.

Like other previous systems, our system uses abstraction, which allows clients to treat method pre/post-conditions opaquely: outside the scope where the predicates are defined, clients can only refer to the names of predicates.

The predicate *OK()* in Figure 2.12 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 use only a fractional permissions for the dependency cells (such as in $c@k\ OK()$, 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.

2.5.2 Simulator for Queues of Jobs

The code in Figures 2.13, 2.14 and 2.15 represents the code from Figures 2.8, 2.9 and 2.10, augmented with predicates and object propositions. 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]. Predicate $Range$ is defined in Figure 1.2.

When changing the code to reflect the modifications in the right image of Figure 2.7, the internal representation of the predicates changes, but the 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
`ConsumeInRange(x,y)` $\equiv \exists o_1, o_2, k_1, k_2.$ `startJobs1`$\rightarrow o_1 \otimes$ `startJobs2`$\rightarrow o_2$ $\otimes o_1 @ k_1$ `Range(x,y)` $\otimes o_2 @ k_2$ `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 modularity to the code.

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

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

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

  public void produce()
   ∃ k.this@k BothInRange() ⊸
     ∃ k.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;}
   }
  }
}
```

Figure 2.13: Producer class

```
public class Consumer {
      Link startJobs;

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

  public Consumer(Link s) {
        startJobs = s;

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

Figure 2.14: Consumer class

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

  predicate WorkingSystem() ≡
    prod1→ o₁⊗ prod2→ o₂
         ⊗ cons1→ o₃⊗ cons2 → o₄
         ⊗ ∃k₁.o₁@k₁ BothInRange()
         ⊗ ∃k₂.o₂@k₂ in BothInRange()
         ⊗ ∃k₃.o₃@k₃ in ConsumeInRange(0,10)
         ⊗ ∃k₄.o₄@k₄ in 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)
   ∃k.this@k WorkingSystem() ⊸
       ∃k.this@k in 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);}
        }
    }
```

Figure 2.15: Control class

2.6 Example: Simple Composite

The composite design pattern [1] is used when clients need to treat individual objects and compositions of objects uniformly. This pattern is applied when there is a hierarchy of objects and compositions that is represented as a tree structure. This pattern is used when the differences between individual objects and compositions of objects are not relevant and the programmer sees that the code to handle multiple objects is almost the same as the code to handle one object. The kind of properties that we care about verifying for the composite pattern refer to objects and composites in a uniform way and are hence general properties, that refer to the whole composite tree. If we expect the tree to be a binary tree, we could verify that it is indeed the case that at all times in the program each node has at most two children or it is a leaf node. If we expect the tree to be a red-black tree, we could verify the properties that such a tree should always satisfy: that each node is either red or black, that the root is black, that all leaves are black, that if a node is red then both its children are black and that every path from a given node to any of its descendant leaf nodes contains the same number of black nodes.

The code for a very simple specification of the composite pattern is given below. Note that this is not the Composite pattern as proposed by Leavens *et al.* [46], where any change to the field *count* causes all the ancestors to be updated. Instead below we give a simpler version of the Composite where only the *count* field of the current object is being updated. The rationale for introducing this example is to make the reader familiar with our specification and predicates used for the Composite pattern, for which we give the full implementation and specification in Section 5.4.

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

  void setLeft(Composite l) {
    l.parent=this;
    this.left=l;
    this.count=this.count+l.count+1;
  }

  void setRight(Composite r) {
    r.parent=this;
    this.right=r;
    this.count=this.count+r.count+1;
  }

 }
```

Figure 2.16: Composite class

The predicates for this class are given in Figure 2.17.

$$\text{predicate } \mathit{count}\ (\text{int c}) \equiv \exists \text{ol, or, lc, rc. this.count} \rightarrow \text{c} \otimes$$
$$\text{c} = \text{lc} + \text{rc} + 1\ \otimes \text{this}@1\ left(\text{ol, lc})$$
$$\otimes\ this@1\ right(\text{or, rc})$$

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

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

Figure 2.17: Predicates for Simple Composite

With the help of those predicates, the specification of the method $setLeft$ is written as follows:

$\exists c_1, c_2.this@1\ count(c_1) \otimes l@1\ count(c_2) \multimap this@1\ count(c_1 + c_2 + 1)$.

Below there is an example that can be verified using separation logic, but can also be verified using object propositions.

```
...
  {}
Composite a = new Composite();
  {a@1 count(0)}
Composite b = new Composite();
  {b@1 count(0) ⊗ a@1 count(0)}
Composite c = new Composite();
  {c@1 count(0) ⊗ b@1 count(0) ⊗ a@1 count(0)}
a.setLeft(b);
  {a@1 count(1) ⊗ c@1 count(0)}
a.setRight(c);
  {a@1 count(2)}
...
```

As can be seen in the example above, all the fractional permissions are equal to 1, meaning that there is no sharing of data in this example. Because the data structure does not have sharing of data and there is no danger of exposing any shared data, it is very suitable to verification using separation logic. But it can also be verified using object propositions. The notion of a fractional permission of 1 is incorporated in object propositions, which basically means that data is not shared and can be changed only from one reference point. Because all the fractional permissions in this example are equal to 1, in the verification using object propositions we do not need invariants. In fact, since all the predicates $count$, $left$ and $right$ have parameters that refer to the fields of the current object, they cannot be invariants.

Chapter 3

Formal System

The programming language that we are using is inspired by Featherweight Java [38], extended to include object propositions. We retained only Java concepts relevant to the core technical contribution of this thesis, omitting features such as inheritance, casting or dynamic dispatch that are important but are handled by orthogonal techniques.

In this chapter we first present the grammar of our new Oprop language. We then discuss our permission splitting rules and the static proof rules that we use in the verification of object oriented code. We continue by describing the dynamic semantics rules and how they are used to prove the soundness of the object proposition rules. Finally in sections 3.5 and 3.5.4 we state and prove the preservation theorem that is the basis for the soundness of our system.

3.1 Grammar

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, 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 an integer. The variable z represents a metavariable for fractions and it has type $double$. In a program, a fraction can be a constant of type double or it can be represented by a metavariable.

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

3.2 Permission Splitting

In order to allow objects to be aliased, we must split a fraction of 1 into multiple fractions less than 1 [25]. The fraction splitting rule is defined in Figure 3.1. 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 3.2.

3.3 Proof Rules

This section describes the proof rules that can be used to verify correctness properties of code. The judgment 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 has type T with postcondition formula R".This judgment 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.

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

The static proof rules also contain the following judgments: $\Gamma \vdash r : C$, $\Gamma; \Pi \vdash R$ and

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

Figure 3.1: Rule for splitting fractions

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

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

Figure 3.2: Rules for adding fractions

$\Gamma; \Pi \vdash r.T; R$. The judgment $\Gamma \vdash r : C$ means that in valid type context Γ, the reference r has type C. The judgment $\Gamma; \Pi \vdash R$ means that from valid type context Γ and linear context Π we can deduce that object proposition R holds. The judgment $\Gamma; \Pi \vdash r.T; R$ 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, we provide the intuition for why our system is sound (the formal soundness theorem is given below in Section 3.5). The first invariant enforced by our 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.

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 our 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. (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 our examples in this thesis

$$\frac{\Gamma, \Pi \vdash R \quad \Gamma \vdash R == R'}{\Gamma, \Pi \vdash R'} \text{(EQUAL)}$$

Figure 3.3: Rule for equality of linear formulas

each object has only one invariant. In future work we 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. We need the rule in Figure 3.3 to express the fact that when two propositions are equivalent in linear logic they are also equivalent in our system, where == is pure (linear) logical equivalence.

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 judgment for existential introduction. The notation $[e'/x]e$ substitutes e' for occurrences of x in e. The FIELD rule checks field accesses analogously.

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

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

NEW checks object construction. We get the new object, a key for each field and the remaining linear context.

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

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.

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

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

$$\frac{\begin{array}{c}\Gamma; \Pi \vdash e_1 : \exists x.T_1; \Pi_2 \\ (\Gamma, x : T_1); \Pi_2 \vdash e_2 : \exists w.T_2; R_2\end{array}}{\Gamma; \Pi \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \exists w.T_2; R_2}\ \textsc{Let}$$

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

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 judgment MTYPE) and then goes on to state the object proposition holding for the result. Our system is using a linear theory, which does not have the contraction or the weakening rules. This means that resources, such as object propositions, cannot be duplicated, and also that unpacked object propositions cannot be dropped when we infer one formula from another (for example $\Pi_1 \vdash \Pi_2$). In this way we make sure that object propositions and the predicates associated with them are not lost in our proof derivations.

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. The fact that we do not allow Π_1 to have unpacked predicates does represent a limitation in our system, however, it is one that goes hand in hand with the advantage of supporting shared predicates.

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

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

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.

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

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 2.12, the method $setInput1$ 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). The judgment $\models$ in the last premise of the UNPACK2 rule is different than the normal $\vdash$ judgment because when stating $\Pi \models r \neq r'$ we do not simply use the object propositions from Π, but instead we state that no two unpacked object propositions in Π refer to the same object.

$$\frac{\begin{array}{c}\Gamma;\Pi \vdash r : C; [\overline{t_2}/\overline{x}]R_2 \otimes \textsf{unpacked}(r@1\ Q_1(\overline{t_1})) \otimes \Pi_1 \\ \texttt{predicate}\ Q_2(\overline{Tx}) \equiv R_2 \in C \quad \texttt{predicate}\ Q_1(\overline{T_1x}) \equiv R_1 \in C \\ \Gamma;(\Pi_1 \otimes r@1\ Q_2(\overline{t_2})) \vdash e : \exists x.T; R\end{array}}{\Gamma;\Pi \vdash \texttt{pack}\ r@1\ Q_2(\overline{t_2})\ \texttt{in}\ e : \exists x.T; R}\ \textsc{Pack1}$$

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

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.

$$\frac{\begin{array}{c}\Gamma;\Pi \vdash r : C; r@1\ Q(\overline{t_1}) \otimes \Pi_1 \\ \texttt{predicate}\ Q(\overline{Tx}) \equiv R_1 \in C \\ \Gamma; (\Pi_1 \otimes [\overline{t_1}/\overline{x}]R_1 \otimes \textsf{unpacked}(r@1\ Q(\overline{t_1}))) \vdash e : \exists x.T; R\end{array}}{\Gamma;\Pi \vdash \texttt{unpack}\ r@1\ Q(\overline{t_1})\ \texttt{in}\ e : \exists x.T; R}\ \text{UNPACK1}$$

$$\frac{\begin{array}{c}\Gamma;\Pi \vdash r : C; r@k\ Q(\overline{t_1}) \otimes \Pi_1 \\ \texttt{predicate}\ Q(\overline{Tx}) \equiv R_1 \in C \qquad 0 < k < 1 \\ \Gamma; (\Pi_1 \otimes [\overline{t_1}/\overline{x}]R_1 \otimes \textsf{unpacked}(r@k\ Q(\overline{t_1}))) \vdash e : \exists x.T; R \\ \forall r', \bar{t} : (\ \mathit{unpacked}(r'@k'\ Q(\bar{t})) \in \Pi \Rightarrow \Pi \models r \neq r')\end{array}}{\Gamma;\Pi \vdash \texttt{unpack}\ r@k\ Q(\overline{t_1})\ \texttt{in}\ e : \exists x.T; R}\ \text{UNPACK2}$$

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

3.4 Dynamic Semantics and Soundness

The dynamic semantics for our language is given in Figure 3.4. Below we describe the definitions used for dynamic semantics support.

$C(\bar{o}) \in$ OBJECTS

v ::= o
(values)

$\mu \in$ OBJECTREFS $\rightharpoonup$ OBJECTS
(stores)

$\rho \in$ INDIRECTREFS $\rightharpoonup$ VALUES
(environments)

$F(\Pi)$ ::= (INDIRECTREFS $\cup$ OBJECTREFS $\cup$ VARIABLES) $\rightharpoonup$ OBJECT PROPOSITIONS
(propositions)

Σ ::= OBJECTREFS $\rightharpoonup$ PREDICATE/INDEX PAIRS
(field store)

Expressions in the language evaluate to values, i.e., object references o. Stores μ associate object references to objects. The dynamic semantics also uses a second heap, the environment

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

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

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

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

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

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

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

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

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

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

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

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

Figure 3.4: Dynamic Semantics Rules

ρ, that connects variable references and the object store μ. The main interesting feature is the use of indirect references, a proof technique adapted from the work of Wolff *et al.* [70]. Object references and indirect references point to runtime objects: object references correspond to heap pointers, while indirect references are an artifact that make type safety easier to prove. In the source language two variables could refer to the same object in the store, but each can have different fractional permissions to that object. The environment ρ keeps track of these differences at runtime, by mapping indirect references l to values v. 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.

We keep track of object propositions in $F(\Pi)$: for all indirect references, object references and variables that point to the same object, $F(\Pi)$ will say which is the object proposition that they point to, whether that object proposition is packed or unpacked. The difference between Π and $F(\Pi)$ is that Π uses $\otimes$ as the operator that concatenates its resources (which can be object propositions, but not only), while $F(\Pi)$ is a map from references to object propositions.

Formally, we defined the linear context Π_i in the following way, where Π_i is only one of the disjunctions in the general linear context Π:

$$\begin{aligned}\Pi_i ::= \quad & \cdot \mid \Pi_i \otimes P_1 \mid \Pi_i \otimes t_1 \;\texttt{binop}\; t_2 \mid \\ & \Pi_i \otimes r.f \to x \mid \Pi_i \otimes \exists \overline{z}.P_2 \mid \Pi_i \otimes \forall \overline{z}.P_3\end{aligned}$$

where P_1 is equal to $r_1@k_1 Pred_1(\overline{t_1})$, P_2 is equal to $r_2@k_2 Pred_2(\overline{t_2})$, P_3 is equal to $r_3@k_3 Pred_3(\overline{t_3})$. We define $F(\Pi_i)$ to be

$$\begin{aligned}F(\Pi_i) ::= \quad & \cdot \mid F(\Pi_i); r_1 \rightsquigarrow r_1@k_1 Pred_1(\overline{t_1}) \\ & \mid F(\Pi_i); r_2 \rightsquigarrow r_2@k_2 Pred_2(\overline{t_2}) \mid F(\Pi_i); r_3 \rightsquigarrow r_3@k_3 Pred_3(\overline{t_3})\end{aligned}$$

where all the formal parameters have been substituted with actual parameters, and the existentially and universally quantified variables have also been substituted with actual values.

Finally in the field store Σ, for each object reference field we keep track of which is the predicate that holds a key to it, and which conjunction that predicate is in. Since we write a formula in disjunctive normal form, we need Σ in order to know the index of the conjunction where to find the predicate for a specific field.

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 judgment $\mu, \rho, e \;\; \rightarrow \;\; \mu', \rho', e'$ as a transition between store/environment/expression triples. The environment supports the proof of type safety by keeping precise track of the outstanding object propositions associated with different references to objects at runtime.

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 environment 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.

The $\vdash \mathit{Prog}$ represents the judgment that a program Prog is well formed, meaning that all methods obey their specification. The necessary helper judgments are presented below:

$$\frac{\vdash \overline{CL}}{\vdash \langle \overline{CL}, e \rangle}\ \textsc{Program}$$

$$\frac{\vdash_C \overline{M}}{\vdash \texttt{class}\ C\ \{\ \overline{T f}\ \overline{Q(\overline{x}) = R}\ \overline{M} \}}\ \textsc{Class}$$

$$\frac{\begin{array}{c} \mathit{MS} = R_1 \multimap R_2 \\ \mathit{this} : C, \overline{x : T}; R_1 \vdash e : T_r; \exists x. R_2 \end{array}}{\vdash_C T_r\ m(\overline{T\ x}) : \mathit{MS} = e}\ \textsc{Method}$$

Similar to the work of Wolff *et al.* [70], the type safety proof must verify that the outstanding object propositions for object o are mutually compatible. Figures 3.5, 3.6, 3.7 and 3.8 present the definitions needed for this. Figure 3.5 represents the outstanding object propositions that exist for different references to objects, at runtime. The *fieldProps*, *ctxProps* and *envProps* functions accumulate information for objects in the store from the fields of objects, the resource context Π and the environment respectively. The *objProps* function selects the object propositions for a particular object reference o. These definitions use square brackets to express list comprehensions and $++$ to mean list concatenation.

The *objProps* function is used to define *reference consistency*, defined in Figure 3.7, the judgment that an object in the store and all references to it are compatible. This judgment defines the consistency of the heap and environment along with the semantics of predicates, it is written $\mu, \Sigma, F(\Pi), \rho \vdash o\ \underline{ok}$ and verifies three key properties. First, all assertions from the current predicate of this object (tracked in Σ) about primitive values of fields hold ($\mathit{primitives}_{ok}$). Second, we gather up all object propositions to this object and check them for consistency. In order to prove that the $\mathit{HeapInvariants}(\overline{P})\ \mathit{hold}$, where $\overline{P} = \mathit{objProps}(\mu, \Sigma, \Pi, \rho, o)$, the conditions in Figure 3.6 have to hold, i.e., the heap should be well formed.

The $\mathit{lookup}(\Sigma(o'), C)(f_i)$ function first uses Σ to obtain the predicate $\mathit{Pred}(\bar{t})$ that holds of o'. It then looks up the defintion of the predicate Pred in the class C and substitutes the formal parameters in the definition of the predicate with the actual ones. Finally, for each field in the predicate body, it returns the object propositions that hold for that field. We need a way to obtain these interior object propositions because our predicates are recursive.

The intuition behind the first heap invariant from Figure 3.6 is that for each object, at most one predicate is unpacked for that object. The second invariant states that at all times the sum of all packed and unpacked permissions to a particular predicate on a particular object is equal to one, and the third invariant states that when looking at a snapshot of the heap at any time one should see that the predicates that are packed do not share field references.

The rules in Figure 3.9 perform variable binding and checking of primitive expressions taken from the definitions of predicates. All the primitive expressions are *ok* if the expressions made of the values that the fields point to check in the Π context. Figure 3.9 presents judgment rules for checking of terms that are in a binary expression, for checking of the simultaneous occurence of values in an expression in our linear theory, for the binding of fields and of variables. The

$$\begin{array}{rcl} objProps(\mu,\Sigma,\Pi,\rho,o) & = & [P|o:P\in props(\mu,\Sigma,\Pi,\rho)] \\ props(\mu,\Sigma,\Pi,\rho) & = & fieldProps(\mu)\ ++\ envProps(\Sigma,\Pi,\rho)\ ++\ ctxProps(\Pi) \\ fieldProps(\mu,\Sigma) & = & ++_{o'\in dom(\mu)}\,[o_i:P_i|\mu(o')=C(\overline{o}), P_i\in lookup(\Sigma(o'),C)(f_i)] \\ envProps(\Sigma,\Pi,\rho) & = & [o:P\mid \rho(l)=o, l:P\in F(\Pi)] \\ ctxProps(\Pi) & = & [o:P\mid o:P\in F(\Pi)] \end{array}$$

Figure 3.5: Outstanding Object Propositions

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

Figure 3.6: Definition for $HeapInvariants(\overline{P})$

$bindFields()$ function does not perform the computation. It is only a rewriting function that replaces the formal variables with actual values for everything in a formula R representing the body of a predicate, except for the object propositions, which are not considered primitives. More specifically, in the rules R_i is rewritten to R_i', where formal parameters are replaced by actual values.

Finally, we check consistency for each reference o in Figure 3.8. *Global consistency* establishes the intrinsic compatibility of a store-environment-context triple. It checks that every object reference in the store satisfies reference consistency, that every reference in the object proposition context $F(\Pi)$ is accounted for in the store and environment, and that indirect references ultimately point to object references. In Figure 3.8, by *ran(f)* we refer to the range of the function argument f and by *dom(f)* we refer to the domain of function f.

$$\frac{\begin{array}{c}\mu(o)=C(\overline{o'}) \\ |o'| = |fields(C)| \\ objProps(\mu,\Sigma,\Pi,\rho,o)=\overline{P} \\ primitives_{ok}(\Pi, lookup(\Sigma(o),C),\mu,o) \\ HeapInvariants(\overline{P})\ hold\end{array}}{\mu,\Sigma,F(\Pi),\rho\vdash o\ \underline{ok}}$$

Figure 3.7: Reference Consistency

$$\frac{\begin{array}{c} ran(\rho)\ \subset\ dom(\mu) \\ dom(F(\Pi))\ \subset\ dom(\rho)\ \cup\ dom(\mu) \\ \{l \mid (l:P) \in F(\Pi)\} \subset \{l \mid \rho(l) = o\} \\ \mu, \Sigma, F(\Pi), \rho \vdash dom(\mu)\ \underline{ok} \end{array}}{\mu, \Sigma, F(\Pi), \rho\ \underline{ok}}$$

Figure 3.8: Object Proposition Consistency

$$\frac{\begin{array}{c} bindFields(R, R, \mu, o) \equiv \overline{\oplus(\overline{\otimes\,(v_1\ binop\ v_2) \otimes \overline{(v_1\ binop\ v_2) \Rightarrow v_3)}}} \\ \Pi \vdash \overline{\oplus(\overline{\otimes\,(v_1\ binop\ v_2) \otimes \overline{(v_1\ binop\ v_2) \Rightarrow v_3)}}} \\ R\ is\ the\ definition\ body\ of\ a\ packed\ predicate \end{array}}{primitives_{ok}(\Pi, R, \mu, o)}\ primitives_{ok}$$

$$\frac{bindFields(t_i, R, \mu, o) \equiv v_i\ \ i = 1, 2}{bindFields(t_1\ binop\ t_2, R, \mu, o) \equiv v_1\ binop\ v_2}\ binop_{ok}$$

$$\frac{bindFields(R_i, R, \mu, o) \equiv R'_i\ \ i = 1, 2}{bindFields(R_1\ \otimes\ R_2, R, \mu, o) \equiv R'_1 \otimes R'_2}\ and_{ok}$$

$$\frac{bindFields(R_i, R_i, \mu, o) \equiv R'_i\ \ i = 1, 2}{bindFields(R_1\ \oplus\ R_2, R, \mu, o) \equiv R'_1 \oplus R'_2}\ or_{ok}$$

$$\frac{\begin{array}{c} bindFields(t_i, R, \mu, o) \equiv v_i\ \ i = 1, 2 \\ bindFields(R_3, R, \mu, o) \equiv R'_3 \end{array}}{bindFields((t_1\ binop\ t_2) \Rightarrow R_3, R, \mu, o) \equiv (v_1\ binop\ v_2) \Rightarrow R'_3}\ implies_{ok}$$

$$\frac{}{bindFields(f \to x, R, \mu, o) \equiv v}\ field_{ignored}$$

$$\frac{\begin{array}{c} \mu[o, f] = v \\ f \to x \in R \end{array}}{bindFields(x, R, \mu, o) \equiv v}\ var_{ok}$$

$$\frac{}{bindFields(r@k\ Q(\bar{t}), R, \mu, o) \equiv \_}\ objprop1_{ignored}$$

$$\frac{}{bindFields(unpacked(r@k\ Q(\bar{t})), R, \mu, o) \equiv \_}\ objprop2_{ignored}$$

Figure 3.9: Rules for $primitives_{ok}$

3.5 Soundness Proof of Formal Object Proposition Rules

Now we state the main preservation theorem that underlies the soundness of our system:
Theorem 1. (Preservation)

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

The proof given below uses induction over the derivation of $\mu, \rho, e \to \mu', \rho', e'$ in the standard way.

Our system inherits a Progress property from related object calculi such as Featherweight Java. In the following sections we first present the Substitution Lemma and its proof, the Memory Consistency lemma and its proof and finally the proof of the Preservation Theorem.

3.5.1 Substitution Lemma

Lemma 1. (Substitution) *If* $\Gamma; R_1 \vdash e : T; \exists x.R$ *and* $\Gamma \vdash \bar{l} : \overline{T_1}$ *and* $\Gamma \vdash \overline{y} : \overline{T_1}$ *then* $\Gamma; ([\bar{l}/\overline{y}]R_1) \vdash [\bar{l}/\overline{y}]e : T; \exists x.[\bar{l}/\overline{y}]R$*, where* $\bar{l}$ *replaces all the formal variables* $\overline{y}$ *in* e.

Proof of Substitution Lemma

The proof is by induction on the derivation of $\Gamma; R_1 \vdash e : T; \exists x.R$. 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, R_1 and R 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 $n|null|true|false$. We know $\Gamma; R_1 \vdash v : T; \exists x.R$. Since v is a value, $[\bar{l}/\overline{y}]e = v$. We now have to prove that $\Gamma; ([\bar{l}/\overline{y}]R_1) \vdash v : T; [\bar{l}/\overline{y}]R$. We trivially obtain this by simply renaming $\overline{y}$ to $\bar{l}$, assuming that $\bar{l}$ are fresh variables.

2. e is a variable z, $z \neq x_i$. The proof in this case is very similar to Case 2.

3. e is the variable y. In this case we have only one variable y and one variable l. Now $[l/y]e = l$ and $T_1 = T$.

   From the premise we know that $\Gamma \vdash l : T$ and that $\Gamma; R_1 \vdash R$. By renaming y to l we obtain that $\Gamma; [l/y]R_1 \vdash [l/y]R$. We use the static rule (TERM) and we obtain that $\Gamma; ([l/y]R_1) \vdash l : T; \exists x.[l/y]R$.

4. e is $r.f_i$. We know that $\Gamma; R_1 \vdash r.f_i : T; \exists x.R$. We also know by inversion that $R_1 \vdash r.f_i \to r_i$ and that $\Gamma; R_1 \vdash [r_i/x]R$. Using the induction hypothesis we have: $([l/y]R_1) \vdash [l/y](r.f_i \to r_i)$ and $\Gamma; ([l/y]R_1) \vdash [l/y][r_i/x]R$. Since $r.f_i$ is just the syntactic representation of a field, the substitution will happen in r_i: $([l/y]R_1) \vdash (r.f_i \to [l/y]r_i)$. Also, we can rewrite $[l/y][r_i/x]R$ as $[([l/y]r_i)/x][l/y]R$ and so we have $\Gamma; ([l/y]R_1) \vdash [([l/y]r_i)/x][l/y]R$. Using the rule (FIELD), we obtain that $\Gamma; ([l/y]R_1) \vdash [l/y]r.f_i : T; \exists x.[l/y]R$, exactly what we wanted.

5. e is $\texttt{new}\ C(\overline{t_1})$. We know $\Gamma; R_1 \vdash \texttt{new}\ C(\overline{t_1}) : \exists z.C; R$, where R is equal to $z.\overline{f} \to \overline{t_1} \otimes R_1$. We also know by inversion that $\Gamma \vdash \overline{t_1 : T}$. Using the induction hypothesis we have $\Gamma \vdash \overline{[l/y]t_1 : T}$. Using the rule (NEW), we obtain that

$\Gamma; ([l/y]R_1) \vdash [l/y]\texttt{new}\ C(\overline{t_1}) : \exists z.C; z.\overline{f} \rightarrow \overline{[l/y]t_1} \otimes [l/y]R_1$, which means
$\Gamma; ([l/y]R_1) \vdash [l/y]\texttt{new}\ C(\overline{t_1}) : \exists z.C; [l/y](z.\overline{f} \rightarrow \overline{t_1} \otimes R_1)$
exactly what we wanted.

6. e is $\texttt{if}(t_1)\{e_1\}\texttt{else}\{e_2\}$. We know $\Gamma; R_1 \vdash \texttt{if}(t_1)\{e_1\}\texttt{else}\{e_2\} : \exists x.T; R_3 \oplus R_2$. We also know by inversion that $\Gamma; (R_1 \otimes t_1 = \mathit{true}) \vdash e_1 : \exists x.T; R_3$ and that $\Gamma; (R_1 \otimes t_1 = \mathit{false}) \vdash e_2 : \exists x.T; R_2$. Using the induction hypothesis, we have $\Gamma; ([l/y]R_1 \otimes t_1 = \mathit{true}) \vdash [l/y]e_1 : \exists x.T; [l/y]R_3$ and that $\Gamma; ([l/y]R_1 \otimes t_1 = \mathit{false}) \vdash [l/y]e_1 : \exists x.T; [l/y]R_2$. By applying the (IF) rule, we obtain that $\Gamma; ([l/y]R_1) \vdash \texttt{if}(t_1)\{e_1\}\texttt{else}\{e_2\} : \exists x.T; [l/y]R_3 \oplus [l/y]R_2$. Since $[l/y]R_3 \oplus [l/y]R_2$ is equal to $[l/y](R_3 \oplus R_2)$ we obtain that $\Gamma; ([l/y]R_1) \vdash \texttt{if}(t_1)\{e_1\}\texttt{else}\{e_2\} : \exists x.T; [l/y](R_3 \oplus R_2)$, exactly what we wanted.

7. e is $\texttt{let}\ x = e_1\ \texttt{in}\ e_2$. We know $\Gamma; R_1 \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \exists w.T; R$. We also know by inversion that $\Gamma; R_1 \vdash e_1 : \exists x.T_1; R_2$. Using the induction hypothesis, we obtain that $\Gamma; ([l/y]R_1) \vdash [l/y]e_1 : \exists x.T_1; [l/y]R_2$. We also know by inversion that $(\Gamma, x : T_1); R_2 \vdash e_2 : \exists w.T; R$. By applying induction to this judgment we obtain that $(\Gamma, x : T_1); [l/y]R_2 \vdash [l/y]e_2 : \exists w.T; [l/y]R$. Now, we can apply the (LET) rule and we obtain that $\Gamma; ([l/y]R_1) \vdash [l/y](\texttt{let}\ x = e_1\ \texttt{in}\ e_2) : \exists w.T; [l/y]R$, which is exactly what we wanted.

8. e is $\texttt{pack}\ r@k\ Q(\overline{t_1})\ \texttt{in}\ e_1$, with $0 < k < 1$. We know that $\Gamma; R_1 \vdash \texttt{pack}\ r@k\ Q(\overline{t_1})\ \texttt{in}\ e_1 : \exists x.T; R$. We also know by inversion that $\Gamma; (R_1 \otimes r@k\ Q(\overline{t_1})) \vdash e_1 : \exists x.T; R$.

   Using the induction hypothesis we obtain that $\Gamma; [l/y](R_1 \otimes r@k\ Q(\overline{t_1})) \vdash [l/y]e_1 : \exists x.T; [l/y]R$, i.e.,

   $\Gamma; [l/y]R_1 \otimes ([l/y]r)@k\ Q([l/y]\overline{t_1}) \vdash [l/y]e_1 : \exists x.T; [l/y]R$. The other two premises of the (PACK2) rule can also be obtained by inversion. From the first premise $\Gamma; R_1 \vdash r : C; [\overline{t_1}/\overline{x}]R_2 \otimes \textsf{unpacked}(r@k\ Q(\overline{t_1})) \otimes \Pi_1$ we can deduce by induction that

   $\Gamma; ([l/y]R_1) \vdash [l/y]r : C; [l/y]([\overline{t_1}/\overline{x}]R_2 \otimes \textsf{unpacked}(r@k\ Q(\overline{t_1})) \otimes \Pi_1)$, i.e.,

   $\Gamma; ([l/y]R_1) \vdash [l/y]r : C; [l/y]([\overline{t_1}/\overline{x}]R_2) \otimes \textsf{unpacked}(([l/y]r)@k\ Q([l/y]\overline{t_1})) \otimes ([l/y]\Pi_1)$

   The second premise is $\texttt{predicate}\ Q(\overline{Tx}) \equiv R_2 \in C$.

   So now we can apply the (PACK2) rule and we get that

   $\Gamma; ([l/y]R_1) \vdash (\texttt{pack}\ ([l/y]r)@k\ Q([l/y]\overline{t_1}))\ \texttt{in}\ [l/y]e_1 : \exists x.T; [l/y]R$.

   Thus, $\Gamma; ([l/y]R_1) \vdash [l/y](\texttt{pack}\ r@k\ Q(\overline{t_1})\ \texttt{in}\ e_1) : \exists x.T; [l/y]R$, exactly what we wanted.

9. e is $\texttt{pack}\ r@1\ Q_2(\overline{t_2})\ \texttt{in}\ e_1$. 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 $\texttt{unpack}\ r@k\ Q(\overline{t_1})\ \texttt{in}\ e_1$ for $0 < k < 1$. We know that

    $\Gamma; R_1 \vdash \texttt{unpack}\ r@k\ Q(\overline{t_1})\ \texttt{in}\ e_1 : \exists x.T; R$. We also know by inversion that $\Gamma; (\Pi_1 \otimes [\overline{t_1}/\overline{x}]R_2 \otimes \textsf{unpacked}(r@k\ Q(\overline{t_1})) \vdash e_1 : \exists x.T; R$.

    Using the induction hypothesis, we obtain that

    $\Gamma; [l/y](\Pi_1 \otimes [\overline{t_1}/\overline{x}]R_2 \otimes \textsf{unpacked}(r@k\ Q(\overline{t_1})) \vdash [l/y]e : \exists x.T; [l/y]R$, i.e.,

    $\Gamma; [l/y]\Pi_1 \otimes [[l/y](\overline{t_1}/\overline{x})]R_2 \otimes \textsf{unpacked}([l/y]r@k\ Q([l/y]\overline{t_1}) \vdash [l/y]e : \exists x.T; [l/y]R$

    The other premises of the (UNPACK2) rule can also be obtained by inversion. From the first premise $\Gamma; R_1 \vdash r : C; r@k\ Q(\overline{t_1}) \otimes \Pi_1$ we obtain by induction that $\Gamma; ([l/y]R_1) \vdash [l/y]r : C; [l/y](r@k\ Q(\overline{t_1}) \otimes \Pi_1)$, i.e.,

$\Gamma; ([l/y]R_1) \vdash [l/y]r : C; [l/y]r@k\ Q([l/y]\overline{t_1}) \otimes [l/y]\Pi_1$

The second premise is `predicate` $Q(\overline{Tx}) \equiv R_2 \in C$.

Now we can apply the (UNPACK2) rule and we get that

$\Gamma; ([l/y]R_1) \vdash$ `unpack` $[l/y]r@k\,Q([l/y]\overline{t_1})$ `in` $[l/y]e_1 : \exists x.T; [l/y]R$. Thus $\Gamma; ([l/y]R_1) \vdash [l/y]($`unpack` $r@k\ in\ Q(\overline{t_1}))$ `in` $[l/y]e_1 : \exists x.T; [l/y]R$, exactly what we wanted to prove.

11. e is `unpack` $r@1\ Q(\overline{t_1})$ `in` e_1. 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 $\Gamma; R_1 \vdash r_0.m(\overline{t_1}) : T_r; R$ where R is

$\exists\ \textit{result}.T_r; [r_0/this][\overline{t_1}/\overline{x}]R \otimes \Pi_1$. We know by inversion that $\Gamma; R_1 \vdash [r_0/this][\overline{t_1}/\overline{x}]R_1 \otimes \Pi_1$. Using the induction hypothesis, we obtain that $\Gamma; ([l/y]\underline{R_1) \vdash} [l/y]([r_0/this][\overline{t_1}/\overline{x}]R_1 \otimes \Pi_1)$. This is equivalent to writing $\Gamma; ([l/y]R_1) \vdash [r_0/this][\overline{[l/y]t_1}/\overline{x}][l/y]R_1 \otimes [l/y]\Pi_1$.

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 $\Gamma \vdash [l/y]r_0 : C_0$ and $\Gamma \vdash \overline{[l/y]t_1 : T}$. Also by inversion we know that $mtype(m, C_0) = \forall \overline{x : T}.\exists result.T_r; R_1' \multimap R_2$.

We can infer that $mtype(m, C_0) = \forall \overline{x : T}.\exists result : T_r.R_1 \multimap R_2$ will hold for $\overline{[l/y]t_1 : T}$ (because of the $\forall$ quantifier of the (MTYPE) judgement). We can now apply the (CALL) rule and we obtain that $\Gamma; ([l/y]R_1) \vdash (([l/y]r_0).m(\overline{[l/y]t_1}))$:

$\exists\ \textit{result}.T_r; [[l/y]r_0/this][\overline{[l/y]t_1}/\overline{x}][l/y]R \otimes [l/y]\Pi_1$.

Since $r_0.m(\overline{[l/y]t_1})$ is equal to $[l/y](r_0.m(\overline{t_1}))$ and $[r_0/this][\overline{[l/y]t_1}/\overline{x}][l/y]R$ is equal to $[l/y]([r_0/this][\overline{t_1}/\overline{x}]R)$, we obtain that

$\Gamma; ([l/y]R_1) \vdash [l/y](r_0.m(\overline{t_1})) : \exists\ \textit{result}.T_r; [l/y]([r_0/this][\overline{t_1}/\overline{x}]R \otimes \Pi_1)$. This is exactly what we wanted to prove.

13. e is $r_1.f_i = t_1$. We know that $\Gamma; R_1 \vdash (r_1.f_i = t_1) : \exists x.T_i; x@k'\ Q'(\overline{t'}) \otimes t_1@k_0\ Q_0(\overline{t_0}) \otimes r_1.f_i \rightarrow t_1 \otimes \Pi_3$. We know by inversion that $\Gamma; R_1 \vdash t_1 : T_i; t_1@k_0\ Q_0(\overline{t_0}) \otimes \Pi_1$. Using the induction hypothesis, we obtain that $\Gamma; ([l/y]R_1) \vdash [l/y]t_1 : T_i; [l/y](t_1@k_0\ Q_0(\overline{t_0}))$. This means that $[l/y](t_1@k_0\ Q_0(\overline{t_0})) = ([l/y]t_1)@k_0\ [l/y]Q_0(\overline{t_0})$. 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 $\Gamma; ([l/y]R_1) \vdash r_1.f_i : T_i; [l/y](r_i'@k'\ Q'(\overline{t'}) \otimes \Pi_2)$. From the third premise $\Pi_2 \vdash r_1.f_i \rightarrow r_i' \otimes \Pi_3$ we get by induction that $([l/y]\Pi_2) \vdash r_1.f_i \rightarrow [l/y]r_i' \otimes [l/y]\Pi_3$. So now we can apply the (ASSIGN) rule and we get that $\Gamma; ([l/y]R_1) \vdash r_1.f_i = [l/y]t_1 : \exists x.T_i; [l/y](x@k'\ Q'(\overline{t'})) \otimes [l/y](t@k_0\ Q_0(\overline{t_0})) \otimes r_1.f_i \rightarrow [l/y]t_1)$. Since $(r_1.f_i = [l/y]t_1)$ is equal to $([l/y](r_1.f_i = t_1))$ and $[l/y](x@k'\ Q'(\overline{t'}) \otimes [l/y](t_1@k_0\ Q_0(\overline{t_0})) \otimes r_1.f_i \rightarrow [l/y]t_1$ is equal to $[l/y](x@k'\ Q'(\overline{t'}) \otimes t@k_0\ Q_0(\overline{t_0}) \otimes r_1.f_i \rightarrow t_1)$, we finally obtain that

$\Gamma; ([l/y]R_1) \vdash [l/y](r_1.f_i = t_1) : \exists x.T_i; [l/y](x@k'\ Q'(\overline{t'}) \otimes t_1@k_0\ Q_0(\overline{t_0}) \otimes r_1.f_i \rightarrow t_1 \otimes \Pi_3)$. 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.

3.5.2 Framing Lemma

If $\Gamma, \Pi_1 \vdash e : T;\ \Pi_2$ and Π_3 contains no unpacked predicates, then $\Gamma, \Pi_1 \otimes \Pi_3 \vdash e : T; \Pi_2 \otimes \Pi_3$.

The proof is straightforward following familiar patterns, similar to the proof of Lemma 8 from page L18.6 of Prof. Frank Pfenning's Lecture 18 on linear logic [63].

3.5.3 Memory Consistency Lemma

We also need to define the Memory Consistency lemma below, which could be seen as a framing lemma that proves that the execution context for each step in the execution is sound. The cases of the Memory Consistency lemma follow the dynamic semantics rules, except for the ommission of the (LET-E), (LET-V), (IF-TRUE) and (IF-FALSE) cases. We ommitted the (LET-E) case because the Preservation Theorem, that we prove by induction on the dynamic semantics rules, uses the inductive argument and the Substitution Lemma in its (LET-E) case and thus we do not need the help of the Memory Consistency lemma. Similarly we use the Substitution lemma in the proof of the (LET-V) case of the Preservation Theorem. In the (IF-TRUE) and (IF-FALSE) cases we start with a larger context that is restricted by one branch or the other of the *if* statement, and hence it is still sound.

Lemma 2. (Memory Consistency)

1. *(*LOOKUP*) 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. *(*NEW*) If* $\mu, \Sigma, F(\Pi), \rho\ \underline{ok}$ *and* $o \notin dom(\mu)$*, then* $\mu' = \mu[o \rightsquigarrow C(\overline{\rho(l)})], \Sigma' = (\Sigma), F(\Pi') = F(\Pi), \rho\ \underline{ok}$*.*
3. *(*LET-O*) 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. *(*PACK-2*) 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. *(*PACK-1*) If* $\mu, (\Sigma_1, \Sigma_2), (F(\Pi_1 \otimes \Pi_2)), \rho\ \underline{ok}$ *and* $unpacked(r@1\ Q_1(\overline{t_1})) \in \Pi_1$*, then*
   $\mu, (\Sigma_2, r \rightsquigarrow Q_2(\overline{t_2})), (F(\Pi_2), r \rightsquigarrow r@1\ Q_2(\overline{t_2})), \rho\ \underline{ok}$*.*
6. *(*UNPACK-2*) 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* $\mu, \Sigma' = (\Sigma_2, r \rightsquigarrow unpacked)$,
   $F(\Pi') = (F(\Pi_2), F([\overline{t_1}/\overline{x}]R_1 \otimes unpacked(r@k\ Q(\overline{t_1}))))$,
   $\rho\ \underline{ok}$*.*
7. *(*UNPACK-1*) If* $\mu, (\Sigma_0, \Sigma_2), (F(\Pi_0 \otimes \Pi_2)), \rho\ \underline{ok}$ *and* $r@1\ Q(\overline{t_1}) \in \Pi_0$
   and `predicate` $Q(\overline{T\ x}) \equiv R_1 \in C$
   then $\mu, \Sigma' = (\Sigma_2, r \rightsquigarrow unpacked)$,
   $F(\Pi') = (F(\Pi_2), F([\overline{t_1}/\overline{x}]R_1 \otimes unpacked(r@1\ Q(\overline{t_1}))))$,
   $\rho\ \underline{ok}$*.*
8. *(*FIELD*) If* $\mu, \Sigma, F(\Pi), \rho\ \underline{ok}$ *and* $\mu(\rho(l)) = C(\overline{o})$ *and* $fields(C) = \overline{Tf}$ *then* $\mu, \Sigma' = (\Sigma, o_i \rightsquigarrow Q), F(\Pi') = (F(\Pi), o_i \rightsquigarrow R), \rho\ \underline{ok}$*, where* $R = x@k\ Q$*.*

9. *(*ASSIGN*) If $\mu, \Sigma = (\Sigma_0, l_2 \rightsquigarrow Q'(\overline{t'})), F(\Pi) = (F(\Pi_0), l_2 \rightsquigarrow l_2@k'\ Q'(\overline{t'}) \otimes x@k_0\ Q_0(\overline{t_0})), \rho\ \underline{ok}$, $l_1.f_i \to x$ and $\rho(l_2) = o_2$, then when $l_1.f_i = l_2$ we have:*
   $\mu' = \mu[\rho(l_1) \rightsquigarrow [o_2/o_i]C(\overline{o})], \Sigma' = (\Sigma, o_2 \rightsquigarrow Q'(\overline{t'}))$,
   $F(\Pi') = (F(\Pi), o_2 \rightsquigarrow o_2@k'\ Q'(\overline{t'}) \otimes$
   $x@k_0\ Q_0(\overline{t_0})), \rho\ \underline{ok}$
10. *(*INVOKE*) If $\mu, \Sigma, F(\Pi), \rho\ \underline{ok}$ where $\Pi = (R_1 \otimes \Pi_1)$ and $R_1 \multimap R$, then $\mu, \Sigma', F(\Pi'), \rho\ \underline{ok}$ where $\Pi' = (R \otimes \Pi_1)$ and Σ' corresponds to Π'. Π_1 does not contain unpacked predicates.*

Proof of memory consistency lemma

1. (LOOKUP)

   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)

   Assuming $\mu, \Sigma, F(\Pi), \rho\ \underline{ok}$ and $o \notin dom(\mu)$, we have to show that

   $\mu' = \mu[o \rightsquigarrow C(\overline{\rho(l)})], \Sigma' = (\Sigma), F(\Pi') = F(\Pi), \rho\ \underline{ok}$. It must be that $\overline{\rho(l) = o'}$ for some objects o'. The objects o' are being referred to from one more place now, from the fields of o but their object propositions do not change. The only objects affected are $o, \overline{o'}$. Since $\overline{\mu(o') = \mu'(o')}$ and

   $\overline{props(\mu, \Sigma, F(\Pi), \rho, o') = props(\mu', \Sigma', F(\Pi'), \rho, o')}$ we can deduce that $\overline{\mu', \Sigma', F(\Pi'), \rho \vdash\ o'\ \underline{ok}}$.

   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. (LET-O)

   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. (PACK-2)

   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 $\Omega_1\ \underline{ok}$, 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 at most 1. Π_2 could also contain object propositions of the form $\mathit{unpacked}(r@k_1\ Q_i())$, with $Q_i \neq Q$, 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. (PACK-1)

   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 $\mathit{objProps}$ 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 $F(\Pi_2)$.

   Since Ω_1 $\underline{ok}$ and $\mathit{unpacked}(r@1\ Q_1(\overline{t_1})) \ \in \ \Pi_1$, there are no more object propositions about r and the predicate Q_2 in Π_2. Π_2 could also contain object propositions of the form $\mathit{unpacked}(r@k_1\ Q_i())$, with $Q_i \neq Q$, but the fields in the predicates are disjoint, according to the heap invariants. Thus, $(\Pi_2, r \rightsquigarrow r@1\ Q_2(\overline{t_2}))$ satisfies the heap invariants, Σ' is compatible with Π' and the primitives are preserved, so μ, Σ', Π', ρ $\underline{ok}$.

6. (UNPACK-2)

   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
   $\Omega_2 = [\mu, \Sigma' = (\Sigma_2, r \rightsquigarrow \mathit{unpacked}),$
   $F(\Pi') = (F(\Pi_2), [\overline{t_1}/\overline{x}]R_1), \rho]$ $\underline{ok}$.

   Let's take an arbitrary o. Since μ and ρ don't change, the only changes in the $\mathit{objProps}$ 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 $\forall r', \bar{t} : (\ \mathit{unpacked}(r'@k'\ Q(\bar{t})) \in (\Pi_0 \cup \Pi_2) \Rightarrow \Pi_0, \Pi_2 \vdash r \neq r')$ 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. Note that when we unpack a predicate we do not get information if the recursive internal object propositions of that predicate are packed or not, we only get the corresponding fraction. Thus, $\mu, \Sigma', F(\Pi'), \rho\ \underline{ok}$

7. (UNPACK-1)

   The proof of this case is very similar to the proof of the previous case UNPACK-2.

8. (FIELD)

   Assuming $\mu, \Sigma, F(\Pi), \rho\ \underline{ok}$ and $\mu(\rho(l)) = C(\overline{o})$ and

   $fields(C) = \overline{T f}$, 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. (ASSIGN)

   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.

   By being able to assign to a field it means that the object proposition from which the field was taken is already unpacked. By changing the value of that field we only influence that unpacked predicate. Nevertheless the primitives have to be *ok* only for packed predicates, as the unpacked predicates can have their primitive operations not hold temporarily.

   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 $\mu', \Sigma', F(\Pi'), \rho\ \underline{ok}$.

10. (INVOKE) We first want to prove that the three heap invariants still hold. The first invariant will hold because Π_1 does not contain any unpacked predicates and R is obtained by entailment from $R1$ which satisfies the $\underline{ok}$ judgment.

    For the second invariant: the fractions that we get out in the post-condition R are the same as the fractions that we put in the pre-condition R_1 and we know that $\mu, \Sigma, F(R_1 \otimes \Pi_1), \rho\ \underline{ok}$.

    For the third invariant: there is no change in the definition of the predicates and hence this heap invariant will hold.

    Since we do not allow unpacked predicates in Pi_1 and we know that the primitives were holding for the pre-condition R_1, the primitives will hold.

This was all that we needed to prove reference consistency as defined in Figure 3.7 and hence $\mu, \Sigma', F(\Pi'), \rho\ \underline{ok}$.

3.5.4 Proof of Preservation Theorem

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

Case (LOOKUP)

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

1. By assumption
   (a) $\Gamma, \Pi \ \vdash\ l : \exists\, x.T; R$
   (b) $\mu, \Sigma, F(\Pi), \rho\ \underline{ok}$
   (c) $\mu, \rho, l\ \to\ \mu, \rho, \rho(l)$
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 field store 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 from Figure 3.8
5. $(\Gamma, o : T), (\Pi_1, o \rightsquigarrow R) \vdash o : \exists x.T; R$ -by the (TERM) proof rule
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, the LOOKUP case of the Memory Consistency lemma
9. $\mu, \Sigma', F(\Pi'), \rho\ \underline{ok}$ -by 6, 8
10. q.e.d -by 7, 9

Case (NEW)

To prove: If $\Gamma, \Pi \vdash new\ C(\bar{l})\ : \exists y.T; R$ and $\mu, \Sigma, F(\Pi), \rho\ \underline{ok}$ and $\mu, \rho, new\ C(\bar{l})\ \to\ \mu', \rho, o$ and $\vdash Prog$ then there exists Π' , Γ' and Σ' such that $\Gamma', \Pi' \vdash o : \exists y.T; R$ and $\mu', \Sigma', F(\Pi'), \rho\ \underline{ok}$.

1. By assumption
   (a) $\Gamma, \Pi \vdash new\ C(\bar{l})\ : \exists y.T; R$
   (b) $\mu, \Sigma, F(\Pi), \rho\ \underline{ok}$
   (c) $\mu, \rho, new\ C(\bar{l}) \to \mu', \rho, o$
   (d) $o \notin dom(\mu)$
   (e) $\mu' = \underline{\mu[o} \to C(\overline{\rho(l)})]$. The intuition here is that if the notation $\bar{l}$ denotes $l_1, l_2, ..., l_n$ then $\overline{\rho(l)}$ denotes $\rho(l_1), \rho(l_2), ..., \rho(l_n)$.
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 $\Gamma' = (\Gamma, o : C), F(\Pi') = (F(\Pi_1), o \rightsquigarrow (\overline{o.f \rightarrow t}))$, where this means $o.f_1 \rightarrow t_1 \otimes o.f21 \rightarrow t_2 \otimes ....$
4. Let $\Sigma' = (\Sigma, o \rightsquigarrow \textsf{unpacked})$
5. $\Gamma', \Pi' \vdash o : \exists z.C; z.\overline{f} \rightarrow \overline{t} \otimes \Pi_1$ -by (TERM)
6. $\mu[o \rightsquigarrow C(\overline{\rho(l)})], (\Sigma, o \rightsquigarrow \textsf{unpacked}), (F(\Pi_1), o \rightsquigarrow (o.\overline{f} \rightarrow \overline{t})), \rho\ \underline{ok}$ -by memory consistency lemma
7. q.e.d. -by 5, 6

Case (LET-O)

To prove: If $\Gamma, \Pi \vdash let\ x = o\ in\ e_2 : \exists\ w.T_2; R_3$ and $\mu, \Sigma, F(\Pi), \rho\ \underline{ok}$ and $\mu, \rho, let\ x = o\ in\ e_2 \ \rightarrow\ \mu, \rho[l \rightsquigarrow o], [l/x]e_2$ and $\vdash\ Prog$ then there exists Π' , Γ' and Σ' such that $\Gamma', \Pi' \vdash [l/x]e_2 : T; \exists\ w.T_2; R_3$ and $\mu, \Sigma', F(\Pi'), \rho[l \rightsquigarrow o]\ \underline{ok}$.

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) $(\Gamma, x : T_1); (R_1 \otimes \Pi_2) \vdash e_2 : \exists w.T_2; R_3$
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. $(\Gamma, l : T_1); (\Pi_2 \otimes R_1) \vdash [l/x]e_2 : \exists w.T_2; [l/x]R_3$ -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 the LET-O case of the Memory Consistency lemma
10. $\mu, \Sigma', F(\Pi'), \rho[l \rightsquigarrow o]\ \underline{ok}$
11. q.e.d. -by 10, 7

Case (LET-E)

To prove: If $\Gamma, \Pi \vdash let\ x = e_1\ in\ e_2 : \exists w.T_2; R_3$ and $\mu, \Sigma, F(\Pi), \rho\ \underline{ok}$ and $\mu, \rho, let\ x = e_1\ in\ e_2 \ \rightarrow\ \mu', \rho', let\ x = e'\ in\ e_2$ and $\vdash\ Prog$ then there exists Π' , Γ' and Σ' such that $\Gamma', \Pi' \vdash let\ x = e'\ in\ e_2 : \exists w.T_2; R_3$ and $\mu', \Sigma', F(\Pi'), \rho'\ \underline{ok}$.

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) $(\Gamma, x : T_1); (R_1 \otimes \Pi_2) \vdash e_2 : \exists w.T_2; R_3$
3. By induction on 1b, 1d, 2a
   (a) $\exists \Gamma_0; \Pi'$ such that $\Gamma_0, \Pi' \vdash e' : \exists x.T_1; R_1 \otimes \Pi_2$
   (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, the (LET) static semantics proof rule
6. q.e.d. -by 3b,5

Case (LET-V) To prove: If $\Gamma, \Pi \vdash let\ x = v\ in\ e_2 : \exists\ w.T_2; R_3$ and $\mu, \Sigma, F(\Pi), \rho\ \underline{ok}$ and $\mu, \rho, let\ x = v\ in\ e_2 \rightarrow \mu, \rho, [v/x]e_2$ and $\vdash Prog$ then there exists Π' , Γ' and Σ' such that $\Gamma', \Pi' \vdash [v/x]e_2 : T; \exists\ w.T_2; R_3$ and $\mu, \Sigma', F(\Pi'), \rho\ \underline{ok}$.

1. By assumption
   (a) $\Gamma, \Pi \vdash let\ x = v\ in\ e_2 : \exists\ w.T_2; R_3$
   (b) $\mu, \Sigma, F(\Pi), \rho\ \underline{ok}$
   (c) $\mu, \rho, let\ x = v\ in\ e_2 \rightarrow \mu, \rho, [v/x]e_2$
   (d) $v \in \{n, true, false\}$
2. By inversion on 1a
   (a) $\Gamma; \Pi \vdash v : \exists x.T_1; \Pi$
   (b) $(\Gamma, x : T_1); \Pi \vdash e_2 : \exists w.T_2; R_3$
3. $\Gamma = (\Gamma_1, v : T_1)$ -by inversion on 2a
4. Let $\Gamma' = \Gamma$, $F(\Pi') = F(\Pi)$, $\Sigma' = \Sigma$
5. $\Gamma; \Pi \vdash [v/x]e_2 : \exists w.T_2; [v/x]R_3$ -by 2b and Substitution Lemma
6. $\mu, \Sigma', F(\Pi'), \rho\ \underline{ok}$
7. q.e.d. -by 6, 5

Case (PACK)

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

To prove: If $\Gamma, \Pi \vdash pack\ r@k\ Q(\overline{t_1})\ in\ e_1 : \exists x.T; R$ and
$\mu, \Sigma, F(\Pi), \rho\ \underline{ok}$ and $\mu, \rho, pack\ r\ to\ R_1\ in\ e_1 \rightarrow \mu, \rho, e_1$ and $\vdash Prog$ then there exists Π' , Γ' and Σ' such that $\Gamma', \Pi' \vdash e_1 : \exists x.T; R$ and $\mu, \Sigma', F(\Pi'), \rho\ \underline{ok}$.

1. By assumption

(a) $\Gamma, \Pi \vdash \mathit{pack}\ r@k\ Q(\overline{t_1})\ \mathit{in}\ e_1 : \exists x.T; R$

(b) $\mu, \Sigma, F(\Pi), \rho\ \underline{ok}$

(c) $\mu, \rho, \mathit{pack}\ r@k\ Q(\overline{t_1})\ \mathit{in}\ e_1 \rightarrow \mu, \rho, e_1$

2. By inversion on (PACK2)

   (a) $\Gamma; \Pi \vdash r : C.[\overline{t_1}/\overline{x}]R_1 \otimes$
   $\textsf{unpacked}(r@k\ Q(\overline{t_1})) \otimes \Pi_1$

   (b) $\Gamma; (\Pi_1 \otimes r@k\ Q(\overline{t_1})) \vdash e : \exists x.T; R$

   (c) `predicate` $Q(\overline{Tx}) \equiv R_1 \in C$

   (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 the (PACK-2) case of the Memory Consistency lemma

6. q.e.d. -by 4, 5

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

To prove: If $\Gamma, \Pi \vdash \mathit{pack}\ r@1\ Q_2(\overline{t_2})\ \mathit{in}\ e : \exists x.T; R$ and $\mu, \Sigma, F(\Pi), \rho\ \underline{ok}$ and $\mu, \rho, \mathit{pack}\ r\ \mathit{to}\ R_1\ \mathit{in}\ e \rightarrow \mu, \rho, e$ and $\vdash \mathit{Prog}$ then there exists Π', Γ' and Σ' such that $\Gamma', \Pi' \vdash e : \exists x.T; R$ and $\mu, \Sigma', F(\Pi'), \rho\ \underline{ok}$.

1. By assumption

   (a) $\Gamma, \Pi \vdash \mathit{pack}\ r@1\ Q_2(\overline{t_2})\ \mathit{in}\ e : \exists x.T; R$

   (b) $\mu, \Sigma, F(\Pi), \rho\ \underline{ok}$

   (c) $\mu, \rho, \mathit{pack}\ r@1\ Q_2(\overline{t_2})\ \mathit{in}\ e \rightarrow \mu, \rho, e$

2. By inversion on (PACK1)

   (a) $\Gamma; \Pi \vdash r : C; [\overline{t_2}/\overline{x}]R_2 \otimes \textsf{unpacked}(r@1\ Q_2(\overline{t_2})) \otimes \Pi_1$

   (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 the (PACK-1) case of the Memory Consistency lemma

6. q.e.d. -by 4, 5

Case (UNPACK)

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

To prove: If $\Gamma, \Pi \vdash \mathit{unpack}\ r@k\ Q(\overline{t_1})\ \mathit{in}\ e : \exists x.T; R$ and $\mu, \Sigma, F(\Pi), \rho\ \underline{ok}$ and $\mu, \rho, \mathit{unpack}\ r\ \mathit{from}\ R_1\ \mathit{in}\ e \rightarrow \mu, \rho, e$ and $\vdash \mathit{Prog}$ then there exists Π', Γ' and Σ' such that $\Gamma', \Pi' \vdash e : \exists x.T; R$ and $\mu, \Sigma', F(\Pi'), \rho\ \underline{ok}$.

1. By assumption

   (a) $\Gamma, \Pi \vdash \mathit{unpack}\ r@k\ Q(\overline{t_1})\ \mathit{in}\ e : \exists x.T; R$

(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) $\Gamma; (\Pi_1 \otimes [\overline{t_1}/\overline{x}]R_1 \otimes$
   $unpacked(r@k\ Q(\overline{t_1}))) \vdash e : \exists x.T; R$

   (c) $\forall r', \bar{t} : (\ unpacked(r'@k'\ Q(\bar{t})) \in \Pi \Rightarrow \Pi \vdash r \neq r')$, meaning that there is no other unpacked predicate Q for r or any alias of r

   (d) `predicate` $Q(\overline{x}) \equiv R_1 \in C$

   (e) $0 < k < 1$

3. Let $F(\Pi') = (F(\Pi_1 \otimes [\overline{t_1}/\overline{x}]R_1)),$
   $r \rightsquigarrow \mathsf{unpacked}(r@k\ Q(\overline{t_1}))),$
   $\Sigma' = (\Sigma_1, r \rightsquigarrow unpacked), \Gamma' = \Gamma$, where Σ_1 corresponds to Π_1

4. $\Gamma', \Pi' \vdash e : \exists x.T; R$ -by 2b, 3

5. $\mu, (\Sigma_1, r \rightsquigarrow unpacked), (F(\Pi_1 \otimes [\overline{t_1}/\overline{x}]R_1)),$
   $r \rightsquigarrow \mathsf{unpacked}(r@k\ Q(\overline{t_1}))), \rho\ \underline{ok}$ -by the case (UNPACK-2) of the Memory Consistency lemma

6. q.e.d. -by 4, 5

Subcase: the static semantics rule corresponding to (UNPACK) is (UNPACK1).
To prove: If $\Gamma, \Pi \vdash unpack\ r@1\ Q(\overline{t_1})\ in\ e : \exists x.T; R$ and $\mu, \Sigma, F(\Pi), \rho\ \underline{ok}$ and $\mu, \rho, unpack\ r\ from\ R_1\ in\ e\ \rightarrow\ \mu, \rho, e$ and $\vdash\ Prog$ then there exists Π' , Γ' and Σ' such that $\Gamma', \Pi' \vdash e : \exists x.T; R$ and $\mu, \Sigma', F(\Pi'), \rho\ \underline{ok}$.

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) $\Gamma; (\Pi_1 \otimes [\overline{t_1}/\overline{x}]R_1 \otimes unpacked(r@1\ Q(\overline{t_1}))) \vdash e : \exists x.T; R$

   (c) `predicate` $Q(\overline{x}) \equiv R_1 \in C$

3. Let $F(\Pi') = F(\Pi_1 \otimes [\overline{t_1}/\overline{x}]R_1 \otimes unpacked(r@1\ Q(\overline{t_1}))),$
   $\Sigma' = (\Sigma_1, r \rightsquigarrow unpacked), \Gamma' = \Gamma$

4. $\Gamma', \Pi' \vdash e : \exists x.T; R$ -by 2b, 3

5. $\mu, (\Sigma_1, r \rightsquigarrow unpacked), F(\Pi_1 \otimes [\overline{t_1}/\overline{x}]R_1 \otimes unpacked(r@1\ Q(\overline{t_1}))), \rho\ \underline{ok}$ -by the (UNPACK-1) case of the Memory Consistency lemma

6. q.e.d. -by 4, 5

Case (IF-TRUE)

To prove: If $\Gamma, \Pi \vdash if(true)\{e_1\}else\{e_2\} : \exists x.T; R$ and $\mu, \Sigma, F(\Pi), \rho$ $\underline{ok}$ and $\mu, \rho, if(true, e_1, e_2) \rightarrow \mu, \rho, e_1$ and $\vdash$ $Prog$ then there exists Π' , Γ' and Σ' such that $\Gamma', \Pi' \vdash e_1 : \exists x.T; R$ and $\mu, \Sigma', F(\Pi'), \rho$ $\underline{ok}$.

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 holds or if R_2 holds
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)

To prove: If $\Gamma, \Pi \vdash if(false)\{e_1\}else\{e_2\} : \exists x.T; R$ and $\mu, \Sigma, F(\Pi), \rho$ $\underline{ok}$ and $\mu, \rho, if(false, e_1, e_2) \rightarrow \mu, \rho, e_2$ and $\vdash$ $Prog$ then there exists Π' , Γ' and Σ' such that $\Gamma', \Pi' \vdash e_2 : \exists x.T; R$ and $\mu, \Sigma', F(\Pi'), \rho$ $\underline{ok}$.

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 holds or if R_2 holds
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)

To prove: If $\Gamma, \Pi \vdash l.f_i : \exists x.T_i; R$ and $\mu, \Sigma, F(\Pi), \rho$ $\underline{ok}$ and $\mu, \rho, l.f_i \rightarrow \mu, \rho, o_i$ and $\vdash Prog$ then there exists Π' , Γ' and Σ' such that $\Gamma', \Pi' \vdash o_i : \exists x.T_i; R$ and $\mu, \Sigma', F(\Pi'), \rho$ $\underline{ok}$.

1. By assumption
   (a) $\Gamma, \Pi \vdash l.f_i : \exists x.T_i; R$
   (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) $fields(C) = \overline{Tf}$

2. By inversion on the static semantics rule (FIELD)

(a) $l.f_i : T_i \; is \; a \; field \; of \; C$

(b) $\Gamma; \Pi \vdash [o_i/x]R$

(c) $l.f_i \rightarrow o_i$

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 the (TERM) proof rule

6. $\mu, \Sigma' = (\Sigma, o_i \rightsquigarrow Q), F(\Pi') = (F(\Pi), o_i \rightsquigarrow R)), \rho \; \underline{ok}$ -by the (FIELD) case of the Memory Consistency lemma

7. q.e.d. -by 5,6

Case (ASSIGN)

To prove: If $\Gamma, \Pi \vdash (l_1.f = l_2) : \exists x.T; R$ and $\mu, \Sigma, F(\Pi), \rho \; \underline{ok}$ and $\mu, \rho, l_1.f = l_2 \rightarrow \mu[\rho(l_1) \rightsquigarrow [\rho(l_2)/o_i]C(\overline{o})], \rho, o_i$ and $\vdash Prog$ then there exists Π' , Γ' and Σ' such that $\Gamma', \Pi' \vdash o_i : \exists x.T; R$ and $\mu[\rho(l_1) \rightsquigarrow [\rho(l_2)/o_i]C(\overline{o})], \Sigma', F(\Pi'), \rho \; \underline{ok}$.

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 1a using the static semantics rule (ASSIGN)

(a) $\Gamma; \Pi \vdash l_2 : T_i; l_2@k_0 \; Q_0(\overline{t_0}) \otimes \Pi_1$, where T is equal to T_i

(b) $\Gamma; \Pi_1 \vdash l_1.f : T_i; r_1@k' \; Q'(\overline{t'}) \otimes \Pi_2$

(c) $\Pi_2 \vdash l_1.f \rightarrow r_1 \otimes \Pi_3$

(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 $\rho(l_2) = o_2$, by Object Proposition Consistency from Figure 3.8

4. Let $\Gamma' = (\Gamma, o_2 : T_i), F(\Pi') = (F(\Pi \otimes l_2@k_0 \; Q_0(\overline{t_0})), o_2 \rightsquigarrow o_2@k' \; Q'(\overline{t'})), \Sigma' = (\Sigma, o_2 \rightsquigarrow Q'(\overline{t'}))$.

5. $\Gamma', \Pi' \vdash o_2 : \exists x.T_i; R$ -by the (TERM) static semantics rule

6. $\mu' = \mu[\rho(l_1) \rightsquigarrow [\rho(l_2)/o_i]C(\overline{o})], \Sigma', F(\Pi'), \rho \; \underline{ok}$ -by the ASSIGN case of the Memory Consistency lemma

7. q.e.d. -by 5, 6

Case (INVOKE)

To prove: If $\Gamma, \Pi \vdash l_1.m(\overline{l_2}) : \exists x.T_r; R'$ and $\mu, \Sigma, F(\Pi), \rho$ $\underline{ok}$ and $\mu, \rho, l_1.m(\overline{l_2}) \rightarrow \mu, \rho, [l_1/this, \overline{l_2}/\overline{x}]e$ and $\vdash Prog$ then there exists Π' , Γ' and Σ' such that $\Gamma', \Pi' \vdash [l_1/this, \overline{l_2}/\overline{x}]e : \exists x.T_r; R'$ and $\mu, \Sigma', F(\Pi'), \rho$ $\underline{ok}$.

(a) By assumption
  i. $\Gamma, \Pi \vdash l_1.m(\overline{l_2}) : \exists x.T_r; R'$
  ii. $\mu, \Sigma, F(\Pi), \rho$ $\underline{ok}$
  iii. $\vdash Prog$
  iv. $\mu, \rho, l_1.m(\overline{l_2}) \rightarrow \mu, \rho, [l_1/this, \overline{l_2}/\overline{x}]e$
  v. $\mu(\rho(l_1)) = C(\overline{o})$, meaning that l_1 is an indirect reference to an object of class C which has $o_1, o_2, ...$ as fields
  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. $mtype(m, C) = \forall \overline{x : T}.\exists result.T_r; R_1 \multimap R$
  iv. $\exists x.T; R' = \exists$ *result*.$T_r; [l_1/this][\overline{l_2}/\overline{x}]R \otimes \Pi_1$, where Π_1 does not contain unpacked predicates

(c) From 7(a)iii 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 and Framing Lemma, we know that $[l_1/this, \overline{l_2}/\overline{x}]e$ will be of the type $\exists x.T_r; R'$. By the (INVOKE) case of the Memory Consistency lemma we know that $\mu, \Sigma', F(\Pi'), \rho$ $\underline{ok}$, where $F(\Pi')$ is equal to $F([l_1/this][\overline{l_2}/\overline{x}]R \otimes \Pi_1)$ and Σ' corresponds to $F(\Pi')$.

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

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.

Chapter 4

Implementation

We wanted to prove that the object proposition methodology was not only modular and sound, but also automatable. We have thus implemented the many aspects of our theory into our Oprop tool, accessible online at *lowcost-env.ynzf2j4byc.us-west-2.elasticbeanstalk.com*, which can be used to statically verify a variety of programs annotated with the Oprop annotations.

In this chapter we first discuss the possible sources of nondeterminism for the object proposition inference algorithm and how we have dealt with each of them. Nondeterminism is unwanted because it can lead to different behaviors of the inference algorithm for different runs on the same program that we want to verify. The inference algorithm does not infer specifications for methods of a program; instead users (programmers) write the pre- and post-conditions before the body of methods, and they write the pack and unpack annotations at various points inside the body of methods. All these specifications are written by users before passing the program to Oprop to be verified. For the inference algorithm to run without asking users to write annotations between every two statements we must propagate object propositions throughout the program, and if this is not done carefully nondeterminism can arise.

Then we introduce the Oprop tool and the formal rules of translation from the Oprop language into the Boogie intermediate verification language. We discuss the most interesting rules and the peculiarities of their implementation in the tool. Our presentation of the rules and our discussion are accompanied by simple examples, meant to facilitate the reader's understanding. We go on to present some alternative approaches of logical encodings, created by other researchers. We also present a comparison between our Oprop tool and the JavaSyp [7] tool and explain why we found it necessary to implement Oprop independently, using different technologies. In subsection 4.5 we present the theorem that expresses the fact that our Boogie translation and the original Oprop program are semantically equivalent. We conclude this chapter with an informal soundness argument.

4.1 Object Proposition Inference Algorithm

The proof rules presented in the section 3.3 are deterministic because at each point in the program there is a unique rule that can be applied. The proof rules were not written in this deterministic form from the beginning. For example, the rule PACK2 was first written as below:

$$\frac{\begin{array}{c}\Gamma; \Pi \vdash r : C; [\overline{t_1}/\overline{x}]R_1 \otimes \mathsf{unpacked}(r@k\ Q(\overline{t_1})) \\ \texttt{predicate}\ Q(\overline{Tx}) \equiv R_1 \in C \quad 0 < k < 1 \\ \Gamma; (\Pi' \otimes r@k\ Q(\overline{t_1})) \vdash e : \exists x.T; R\end{array}}{\Gamma; \Pi \otimes \Pi' \vdash \texttt{pack}\ r@k\ Q(\overline{t_1})\ \texttt{in}\ e : \exists x.T; R}\ \textsc{Pack2}$$

The above rule is not deterministic because there is a context split that has to be guessed. The tool Oprop would have to guess which part of the linear context in the conclusion is Π and which part is Π'. We have rewritten the PACK2 rule to make it deterministic and to eliminate the need to split contexts. The current form of the rule is:

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

In order to manage context splits, we have used resource management techniques for linear proof search [27].These techniques add an additional output, the "leftover" resources, to judgments. The idea of resource management is that we can make context splits deterministic by sending all resources for proving the first premise, then use the leftover resources to prove the second premise, then use the leftover resources from the second premise to prove the third premise, and so on.

Another source of non-determinism is that object propositions can be split an arbitrary number of times and merged back together. The formal system "guesses" exactly how a given permission inside of an object proposition has to be split up and merged in order to satisfy different object propositions (for example in order to satisfy a pre-condition). For example, a fractional permission of 1 can be split into two fractions $\frac{1}{4}$ and $\frac{3}{4}$, or into two fractions of $\frac{1}{2}$. In this case, we would have the following constraints C for the fractions k_1 and k_2: $k_1 + k_2 = 1$, $k_1, k_2 \in (0, 1)$. We track constraints C together with the current linear context Π. The constraints accumulated during the verification procedure allow us to know everything about the permissions needed inside a piece of code. In order to prove that a program is correct, we will have to prove that the constraints are satisfiable. As discussed in [18], Fourier-Motzkin elimination can be used to check the satisfiability of the fractional constraints.

One final source of non-determinism comes from situations where multiple proof rules of our linear theory are applicable. For example, for proving the choice $P_1 \oplus P_2$ there are two proof rules that can be applied:

$$\frac{\Gamma; \Pi \vdash P_1}{\Gamma; \Pi \vdash P_1 \oplus P_2} \oplus_L \quad \frac{\Gamma; \Pi \vdash P_2}{\Gamma; \Pi \vdash P_1 \oplus P_2} \oplus_R$$

There are two options when multiple proof rules can be applied: we can either use backtracking or we can use forward reasoning. Backtracking would mean that we arbitrarily choose one of the applicable rules and if it turns out that the chosen rule does not work, we roll back and try the other one.

In our setting, backtracking is not the best solution. We are trying to verify an entire program. For doing this, we have to prove linear theory predicates for every method call and object construction. Whenever such a proof does not succeed, we would have to backtrack to the last call or construction site in the program where we made a choice. Similarly to [18], our tool implements a dataflow analysis. Backtracking to the last call or construction site as part of a dataflow analysis is difficult because we would have to roll back the entire state of the flow analysis. Due to these reasons, we choose to carry all possible choices forward, knowing that we pay the cost in the time of verification.

4.2 The Oprop Tool

We implemented the object proposition system in our Oprop tool, as a static dataflow analysis for the Oprop language. The purpose of the tool is to show that the object proposition methodology is automatable and can be used to automatically verify both simple programs and complex ones such as the composite instance. In the remainder of this section we discuss practical details of the implementation. The Oprop tool is able to verify the correctness of single-threaded programs. The extension of Oprop into a tool that can verify multi-threaded programs is left as future work.

In order to specify method pre- and post-conditions using object propositions, developers need to use annotations that can express linear theory concepts and fractional permissions. Annotations are the main way in which developers interact with Oprop.

The Oprop tool first translates the code and specifications into the intermediary language Boogie, which is also used as an intermediary language by Dafny and Chalice of the RiSE group at Microsoft Research [16]. We then use that translation as the input to the Boogie tool, which sends it to the Z3 theorem prover [29] to obtain the final verification result.

The Boogie verifier tool uses Z3 as its backend to prove properties about integers. The Z3 theorem prover is perfect for us, since our abstract predicates use integers to express the properties of fields. Reasoning about integers in an automated way can be difficult, but Z3 is one of the most prominent satisfiability modulo theories (SMT) solvers and it is used in related tools such as Boogie [16], Dafny [48], Chalice [47] and VCC [28]. We counteract the problems that Z3 has in finding the witnesses for the existentially quantified variables by having the programmer specify those witnesses.

4.3 Encoding Our Linear Theory into First Order Logic with Maps

In order to use Z3 for the verification of the generated conditions, we need to encode our extended fragment of linear logic, i.e., our linear theory, into Boogie, which is based on first order logic and uses maps as first class citizens of the language. By 'extended fragment of linear logic' we mean the fragment of linear logic containing the operators $\otimes$ and $\oplus$, that we extend with the specifics of our object propositions methodology. Specifically, we need to encode R described in the grammar in Section 3.1. The crux of the encoding is in how we treat the fractions of the object propositions, how we keep track of them and how we assert statements about them. For object propositions, we encode whether they are packed or unpacked, the amount of the fraction that they have and the values of their parameters. Fractions are intrinsically related to keeping track of resources, the principal idea of linear logic. The challenge was to capture all the properties of the Oprop language and soundly translate them into first order logic statements. We were able to use the *map* data structure that the Boogie language provides to model the heap and the fields of each class. The maps were also helpful for keeping track of the fractions associated with each object, as well as for knowing which object propositions were packed and which were unpacked at all points in the code.

4.3.1 The Formal Translation Rules

The example `SimpleCell.java` in Figure 4.1 shows an Oprop class. The language used, Oprop, is a simplified version of Java augmented with the object propositions annotations. This example differs from our pure theory in a number of ways: every Oprop input file has to have the declaration of the enclosing package as first statement, each object proposition uses the symbol `#` instead of `@` because the symbol `@` is already used in Java for writing annotations in the code, the linear conjunction and disjunction that we use in our formal grammar in Figure 3.1 are replaced by `&&` and `||` in the Oprop code.

When the object `c` is created in the `main()` method, we have to specify the predicate that holds for it in case the object becomes shared in the future. Since the predicate `PredVal` has one existentially quantified variable and the Boogie tool cannot instantiate existential variables, we give the witness `2` for the variable `int v` existentially quantified in the body of the predicate `PredVal`. In general, whenever there is an existential Oprop statement in the code, we pass the witnesses for that statement explicitly.

When we unpack a predicate we check that the provided witness is the right one; we do not assume that the programmer provided the right witness. We implemented the translation strategy in this way because the programmer might make a mistake and provide the wrong witness. There are other tools like the PVS verification system [58] where the programmer provides the witnesses and the tool verifies that they are correct. In the *PVS Language Reference* [59] at page 39 it is stated that a proof obligation is generated with the provided witness in place of the existential variable, which is then verified. We also explicitly verify that the witness is correct when we pack a predicate thus making our translation strategy symmetrical for packing and unpacking.

```
package x;

class SimpleCell {
 int val;
 SimpleCell next;

 predicate PredVal() = exists int v : this.val -> v && v<15

 predicate PredNext() = exists SimpleCell obj :
  this.next -> obj && (obj#0.34 PredVal())

 void changeVal(int r)
 ~double k : requires (this#k PredVal()) && (r<15)
 ensures this#k PredVal()
 {
  unpack(this#k PredVal())[this.val];
  this.val = r;
  pack(this#k PredVal())[r];
 }

 void main() {
  SimpleCell c = new SimpleCell(PredVal()[2])(2, null);
  SimpleCell a = new SimpleCell(PredNext()[c])(2, c);
  SimpleCell b = new SimpleCell(PredNext()[c])(3, c);

  unpack(a#1 PredNext())[c];
  unpack(b#1 PredNext())[c];
  c.changeVal(4);
 }
}
```

Figure 4.1: SimpleCell.java

```
type Ref;
const null: Ref;
var val: [Ref]int;
var next: [Ref]Ref;
var packedPredNext: [Ref] bool;
var fracPredNext: [Ref] real;
var packedPredVal: [Ref] bool;
var fracPredVal: [Ref] real;

procedure PackPredNext(obj: Ref, this:Ref);
  requires (packedPredNext[this]==false) &&
    (((fracPredVal[next[this]] >= 0.34))) && (next[this]==obj);
procedure UnpackPredNext(obj: Ref, this:Ref);
  requires packedPredNext[this] &&
    (fracPredNext[this] > 0.0);
  requires (next[this]==obj);
  ensures (((fracPredVal[next[this]] >= 0.34))) && (next[this]==obj);

procedure PackPredVal(v:int, this:Ref);
  requires (packedPredVal[this]==false) &&
    ((v<15)) && (val[this]==v);
procedure UnpackPredVal(v:int, this:Ref);
  requires packedPredVal[this] &&
    (fracPredVal[this] > 0.0);
  requires (val[this]==v);
  ensures ((v<15)) && (val[this]==v);

procedure SimpleCell(v:int, n:Ref, this:Ref)
  modifies next,val;
  ensures ((val[this]==v)&&(next[this]==n));
  ensures (forall x: Ref::((x!=this)==>(next[x]==old(next[x]))));
  ensures (forall x: Ref::((x!=this)==>(val[x]==old(val[x]))));
{ val[this]:=v;
  next[this]:=n; }

procedure changeVal(r:int, this:Ref)
  modifies packedPredVal,val;
  requires (this != null) && (((packedPredVal[this] ) &&
    (fracPredVal[this] > 0.0))&&(r<15));
  ensures ((packedPredVal[this] ) &&
    (fracPredVal[this] > 0.0));
  requires (forall x:Ref :: packedPredVal[x]);
  ensures (forall x:Ref :: packedPredVal[x]);
  ensures (forall x:Ref :: (fracPredVal[x]==old(fracPredVal[x])));
```

Figure 4.2: simplecell.bpl

```
{
  assume (forall y:Ref :: (fracPredVal[y] >= 0.0) );
  call UnpackPredVal(val[this], this);
  packedPredVal[this] := false;
  val[this]:=r;
  call PackPredVal(r, this);
  packedPredVal[this] := true;
}

procedure main(this:Ref)
  modifies fracPredNext,fracPredVal,next,
    packedPredNext,packedPredVal,val;
  requires (forall x:Ref :: packedPredNext[x]);
  requires (forall x:Ref :: packedPredVal[x]);
{
  var c:Ref;
  var a:Ref;
  var b:Ref;
  assume (c!=a) && (c!=b) && (a!=b) ;
  assume (forall y:Ref :: (fracPredNext[y] >= 0.0) );
  call SimpleCell(2,null,c);
  packedPredVal[c] := false;
  call PackPredVal(2,c);
  packedPredVal[c] := true;
  fracPredVal[c] := 1.0;
  call SimpleCell(2,c,a);
  packedPredNext[a] := false;
  call PackPredNext(c,a);
  fracPredVal[c] := fracPredVal[c] - 0.34;
  packedPredNext[a] := true;
  fracPredNext[a] := 1.0;
  call SimpleCell(3,c,b);
  packedPredNext[b] := false;
  call PackPredNext(c,b);
  fracPredVal[c] := fracPredVal[c] - 0.34;
  packedPredNext[b] := true;
  fracPredNext[b] := 1.0;
  call UnpackPredNext(c, a);
  fracPredVal[c] := fracPredVal[c] + 0.34;
  packedPredNext[a] := false;
  call UnpackPredNext(c, b);
  fracPredVal[c] := fracPredVal[c] + 0.34;
  packedPredNext[b] := false;
  call changeVal(4,c);
}
```

Figure 4.3: simplecell.bpl - cont.

The Oprop class in the file *SimpleCell.java* is a point of reference through the presentation of the translation rules. Even though the class is written in the Oprop language, the extension of the file remains *.java*. The translation into the Boogie language is given in Figures 4.2 and 4.3.

Translation of fractions

This is a general description of the way we treat fractions in the translation of Oprop into Boogie. For each predicate `Pred` we have a map `fracPred` declared as follows

`var fracPred : [Ref] real`. You can see two such maps on lines 6 and 8 of Figure 4.2 representing the fraction maps for the predicates `PredNext` and `PredVal`

For each object `obj`, this map points to the value of type real of the fraction `k` that is in the object proposition `obj@k Pred(`$\bar{t}$`)`. The map `fracPred` represents all the permissions on the stack. Since `fracPred` is a global variable it always contains the values of the fractions for all objects. An important distinctions is that in a method we only reason about the values stored in `fracPred` for locally accessible objects (objects that are mentioned in the precondition of that method). As we go through the body of that method, the value of `fracPred` for the objects that are touched in any way changes; these are objects that we get out of a parent object while unpacking, or objects that we put into a parent when packing, or objects that we pass to a method call. Say `fracPred` for object `obj` is specified in the precondition of a method to have a certain value `val`. If in the body of that method we unpack an object proposition recursively containing `obj@k Pred(`$\bar{t}$`)`, we add `k` to `fracPred[obj]`. On the other hand, if in the body of that method we pack an object proposition recursively containing `obj@k Pred(`$\bar{t}$`)`, we subtract `k` from `fracPred[obj]`. When we call another method inside the body of a method, we do not subtract or add the fractions appearing in the pre- or postcondition of the called method because if `fracPred` was modified in the called method, `fracPred` will be part of the global variables mentioned in the `modifies` clause of that method and the modifications will appear in the postcondition of the called method. If `fracPred` is indeed modified in the called method but it does not appear in the postcondition, we assume that the programmer does not know exactly what happens to that fraction and hence we add `fracPred[obj] := 0.0` after the called method, for soundness reasons.

Translation of our linear theory into first order logic

The translation of our linear theory (LL) into first order logic (FOL) is given in the following paragraphs of this subsection, where we present the rules of translation of our Oprop language into the Boogie intermediate verification language. Each non-terminal from the grammar in the previous section has a corresponding translation rule. For each rule we point to the lines where it appears in the result of the translation in Figures 4.2 and 4.3. If a verification involves multiple Oprop files, they need to be written in the same Boogie .bpl file. The Oprop tool at *lowcost-env.ynzf2j4byc.us-west-2.elasticbeanstalk.com* does this concatenation of the translated files automatically.

```
function extractPredicates(ClDecl[]) returns String {
    String result = "";
    for each (predName in predicates(ClDecl[])) {
        result += "var packed" + predName + ": [Ref] bool;";
        result += "var frac" + predName + ": [Ref] real;";
        for each (paramName:paramType of parameters(predName) that
          do not correspond to a field) {
            result += "var param" + predName + paramName + ": [Ref] "
               + paramType+";";
        }
    }
    return result;
}
```

Figure 4.4: extractPredicates(ClDecl[]) translation helper function

Object references

At the start of each Boogie program we declare the type `Ref` that represents object references, as can be seen on line 1 in the `SimpleCell` example. We declare a map from a reference `r` to a real representing the fraction `k` for each object proposition `r@k Q(`$\bar{t}$`)`. We declare a second map from a reference `r` to a boolean, keeping track of which objects are packed. Each key points to `true` if and only if the corresponding object proposition is packed for that object. For each predicate `Q`, we have a map keeping track of fractions and a map keeping track of the packed objects. The result of these translation rules is shown on lines 5 to 8 in the `SimpleCell` example. We have defined using pseudocode the helper translation function extractPredicates(ClDecl[]) in Figure 4.4 that looks inside all the input classes, given as a parameter array, and extracts the predicate names for the purpose of declaring the maps `packed` and `frac` for each predicate. Figure 4.5 presents a slightly more detailed version of the same helper function. The function extractPredicates(ClDecl[]) also generates maps for each predicate parameter that does not correspond to any field (such a parameter could be a bound value an a field).

It is important to generate all these maps in the beginning of the translation because predicates can be recursive and can refer to other predicates (and thus they might access the maps corresponding to those predicates). If not all the global maps are declared in the beggining of the `.bpl` file, we might encounter the situation where such a map is accessed before having been declared.

trans(Prog) ::= `type Ref;`
`const null : Ref;`
extractPredicates(ClDecl[])
trans($\overline{\text{ClDecl}}$) trans(e)

Class declarations

A class declaration is made of the field, predicate, constructor and method declarations.

```
function extractPredicatesLongVersion(ClDecl[]) returns String {
    String result = "";
    set<String> predicateSet;
    for each c:Class from ClDecl[] {
        for each p:Predicate {
            add p.name to predicateSet;
        }
    }
    for each (pName in PredicateSet) {
        result += "var packed" + pName + ": [Ref] bool;";
        result += "var frac" + pName + ": [Ref] real;";
        for each (paramName:paramType of parameters(pName) that
          do not correspond to a field) {
            result += "var param" + pName + paramName + ": [Ref] " +
               paramType+";";
        }
    }
    return result;
}
```

Figure 4.5: extractPredicatesLongVersion(ClDecl[]) translation helper function

trans(ClDecl) ::= $\overline{\text{trans(FldDecl)}}$ $\overline{\text{trans(PredDecl)}}$
$\overline{\text{trans(ConstructorDecl)}}$ $\overline{\text{trans(MthDecl)}}$

Each field is represented by a map from object references to values, representing the value of that field of that object. You can see the maps declared for the fields `val` and `next` on lines 3 and 4.

trans(FldDecl) ::= `var f: [Ref]`trans(T);

Declarations of abstract predicates

The declaration of an abstract predicate has several steps. The first procedure is used for packing the predicate `Q`, while the second one is used for unpacking it.

trans(PredDecl) ::=

`procedure PackQ(`$\overline{\text{x : trans(T)}}$, extractExistentials(Q), `this:Ref`);
`requires (packedQ[this] == false);`
`requires` removePacked(trans(R));
`procedure UnpackQ(`$\overline{\text{x : trans(T)}}$, extractExistentials(Q), `this:Ref`);
`requires packedQ[this];`
`ensures` removePacked(trans(R));

The procedure `PackQ` is called in the code whenever we have to pack an object proposition, according to the *pack(...)* annotations that the programmer inserted in the code. Right after calling the *PackQ* procedure, we write `packedQ[this] := true;` in the Boogie code. We do this right after calling `PackQ`, instead of in the `PackQ` postcondition, because only the for-

```
function extractExistentialsLongVersion(String predicate1)
returns SetOfVariableNameAndType {
    SetOfVariableNameAndType setPairs;
    for each (existentialVariable v in predicate1) {
         add v to setPairs;
    }
    return setPairs;
}
```

Figure 4.6: extractExistentialsLongVersion() translation helper function

```
function extractExistentials(String predicate1)
returns SetOfVariableNameAndType {
    return {v:T | "exists v:T" appears in predicate1};
}
```

Figure 4.7: extractExistentials() translation helper function

```
function removePacked(String transl)
returns transl from which all mentions
        of packedQ[] or (packedQ[] == false) are removed,
 for any Q;
}
```

Figure 4.8: removePacked() translation helper function

mer alternative actually changes the value in the `packedQ` map. We tried to add `ensures packedQ[this]` to the postconditions of the `PackQ` procedure and it has lead to contradictions in the resulting Boogie proof. This is the reason why `packedQ[this]` is not in the post-condition for the procedure `PackQ`. After calling the *PackQ* procedure, we also write the statements that manipulate the fractions that appear in the body of the predicate that we are packing. When packing a predicate, we subtract from the current value of fractions.

Whenever there is a packed object proposition in the body of a predicate, for example in the body of the predicate `P` we have the packed object proposition `r1@k Q()`, we model it in the following way: in the procedures `UnpackP` and `PackP` we have `requires (fracQ[r1] > 0.0)` and `ensures (fracQ[r1] > 0.0)` respectively. We do not have `requires packedQ[r1]` and `ensures packedQ[r1]` respectively, in the body of a predicate. The intuition here is that we assume the object propositions appearing in the initial definition of a predicate are unpacked. For the pre- and pos-conditions of procedures our methodology guarantees that if an object proposition is unpacked, it will appear in the specifications as unpacked. The reasoning is that in the case of a predicate we have a pointer to an object proposition, that might be packed or not. It could be that this object proposition is unpacked at the moment, but will be packed at a later time. For all procedures where we might have unpacked object propositions in the pre-conditions of that procedure, we add a statement of the form `requires (forall y:Ref :: (y!=obj1) ==> packedQ[y])`. This statement states that all the object propositions that are not explicitly mentioned to be unpacked will be considered packed. For the `requires forall` statement just mentioned, assume that the object `obj1` has been explicitly mentioned in an unpacked object proposition. The upside is that we can rely on such `ensures forall` statements but we also need to prove them as post-conditions of procedures.

In Figure 4.7 we have written using pseudocode the helper translation function extractExistentials( ). This pseudocode is similar to the function that we wrote in the translation code in our Oprop tool and it extracts all the existentially quantified variables in the predicate `Q` and adds them to the list of parameters of both the `PackQ` and `UnpackQ` procedures. After adding the existential parameters we also add the parameter `this:Ref` to the list of parameters, representing the callee of the respective procedure. This callee will be replaced with the actual callee in the body where the `PackQ` or `UnpackQ` procedures are called.

We have observed that it is very difficult for the Boogie tool to prove something of the form `∃ x:int :: formula(x)`. The Boogie tool does not know and cannot find out which is the right instantiation value, even if the formula is very simple. Because of this, the programmer has to specify in the Java code the right instantiation value whenever there is a variable that is existentially quantified in the Oprop annotations. In the case of the `SimpleCell.java` example you can see the explicit mention of the existentially quantified variables on lines 22-24 and 26-27 of Figure 4.1 when the constructor is called and when the unpacking happens, respectively. You can see the procedures `PackPredNext` and `PackPredVal` for predicates `PredNext` and `PredVal` on lines 10 to 12 and 19 to 21 respectively.

Similarly, the procedure `UnpackQ` is called in the code whenever we need to unpack an object proposition, whenever we need to access the field of an object or we need to add together fractions in order to get the right permission (usually when we need a permission of 1 in order to modify a predicate). The procedure `UnpackQ` is inserted in the code whenever the programmer inserted the *unpack(...)* annotation in the Java code. Right after calling the procedure `UnpackQ`, we write `packedQ[this] := false;` in the code. We also write the statements that manipulate by addition the fractions that appear in the body of the predicate that we are packing. You can see the procedures `UnpackPredVal` and `UnpackPredNext` for predicates `PredVal` and `PredNext` on lines 22 to 26 and 13 to 17. In the `ensures` statement of the predicate `PredNext` we did not write `fracPredNext[this] >= 0.34`. This not needed because we are going to add this fraction in the caller, right after calling `UnpackPredNext`.

For each class we write a constructor. For the class `SimpleCell` the translation of the constructor is on lines 28 to 34.

trans(ConstructorDecl) ::= `procedure ConstructMyClass(`$\overline{x}$`,this:Ref);`
`ensures` $\overline{(f[this] == x)}$

After calling the constructor in the code we add

`packedQ[this] := true` && `fracQ[this] := 1.0`

assuming the object that is being constructed is packed to the predicate `Q`. The user specifies to our tool which predicate they intend to pack to by adding the name of that predicate and the parameters to that predicate, if there are any, in the call to the constructor. This can be seen on lines 22-24 in Figure 4.1 right after the name of the constructor `SimpleCell`. For our `SimpleCell` example the constructor is called multiple times in our `main` function and you can see one such call and the statements that are written right after the call on lines 70 to 74 in the `.bpl` translation. The user specifies to our Oprop tool which predicate they intend to pack to by writing the name of the predicate, together with any parameters that the predicate needs, in the call to the constructor.

Translation of a method

A method is translated into a `procedure` in Boogie. When specifying a method, we have to specify the variables that it modifies, its precondition and its postcondition. We define the method `changeVal` in the `SimpleCell` class on lines 36 to 57. For each method we have added two kinds of statements that we call `requires forall` and `ensures forall`. We present the way we construct each of these statements in the helper functions in Figure 4.9 and 4.11. In Figures 4.10 and 4.12 we present the longer versions of the same two functions, but now with more technical details and closer to the actual code existant in our Oprop tool.

The `requires forall` statement explicitly states the object propositions that are packed at the beginning of a method, which are almost all object propositions in most cases. Since there are no unpacked object propositions in the preconditions of the method `changeVal`, the `requires forall` states that all the object propositions for the `PredVal` predicate are packed. Each `requires forall` or `ensures forall` statement refers to a single predicate and thus each method might have multiple such statements.

The `modifies` clause of a procedure in Boogie has to state all the global variables that this procedure modifies, through assignment, and all the variables that are being modified by other procedures that are called inside this procedure.

The `modifies` clause that Boogie needs for each procedure is a very insidious clause: it tells Boogie that all the values of a certain field have been modified, for all references. This leads to us not being able to rely on many properties that were true before entering a procedure. We

```
function constructRequiresForall(String methodName1)
returns String {
    String result = "";
    for all predicates P that appear unpacked in
    preconditions(methodName1) {
        result += "requires (forall param ::
            (param != r1 && ... && param != rN) ==> packedP[param])"
        where r1...rN are the references
            corresponding to the unpacked object propositions
    }
    for all predicates P that appear only in packed form in
    preconditions(methodName1) {
        result += "requires (forall param :: packedP[param])"
    }
    return result;
}
```

Figure 4.9: constructRequiresForall() translation helper function

counteract the effect of the `modifies` by adding statements of the form `ensures (forall y:Ref :: (fracP[y] == old(fracP[y])) )` and
`ensures (forall y:Ref :: (packedP[y] == old(packedP[y])) )`, for all `fracP`, `packedP` or global fields maps that were mentioned in the `modifies` clause of the current procedure. Of course, if the value of the maps `fracP`, `packedP`, etc. does change in the method we do not add these `ensures forall` statements. The exact algorithms for inferring the `requires forall` and `ensures forall` statements for the `packedQ` global variables are given in Figures 4.9 and 4.11.

Figure 4.13 gives the intuition of how the set of modified variables for each method is calculated while Figure 4.14 presents a more detailed view. In Figure 4.14 the implementation of the addition of `list2` to `list1` in line `add the modifies list list2 of mc2 to list1` is done with a least fixed point in the case of recursion.

trans(MthSpec) ::= `modifies` getModifiesVariables();
`requires (this!=null) &&` trans(R);
constructRequiresForall(m)
`ensures` trans(R);
constructEnsuresForallPacked(m)

Below we give the translation of a method declaration.

trans(MthDecl) ::= `procedure m(` $\overline{\text{trans(T) x}}$ `) returns (`r:trans(T)`)`
trans(MthSpec)
{ $\overline{\text{assume (forall y:Ref :: (fracQ[y] >= 0.0));}}$
$\overline{\text{trans}(e_1)}$; `var r :=` trans(e_2); `return r;` }

We explicitly assume in the beginning of the body of each method that all fractions that are mentioned in the pre- or post-conditions of that method are larger than 0. We need to explicitly add these assumptions because the Boogie tool does not have any pre-existing assumptions about

```
function constructRequiresForallLongVersion(String methodName1)
returns String {
    String result = "";
    mapPredicateObject mapPredObjUnpacked;
    setPredicateObject setPredObjPacked;
    foreach (objectProposition objProp1
            in preconditions(methodName1) {
        if (objProp1 is unpacked) {
            mapPredObjUnpacked[objProp1.predicate].add(objProp1.object
               );
        }
        if (objProp1 is packed) {
            add (objProp1.predicate, objProp1.object)
            to setPredObjPacked;
        }
    }
    foreach (predicate key1 in mapPredObjUnpacked) {
        result += "requires (forall param :: (" +
                  "(param!=" + mapPredObjUnpacked[key1][0] + ")";
        int size = mapPredObjUnpacked[key1].size();
        foreach (object1 = mapPredObjUnpacked[key1][1]
                 until mapPredObjUnpacked[key1][size -1])
             {
                  result += "&& (param!=" + object1 + ")";
             }
        result += " ) ==>
                  packed" + pair1.predicate + "[param]);";
    }
    foreach (predicateObject pair1 in setPredObjPacked) {
        result += "requires (forall param ::" +
                  "packed" + pair1.predicate + "[param]);";
    }
    return result;
}
```

Figure 4.10: constructRequiresForallLongVersion() translation helper function

```
function constructEnsuresForallPacked(String methodName1)
returns String {
    String result = "";
    for all predicates P that appear in modifiesSet(methodName1)
    and appear in postconditions(methodName1)
    and appear in preconditions(methodName1) {
        if P only appears as packed in postconditions(methodName1) {
            result += "ensures (forall param::packedP[param])";
        } else {
            result += "ensures (forall param ::
                (param != r1 && ... && param != rN)
                ==> packedP[param])";
            where r1...rN are the references corresponding to object
            propositions containing P that appear packed in the
            preconditions but unpacked in the postconditions
        }
    }
    return result;
}
```

Figure 4.11: constructEnsuresForallPacked() translation helper function

our global maps representing fractions.

In the translation of predicates, when fractions are existentially and universally quantified, it means that the value of the map `fracQ` for the appropriate key is strictly greater than zero.

In our notation below we are using the separator '|' to separate the translation for the grammar pieces, in the same order as they are specified in Section 3.1. To help the reader, we reproduce the definition of R from the Oprop grammar:

R ::= P | R ⊗ R | R ⊕ R |
∃x:T.R | ∃z:double.R | ∃z:double.z binop t ⇒ R |
∀x:T.R | ∀z:double.R | ∀z:double.z binop t ⇒ R |
t binop t ⇒ R

The translation of R is:

trans(R) ::= trans(P) | translateAnd(R,R) |
trans(R) || trans(R) |
addExistential(trans(R), t:trans(T)) |
addExistentialFrac(trans(R), k:trans(double)) |
addExistentialFracBinop(trans(R), k:trans(double), `k binop t`) |
`var` t:trans(T);trans(R) |
`var` k:real;trans(R) |
`var` k:real;k trans(binop) t ==>trans(R) |
t trans(binop) t ==> trans(R)

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

```
function constructEnsuresForallPackedLongVersion(String methodName1)
returns String {
    postcond1 = postcondition(methodName1);
    precond1 = precondition(methodName1);
    String result = "";
    foreach (predicate p1 in modifiesSet) {
        if (p1 is mentioned in postcond1
            forall objects obj1 for which
            p1(obj1) is in precond1
        {
            if (all p1(obj1) in postcond1 are packed) {
                result += "ensures (forall param::
                  packed "+p1+"[param])";
            } else {
                let obj2 be the ordered set of objects for which
                packedp1[obj2]==false in postcond1;
                result += "ensures (forall param :: (" +
                    "(param!=" + obj2[0] + ")";
                int size = obj2.size();
                foreach (object1 = obj2[1] until obj2[size -1])
                {
                    result += "&& (param!=" + object1 + ")";
                }
                result += " ) ==>
                  packed" + p1 + "[param]);";
             }
        }
    }
    return result;
}
```

Figure 4.12: constructEnsuresForallPackedLongVersion() translation helper function

```
function getModifiesVariables(String method1)
returns List {
    list = variables lf1 that appear in
        assignment statements lf1:=rf1 in method1;
    add to list all the modifiesVariables of
        methods called in method1;
    return list;
}
```

Figure 4.13: getModifiesVariables() translation helper function

```
function getModifiesVariablesLongVersion(String method1)
returns ListOfVariableNames {
    ListOfVariableNames list1={}
    foreach (statement st1 in method1) {
        if (st1 is of form lf1 := rf1) {
            add lf1 to list1;
        }
    }
    foreach (method call mc2 in method1) {
        add the modifies list list2 of mc2 to list1;
    }
    return list1;
}
```

Figure 4.14: getModifiesVariablesLongVersion() translation helper function

```
function addExistential(trans(R), t:trans(T))
{
    MethodOrPredDeclaration result;
    let methodOrPred(params) be the method
    or predicate in the body of which R is found;
    update methodOrPred(params) to be
        methodOrPred(params, t:trans(T));
    return trans(R);
}
```

Figure 4.15: addExistential() translation helper function

```
function addExistentialFrac(trans(R), k:trans(double))
{
    MethodOrPredDeclaration result;
    let methodOrPred(params) be the method
    or predicate in the body of which R is found;
    update methodOrPred(params) to be methodOrPred(params, k:real);
    return trans(R);
}
```

Figure 4.16: addExistentialFrac() translation helper function

```
function addExistentialFracBinop(trans(R), k:trans(double), k binop t)
{
    MethodOrPredDeclaration result;
    let methodOrPred(params) be the method
    or predicate in the body of which R is found;
    update methodOrPred(params) to be methodOrPred(params, k:real) and
    add "assert k binop t;"(or "k binop t") to body of methodOrPred;
    return trans(R);
}
```

Figure 4.17: addExistentialFracBinop() translation helper function

```
function translateAnd(R1, R2) returns FOLFormula {
  let R = DNF(R1 cross R2)
  let R' = FOL(coalesce(R))
  return transAtoms(R') }
where
  DNF(R) converts linear formula R to disjuctive normal form
  coalesce(R) merges atoms in the same conjunction by adding fractions
  FOL(R) replaces linear connectives with first-order logic
     connectives
  transAtoms(R) translates the atoms of R, leaving connectives
     unchanged
}
```

Figure 4.18: translateAnd() translation helper function

of one or more multiplicative conjunctions of one or more of the predicates
P, $\exists$x:T.R, $\exists$z:double.R, $\exists$z:double.z binop t $\Rightarrow$ R, $\forall$x:T.R, $\forall$z:double.R,
$\forall$z:double.z binop t $\Rightarrow$ R and t binop t $\Rightarrow$ R. In practice if a formula contains quantifiers, we first move the quantifiers on the outside of the formula and then convert it to disjunctive normal form. We need the formula to be in fisjunctive normal form when we apply our translation strategy from Oprop into Boogie, described in Chapter 4.

Let us examine a predicate: let us call it `OK` and assume it has an existential statement in its definition. If the `OK` predicate was originally written "$\exists$ v,d: val -> v $\otimes$ dbl -> d $\otimes$ d == v*2", in the Boogie translation we eliminate the existential quantifier and the variables `v` and `d`, and instead use the global variables of the fields to refer to the current value of a field. The implementation of this idea can be seen in the `SimpleCell` example on line 21, where the `val` field is existentially quantified and the Boogie global variable `val[this]` is used instead in the definition of the predicate. Figures 4.15, 4.16 and 4.17 express the fact that we side-effect the method/predicate surrounding formula R to add the existential parameter t. We assume that the parameter t does not capture any other parameters. This parameter is added to the list of parameters of the enclosing method/parameter in which the formula is.

Translation of object propositions

An object proposition `r@k Q`$(\bar{t})$ is translated by asserting that the value of the `packedQ` map for the parameters t and reference `r` is true and the value of `fracQ` for the same parameters and reference is $>= k$ if k is a constant or is > 0 if k is a bound variable. Figure 4.19 only looks at object propositions in isolation. According to the grammar in Section 3.1 other formulas R could be $\exists$z:double.z $\geq c$ t $\Rightarrow$`r@z Q`$(\bar{t})$, or $\forall$z:double.z $\geq c$ $\Rightarrow$`r@z Q`$(\bar{t})$. If there is no field corresponding to parameter `t`, we will not add the equality `field1[r] == t` to the translation of the object proposition.

When an object proposition `r@k Q`$(\bar{t})$ is unpacked, `unpacked(r@k Q`$(\bar{t})$`)` will appear in the annotations. This syntax is only used in pre- and post-conditions of methods, and when we need to add the fractions of two object propositions inside a method, one of which is unpacked. When we do not know the exact value of a field `f` for object `obj`, we employ two tactics. The first one is to use `obj.f` as the actual parameter (for example, `unpacked(op@k1 left(op.left, op.left.count))`), which means that the value of field `f` for object `obj` is its value at this time, whatever that is. This is similar to the *ghost fields* idea [48] from K. Rustan M. Leino's research. The second one is to use an existentially quantified expression such as `exists c:int` (for example) in the pre-condition of a method and use `c` as the value of the field `obj.f` mentioned above. In both cases the value of the parameter is equal to the current value of the field, the value that the field points to. The current implementation of Oprop does not equate the value of fraction `fracQ[r]` with `k` and instead it simply states that `fracQ[r] > 0.0` when this fraction is a metavariable. This current implementation does not keep track of the exact value of fractions when they are existentially quantified, but we need to keep track of their exact value in order for the translation to be sound. All the examples that we have translated are sound when using `fracQ[r] > 0.0` whenever there is an existentially quantified object proposition in the pre-condition and a separate one in the post-condition. We leave the implementation of Oprop where we set `fracQ[r]` to be equal to the metavariable `k` as future work

```
function translateObjectProposition(r@k Q(t))
returns String {
    String result = "";
    result += "(fracQ[r] == k)";
    if (k is a metavariable) {
        result += " && (k > 0.0) && (k < 1.0)";
    }
    if parameter t corresponds to a field of r {
        say that field is field1;
        result += " && (field1[r]==t)";
    }
    if parameter t does not correspond to a field of r {
        say X is the formal parameter of Q corresponding to t;
        result += " && (paramQX[r]==t)";
    }

    return result;
}
```

Figure 4.19: translateObjectProposition() translation helper function

- the needed implementation changes are minimal and not needed for the translations of all the examples in this thesis.

The definition of P from the Oprop grammar from Section 3.1 is:

P ::= r@k Q ($\bar{\text{t}}$) | `unpacked(`r@k Q ($\bar{\text{t}}$)`)` |
$r.f \rightarrow$ x | t binop t

The translation of P is:

trans(P) ::= `packedQ[r]` &&
translateObjectProposition(`r@k Q(`$\bar{t}$`)`) |
`(packedQ[r] == false)` &&
translateObjectProposition(`r@k Q(`$\bar{t}$`)`) |
`f[r]==`x `&& fFieldPermission[r]==true` | t trans(binop) t

If `packedQ[r]` is `true` in our implementation it means that we know for sure that the object that `r` points to is packed. If `packed[r]` is `false` it means that we know for sure that it is unpacked. When we do not know for sure if the object is packed or unpacked, we do not use either of the statements `packedQ[r] == true` or `packedQ[r] == false`; as you can see in the translation trans(PredDecl), mentions of `packedQ` are removed from the body of predicates.

It could happen that inside a method body a method call uses only a fraction of an unpacked object proposition about `r` and the postcondition of the method states that the same object proposition is packed. In cases like these, which appear when there is a method call inside a method body, the `packedQ[r]` should be set to `false` because there is at least a fraction of an object proposition that is `false`. This is the only case that can lead to a fraction of the same object proposition be in the packed state and another fraction be in the unpacked state. The crux of soundness in our system is that an unpacked predicate cannot be unpacked again. Thus even if only a fraction of an object proposition *r@k Q($\bar{t}$)* is unpacked, $packedQ[r]$ should be $false$. This idea is reflected in the *coalescePacked()* translation function in Figure 4.21.

The intuition is the following: in the pre-condition of a method, we will have either `packedQ[r] == true` or `packed[r] == false` for a particular object `r`. If in the body of the method there is a call to another method `m1()` that changes `packed[r]` from `true` to `false`, even if only for a fraction referring to `r` and predicate `Q` and not for all the existing fraction, then after the call to the method `m1()` `packedQ[r]` will be equal to `false`.

We need a map `fFieldPermission` for each field `f` of an object, that keeps track of that field. This is because fields are considered resources and they cannot be duplicated. Since the translation `f[r]==x` can also be obtained from translating an object proposition, we need to distinguish how we express what the current value of a field is from the fact that a field permission is available to be used. Our current implementation of Oprop does not have maps such as `fFieldPermission` because we observed the need for them when proving the soundness theorem of the translation. It would not be difficult to add these maps and we leave that as future work.

When we unpack an object proposition that has another object proposition inside its definition, we only get a fraction to the object proposition inside and thus this case cannot lead to the same situation just described.

You can see the translation of the packed object proposition `obj#0.34 PredVal()` both inside the `PackPredNext` and `UnpackPredNext` procedures corresponding to the predicate `PredNext`.

A fraction (having a numerator and a denominator) in Oprop is translated to its real representation in Boogie.

The `fracQ` global map represents the current fraction we hold to each object for predicate `Q`. When verifying a procedure, we start with the `fracQ` in that procedure's `requires` clause.

Let us say we are monitoring `fracQ` for the object `obj` and initially we have `fracQ[obj] == 1.0`. Whenever we call a `Pack` procedure for which the `requires` states that `fracQ[obj] == x`, the current value of `fracQ[obj]` becomes `1.0 - x`. If now we call an `Unpack` procedure for which the `ensures` states that `fracQ[obj] >= y`, the current value of `fracQ[obj]` becomes `1.0 - x + y`. In this way we keep track of the fraction to an object through a certain predicate. We start with the fraction from the `requires` clause of the current procedure. When a `Pack` procedure is called that requires a certain fraction, we subtract that amount from the current fraction. When an `Unpack` procedure is called that ensures a certain fraction, we add to the current fraction. For the `SimpleCell` example you can see that on line 93 the `changeVal` method is called. Since both the precondition and the postcondition of the method require a fraction of `k` to the predicate `PredVal` for `this` object, they cancel each other and that is why you do not see any statements that manipulate the fractions after the call to the method `changeVal`. If the fraction required in the precondition was different that the fraction ensured by the postcondition, you would see the fraction manipulation statements after the call.

Translation of packing/unpacking

When we pack an object to a predicate `Q`, we write the statement `call PackQ(..., this)`. Right after this statement, we write `packedQ[this] := true`. You can see an example of such a call on lines 71 to 74, together with the statements that assign `true` to the global `packed` map for this object and the statement that subtracts the fraction that is used for the predicate `PredVal` and the object `c` when the packing occurs.

Similarly, when we unpack an object from an object proposition that refers to predicate `Q`, we write the statement `call UnpackQ(..., this)`. Right after this statement we write `packedQ[this] := false`. You can see an example of such a call on lines 87 to 89. As opposed to packing, where we usually consume a fraction of an object proposition, when we unpack an object proposition we obtain a fraction and so we have a fraction manipulation statement that adds to the current value of a fraction, as seen on line 88.

The fraction k of an object proposition `r@k Q(`$\bar{(t)}$`)` can be retrieved from the global map `fracQ` corresponding to the predicate `Q`, under entry `r`.

In Figure 4.20, a fraction `frac1` can be a metavariable or a constant, and the manipulation of fractions is different depending on this differentiation. In the pseudocode we refer to `precondition(predicate1)` and `postcondition(predicate1)`; although a predicate has a single definition, the precondition of a predicate refers to the precondition of the corresponding `Pack` function (or `Unpack` function, depending which one it is called). The `Pack` and `Unpack` functions corresponding to predicates do have preconsitions and postconditions.

In the first `for` loop in Figure 4.20 - for fractions appearing in the preconditions of a predicate, if the fraction is a metavariable we divide the current value of the fraction by 2. This is because when the specification states that a fraction of `k` will be consumed, we do not know exactly how much of the fraction will be consumed. We decided to divide by 2 to state that half of the fraction will be consumed. For metavariable fractions appearing in the postconditions of a predicate we again do not know the exact value that will be added to the existing value of a fraction. We could write the symmetrical statement and multiply by 2 the current value of the fraction that we already have, but the problem with that is that sometimes we start with zero for a particular fraction (for example in the beginning of a procedure). Even if we multiply by 2, we would still have 0. That is why we add the 0.1 (0.1 is a random value that we chose) to the current value of the fraction. When packing a predicate, the fractions that we need to consume will appear in the preconditions of the `Pack` procedure for that predicate, so the codepath will go in the first `for` loop of Figure 4.20. On the other hand when we unpack a predicate the codepath will go in the second `for` loop of Figure 4.20.

Translation of expressions

The definition of e in the Oprop grammar from Section 3.1 is:

$$
\begin{array}{lll}
\mathsf{e} & ::= & \mathsf{t} \mid r.f \mid r.f = \mathsf{t} \mid r.m(\bar{\mathsf{t}}) \mid \\
 & & \texttt{new}\, C(Q(\bar{\mathsf{t}})\ [\bar{\mathsf{t}}])\ (\bar{\mathsf{t}}) \mid \\
 & & \texttt{if}\ (\mathsf{t})\ \{\ \mathsf{e}\ \}\ \texttt{else}\ \{\ \mathsf{e}\ \} \mid \texttt{let}\ \mathsf{x} = \mathsf{e}\ \texttt{in}\ \mathsf{e} \mid \\
 & & \mathsf{t}\ \mathsf{binop}\ \mathsf{t} \mid \mathsf{t}\ \&\&\ \mathsf{t} \mid \mathsf{t} \,\|\, \mathsf{t} \mid\ !\ \mathsf{t} \mid \\
 & & \texttt{pack}\ r@\mathsf{k}\ Q(\bar{\mathsf{t}})\ [\bar{\mathsf{t}}]\ \texttt{in}\ \mathsf{e} \mid \texttt{unpack}\ r@\mathsf{k}\ Q(\bar{\mathsf{t}})\ [\bar{\mathsf{t}}]\ \texttt{in}\ \mathsf{e}
\end{array}
$$

```
function writeFractionManipulations(String predicate1)
returns String {
    String result = "";
    foreach (objectProposition objProp1 in precondition(predicate1) {
        if (fraction frac1 in objProp1 is a metavariable) {
            result += "frac"+objProp1.predicateName + "[" +
                    objProp1.object + "]:="+
                    "frac"+objProp1.predicateName + "[" +
                    objProp1.object + "]/2.0;";
        } else {
            result += "frac"+objProp1.predicateName + "[" +
                    objProp1.object + "]:="+
                    "frac"+objProp1.predicateName + "[" +
                    objProp1.object + "]-"+
                    objProp1.fraction+";";
        }
    }
    foreach (objectProposition objProp1 in postcondition(predicate1) {
        if (fraction frac1 in objProp1 is a metavariable) {
            result += "frac"+objProp1.predicateName + "[" +
                    objProp1.object + "]:="+
                    "frac"+objProp1.predicateName + "[" +
                    objProp1.object + "] + 0.1;";
        } else {
            result += "frac"+objProp1.predicateName + "[" +
                    objProp1.object + "]:="+
                    "frac"+objProp1.predicateName + "[" +
                    objProp1.object + "]+"+
                    objProp1.fraction+";";
        }
    }
    return result;
}
```

Figure 4.20: writeFractionManipulations() translation helper function

```
function coalescePacked(m: MthDecl) {
    String result = "";
    for each (objProp obj in postcondition(m)) {
        let r be the reference and Q the predicate of obj;
        if (packedQ[r] is true before calling m,
           but packedQ[r] is false in the postcondition)
        {
            result+= "packedQ[r] := false";
        } else
        if (packedQ[r] is false before calling m,
           but packedQ[r] is true in the postcondition)
        {
            result+= "packedQ[r] := false";
        }
    }
    return result;
}
```

Figure 4.21: coalescePacked() translation helper function

The translation of expressions e is:

trans(e) ::= trans(t) | `f[r]` | `f[r]:=` trans(t) | `call m(r, trans(`$\bar{t}$`));`
writeFractionManipulations(m); coalescePacked(m) |
`var c: Ref; call C(`$\overline{t3}$`,c);packedQ[c]:=false;`
`call PackQ(`$\bar{t}$`,c); packedQ[c]:=true;`
`fracQ[c]:=1.0;` | writeFractionManipulations(Q);
`call PackQ(trans(`$\overline{t1}$`), trans(`$\overline{t2}$`), c);`
`packedQ[c]:=true; fracQ[c]:=1.0;`
`if` (trans(t)) { trans(e1) } `else` { trans(e2) } |
x := trans(e1); trans(e2) |
trans(t) trans(binop) trans(t) | trans(t) && trans(t) |
trans(t) || trans(t) | ! trans(t) |
`call PackQ(trans(`$\bar{t}$`), trans(`$\bar{t}$`), this);`
`packedQ[this]:=true;`
writeFractionManipulations(Q); trans(e); |
`call UnpackQ(trans(`$\bar{t}$`), trans(`$\bar{t}$`), this);`
`packedQ[this]:=false;`
writeFractionManipulations(Q); trans(e);

In the translation of expressions, e is expression-based, but the object creation case (`new`) and the `pack` and `unpack` cases do not generate expressions in the Boogie translation. Instead we translate to A-normal form in these cases.

There are situations when the proof has two cases. For example in a binary tree, it might be that we are going on the right branch or the left branch. This is the case in the Composite example, that is described in detail in Section 5.4. The Composite example in this thesis is one possible instantiation of the Composite design pattern. Our implementation is as an acyclic binary tree, where each Composite object has a reference to its left and right children and to its parent. At some point in the proof of the composite it happens that `this==this.parent.left` or `this==this.parent.right`. In this case, we add an `if-else` statement as part of the verification to illustrate the two cases, on lines 124, 137 and 150 in Figure 5.44. This `if` needs to be added by the programmer as annotations in the Java code as a normal Java *if* expression. In each branch of the `if` there are different `Pack/Unpack` annotations and since my Oprop tool is not inferring the `Packing/Unpacking` annotations, the programmer is the one that has to write in each branch of the `if` the right `Pack/Unpack` annotations. Also, this separation into cases happens at a random point in the proof, so Boogie would not have a way to know where to insert the `if`.

If we have multiple declarations of the form `var c:  Ref`, `var d:  Ref`, we also add the assumption statement `assume (c!=d)` because the Boogie tool does not assume that these two variables are different, while the Java semantics does assume this.

A term can be a variable or a constant.

t ::= x | `n` | `null` | `true` | `false`

Variables come in two types: reference variables `r` and integer variables `i`.

x ::= `r` | `i`

trans(binop) ::= $+ \mid - \mid$ trans(%) $\mid == \mid != \mid <= \mid < \mid >= \mid >$

Boogie does not have the binary operator modulo `%`, but it can be represented by the function *modulo* and its associated axioms.

```
function modulo(x:int, y:int) returns (int);
axiom (forall x:int, y:int ::  {modulo(x,y)}
 modulo(x,y) == x - x/y * y
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)));
```

The types used in Oprop, according to the grammar in Section 3.1 are

T ::= *C* | `int` | `boolean` | `double`.

Their translation is: trans(T) ::= `Ref` | `int` | `bool` | `real`. These are the basic types that we use, but one can define new ones in Boogie, based on the existing ones.

4.3.2 Other Approaches to Logical Encodings

Other researchers have encoded linear logic or fragments of it into first order logic. In his Ph.D. thesis [64] Jason Reed presents an encoding of the entirety of linear logic into first order logic. His encoding does not use fractions, so it is not possible for us to use it without enriching it with a way to encode fractions. While his encoding is sound, it is too complicated for our needs (considering that we only need to encode an extended fragment of linear logic). We decided against using his encoding since we were able to come up with a simpler encoding better suited for our system and our implementation. The major technical difference between Reed's encoding and ours is that he encodes uninterpreted symbols while our encoding is done inside the theory of object propositions - the smaller fragment of linear logic on top of which we have added the object propositions. His encoding is suited to any formula written in linear logic, irrespective of its meaning, while ours is targeted towards formulas written in our extended fragment of linear logic, that have a specific semantics.

Heule *et al.* [36] present an encoding of abstract predicates and abstraction functions in the verification condition generator Boogie. Their encoding is sound and handles recursion in a way that is suitable for automatic verification using SMT solvers. It is implemented in the automatic verifier Chalice. Since our system differs from theirs in the way we handle fractions (they need a full permission in order to be able to modify a field, while we are able to modify fields even if we have a fractional permission to the object enclosing the field), we came up with an encoding that is specific to our needs.

From a technical point of view, their encoding is quite different from ours. 'Upon a call, the caller relinquishes the required permissions (the caller exhales the precondition) and transfers them to the callee (the callee inhales them). Conversely, when a method terminates, the method exhales its postcondition, while its caller inhales it.' The $exhale$ operation aggressively havocs the heap and preserves information only for those locations to which the method holds direct or known-folded permission after the exhale. The reason they havoc the heap is so they don't have to compute transitive modifies clauses–which we have to do. This actually may make their verification more modular because they can do separate analysis whereas we need all code to be present in order to do the `modifies` clause analysis. This is one of the technical differences that make their encoding to be different than ours.

In their encoding, permissions are tracked using permission masks, which map locations to booleans. The variable $Mask$ stores the current mask, which represents the direct permissions held by the current method. The exhale operation recursively traverses the assertion to be exhaled, asserting all logical properties, and removing the required permissions from the current mask. In contrast to $exhale$, the translation of $inhale$ is parameterized by a mask because it sometimes operates on primary and sometimes on known-folded permissions stored in a particular predicate mask. The known-folded permissions of a predicate instance are those for which the program previously held the direct permission, or which are folded inside a predicate instance that has been retained up to the current program point. The known-folded permissions are very different conceptually from the fractional permissions that we use for object propositions and thus they lead to a different way of encoding. In their encoding, in addition to recording direct permissions, Heule *et al.* record for each predicate instance those locations to which the predicate holds known-folded permission.

4.4 Comparison between the JavaSyp Tool and the Oprop Tool

Kevin Bierhoff has implemented a tool called JavaSyp [7] that uses the SMT solver Z3 to formally verify Java code. Since JavaSyp implements a similar permission system, but for typestate instead of object propositions, we initially wanted to modify JavaSyp in order to implement Oprop. That proved to be a difficult task because there are many details that are different between JavaSyp and what Oprop needs to do:

- JavaSyp uses borrowing and capture/release because the tool does not implement fractional permissions. Oprop does not use borrowing, but instead it uses fractional permissions. Fractions give more precision than the borrowing mechanism and so we have implemented fractional permissions.

- Oprop will implement the pack/unpack mechanism, while JavaSyp does not implement this mechanism. JavaSyp implements instead "exposure blocks" that show how fields should be accessed. These features are closely related: when the fields of an object are unpacked (when the object proposition that encapsulated them is unpacked), we can think of them as being "exposed". In Bierhoff's system, there are "unique" and "immutable" exposure blocks. Fields can be assigned inside unique exposure blocks, with field reads yielding the field's original permission. Inside an immutable block, reading fields results in a weakened field permission. The technical difference is that in our system we do not have immutable permissions, but instead one can always write to the fields of an object (in some cases, one has to make sure that the invariant is preserved). We acknowledge that this is just an incidental difference and the ideas of pack/unpack vs. exposure blocks are very similar.
- JavaSyp has state invariants, but Oprop will not have them and instead it will have invariants for objects that are shared.

4.5 Equivalence of the Oprop Language and its Translation into the Boogie Language

The soundness theorem below formally states that if the translation into Boogie of an Oprop formula holds in first order logic then the initial formula holds in our linear theory. The completeness theorem below states the opposite direction: if a formula holds in our linear theory then its translation will hold in first order logic. By proving both the soundness and the completeness theorems we prove that the formulas of the Oprop language and their translation into formulas written in Boogie, as described above, are equivalent.

We are not stating that the whole Oprop language is equivalent to its translation into Boogie–just formulas. Boogie's semantics is defined via trace sets and the trace set style of semantics is quite distant from the standard dynamic semantics approach that we use. Furthermore modeling all the detailed constructs in Java has been done in other settings and is not interesting–but the semantics of the formulas are at the heart of translating the linear theory down to Boogie so that is what we chose to focus on for equivalence purposes. Later in Section 4.6 we have a separate, less formal, discussion of why the other parts of the translation are sound.

We present the semantic rules for our linear theory formulas in Figures 4.22 and 4.23. In these figures Γ contains typings for term variables, Π_0 contains the persistent truths and Π_1 contains the resources. We have adapted these semantic rules from Prof. Frank Pfenning's 2012 Carnegie Mellon course on linear logic [63]. We have divided the Π that we used in Section 3.1 into Π_0 and Π_1, to separate the persistent truths and the ephemeral resources. In fact the context Π contains the preconditions of the particular method inside which we have to prove a formula R and we have the equality $\Pi = \Pi_0; \Pi_1$. We have already given a number of semantic rules for our system in Section 3.3. While the rules in Section 3.3 use entailment, we have not precisely defined it there as the relationship is mostly standard. We are presenting it in this section to both

$$\frac{\Gamma;\Pi_0;\Pi_1 \vdash\ r@\mathsf{k/2}\ Q\,(\bar{\mathsf{t}})\ \otimes r@\mathsf{k/2}\ Q\,(\bar{\mathsf{t}})}{\Gamma;\Pi_0;\Pi_1 \vdash\ r@\mathsf{k}\ Q\,(\bar{\mathsf{t}})}\ (OPack_1)$$

$$\frac{\Gamma;\Pi_0;\Pi_1 \vdash\ r@\mathsf{k*2}\ Q\,(\bar{\mathsf{t}})}{\Gamma;\Pi_0;\Pi_1 \vdash\ r@\mathsf{k}\ Q\,(\bar{\mathsf{t}})\ \otimes r@\mathsf{k}\ Q\,(\bar{\mathsf{t}})}\ (OPack_2)$$

$$\frac{\Gamma;\Pi_0;\Pi_1 \vdash\ r@\mathsf{k/2}\ Q\,(\bar{\mathsf{t}})\ \otimes \texttt{unpacked}(r@\mathsf{k/2}\ Q\,(\bar{\mathsf{t}})\,)}{\Gamma;\Pi_0;\Pi_1 \vdash\ \texttt{unpacked}(r@\mathsf{k}\ Q\,(\bar{\mathsf{t}})\,)}\ (OUnpack_1)$$

$$\frac{\Gamma;\Pi_0;\Pi_1 \vdash\ \mathsf{R1} \quad \Gamma;\Pi_0;\Pi_1' \vdash\ \mathsf{R2}}{\Gamma;\Pi_0;\Pi_1,\Pi_1' \vdash\ \mathsf{R1} \otimes \mathsf{R2}}\ (\otimes)$$

$$\frac{\Gamma;\Pi_0;\Pi_1 \vdash\ \mathsf{R1}}{\Gamma;\Pi_0;\Pi_1 \vdash\ \mathsf{R1} \oplus \mathsf{R2}}\ (\oplus_L)$$

$$\frac{\Gamma;\Pi_0;\Pi_1 \vdash\ \mathsf{R2}}{\Gamma;\Pi_0;\Pi_1 \vdash\ \mathsf{R1} \oplus \mathsf{R2}}\ (\oplus_R)$$

Figure 4.22: Semantics for Linear Theory Formulas

define precisely what it means in our system and allow us to prove properties about it. The rules in Figures 3.1 and 3.2 are similar to the rules that we present in this section in Figure 4.23, but they are only simplified versions of the ones in the current section. The details of the rules from Figure 4.23 are needed in our proof below, that shows that the translation of formulas into Boogie is both sound and complete.

Theorem 2 (Soundness Theorem)**.** *For a formula R that is written in our linear theory and parses according to the grammar in Section 3.1, if trans*$(\Gamma;\Pi_0;\Pi_1) \vdash^{FOL}$ *trans(R) then* $\Gamma;\Pi_0;\Pi_1 \vdash^{LL}$ *R.*

Proof. The proof will follow the cases of R in the grammar in Section 3.1, but it will be done by induction on the complexity of formula R, i.e., on the depth of logical connectives in the formula. For each formula R we have to prove that if trans(R) is true in first order logic then R is true in our linear theory.

- **Case** $P3$
  To prove: If trans$(\Gamma;\Pi_0;\Pi_1) \vdash^{FOL}$ trans(P) then $\Gamma;\Pi_0;\Pi_1 \vdash^{LL}$ P, where:

  1. R = P = $r.f \rightarrow$ x
  2. trans(P) $= (f[r] == x)\ \&\&\ (fFieldPermission[r] == true)$

  Proof: This proof is based on the structure of Π, which shows that $r.f \rightarrow x$ can only be at the top level of Π or nested inside a universal or existential quantifier. The definition of Π is presented in Section 3.3 and reproduced below:

$$\frac{\Gamma, \text{M:T}; \Pi_0; \Pi_1, \text{R\{M/x\}} \vdash C}{\Gamma; \Pi_0; \Pi_1, \exists\text{x:T.R} \vdash C}\ (\exists_1 L)$$

$$\frac{\Gamma \vdash \text{M:T} \quad \Gamma; \Pi_0; \Pi_1 \vdash \text{R\{M/x\}}}{\Gamma; \Pi_0; \Pi_1 \vdash \exists\text{x:T.R}}\ (\exists_1 R)$$

$$\frac{\Gamma, \text{F:double}; \Pi_0, \text{F} > 0; \Pi_1, \text{R\{F/z\}} \vdash C}{\Gamma; \Pi_0; \Pi_1, \exists\text{z:double.R} \vdash C}\ (\exists_2 L)$$

$$\frac{\Gamma \vdash \text{F:double} \quad \Pi_0 \vdash \text{F} > 0 \quad \Gamma; \Pi_0; \Pi_1 \vdash \text{R\{F/z\}}}{\Gamma; \Pi_0; \Pi_1 \vdash \exists\text{z:double.R}}\ (\exists_2 R)$$

$$\frac{\Gamma, \text{F:double}; \Pi_0, \text{F binop t}; \Pi_1, \text{R\{F/z\}} \vdash C}{\Gamma; \Pi_0; \Pi_1, \exists\text{z:double.z binop t} \Rightarrow \text{R} \vdash C}\ (\exists_3 L)$$

$$\frac{\Gamma \vdash \text{F:double} \quad \Pi_0 \vdash \text{F binop t} \quad \Gamma; \Pi_0; \Pi_1 \vdash \text{R\{F/z\}}}{\Gamma; \Pi_0; \Pi_1 \vdash \exists\text{z:double.z binop t} \Rightarrow \text{R}}\ (\exists_3 R)$$

$$\frac{\Gamma \vdash m : T; \Gamma; \Pi_0; \Pi_1, \text{R\{m/x\}} \vdash C}{\Gamma; \Pi_0; \Pi_1, \forall\text{x:T.R} \vdash C}\ (\forall_1 L)$$

$$\frac{\Gamma, m : T; \Pi_0; \Pi_1 \vdash \text{R\{m/x\}}}{\Gamma; \Pi_0; \Pi_1 \vdash \forall\text{x:T.R}}\ (\forall_1 R)$$

$$\frac{\Gamma \vdash \text{f:double}; \Gamma, \Pi_0, \text{f > 0}; \Pi_1, \text{R\{f/z\}} \vdash C}{\Gamma; \Pi_0; \Pi_1, \forall\text{z:double.R} \vdash C}\ (\forall_2 L)$$

$$\frac{\Gamma, \text{f:double}; \Pi_0, \text{f > 0}; \Pi_1 \vdash \text{R\{f/z\}}}{\Gamma; \Pi_0; \Pi_1 \vdash \forall\text{z:double.R}}\ (\forall_2 R)$$

$$\frac{\Gamma \vdash \text{f:double}; \Gamma, \Pi_0, \text{f binop t}; \Pi_1 \text{R\{f/z\}} \vdash C}{\Gamma; \Pi_0; \Pi_1, \forall\text{z:double.z binop t} \Rightarrow R \vdash C}\ (\forall_3 L)$$

$$\frac{\Gamma, \text{f:double}; \Pi_0, \text{f binop t}; \Pi_1 \vdash \text{R\{f/z\}}}{\Gamma; \Pi_0; \Pi_1 \vdash \forall\text{z:double.z binop t} \Rightarrow R}\ (\forall_3 R)$$

$$\frac{\Gamma; \Pi_0; \Pi_1 \vdash \text{t1 binop t2} \Rightarrow \text{R}}{\Gamma; \Pi_0, \text{t1 binop t2}; \Pi_1 \vdash \text{R}}\ (tbint)$$

$$\frac{}{\Gamma; \Pi_0; \Pi_1, \text{R} \vdash \text{R}}\ (id)$$

Figure 4.23: Semantics for Linear Theory Formulas - cont.

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

For $(f[r] == x)$ && $(fFieldPermission[r] == true)$ to be the conclusion, it must appear in the context Π. We need the $(fFieldPermission[r] == true)$ part of the translation to distinguish from $f[r] = x$ that could come from the ***translateObjectProposition()*** in Figure 4.19. The structure of Π and the translation rules mean that $(f[r] == x)$&& $(fFieldPermission[r] == true)$ can only be at the top level of trans(Π) or nested inside a universal or existential quantifier. We could imagine an implication being introduced in the definition of Π above, but it could be easily eliminated and removed from any proof. So let us assume a cut-free proof, without loss of generality. $(f[r] == x)$ && $(fFieldPermission[r] == true)$ can only come from an identity rule or via a quantifier elimination rule. The identity and quantifier elimination rules are analogous for our linear theory and FOL, so a structurally equivalent proof must exist in the linear system. This means that $r.f \rightarrow x$ can only be obtained via an identity rule or a quantifier elimination rule in the linear theory. In both cases we get that $\Gamma; \Pi_0; \Pi_1 \vdash^{LL} r.f \rightarrow x$, i.e., $\Gamma; \Pi_0; \Pi_1 \vdash^{LL} \mathsf{P}$.

- **Case** $P1$

To prove: If trans($\Gamma; \Pi_0; \Pi_1$) $\vdash^{FOL}$ trans(P) then $\Gamma; \Pi_0; \Pi_1 \vdash^{LL}$ P, where:

1. R = P = *r*@k *Q* $(\bar{\mathsf{t}})$
2. trans(P) = `packedQ[r]` && translateObjectProposition(`r@k Q(`$\bar{t}$`)`)

Proof: The definition of the function translateObjectProposition() defined in Figure 4.19 states:

```
if parameter t corresponds to a field of r {
    say that field is field1;
    result += "(field1[r]==t)";
}
if parameter t does not correspond to a field of r {
    say X is the formal parameter of Q corresponding to t;
    result += " && (paramQX[r]==t)";
}
```

This means that trans([*r*/*this*]*Q* $(\bar{\mathsf{t}})$) holds in FOL. Using the induction hypothesis we obtain that [*r*/*this*]*Q* $(\bar{\mathsf{t}})$ holds in LL. Also from the definition of the function translateObjectProposition() we know that

```
result += "(fracQ[r] == k)";
if (k is a metavariable) {
    result += " && (k > 0.0) && (k < 1.0)";
}
```

This means that in trans(Π_1) there is a value $fracQ$[r] that corresponds to k. We also know that in trans(Π_1) we have `packedQ[r]`. Since we devised $fracQ$[r] and `packedQ[r]` to mean exactly the value of the fraction and the packing state corresponding to the object proposition *r*@k *Q* $(\bar{\mathsf{t}})$, we have all the ingredients showing that *r*@k *Q* $(\bar{\mathsf{t}})$ holds in LL.

- **Case** $P2$

  To prove: If trans($\Gamma;\Pi_0;\Pi_1$) $\vdash^{FOL}$ trans(P) then $\Gamma;\Pi_0;\Pi_1 \vdash^{LL}$ P, where:

  1. R = P = `unpacked(`*r*@k *Q* `(`$\bar{\mathrm{t}}$`))`
  2. trans(P) = `(packedQ[r]==false)` &&
     translateObjectProposition(`r@k Q(`$\bar{t}$`))`

  Proof:
  The definition of the function translateObjectProposition() defined in Figure 4.19 states:

  ```
  if parameter t corresponds to a field of r {
      say that field is field1;
      result += "(field1[r]==t)";
  }
  if parameter t does not correspond to a field of r {
      say X is the formal parameter of Q corresponding to t;
      result += " && (paramQX[r]==t)";
  }
  ```

  This means that the fields of r are equal to the corresponding parameters of predicate Q in FOL, whether they are the actual fields coming from the parameter of an existential expression, or they are the formal fields, as discussed when we defined the translation function translateObjectProposition() in Figure 4.19. Using the induction hypothesis (case $P3$) we obtain that those fields point to the values of the corresponding parameters of predicate Q in LL.
  Also from the definition of the function translateObjectProposition() we know that

  ```
  result += "(fracQ[r] == k)";
  if (k is a metavariable) {
      result += " && (k > 0.0) && (k < 1.0)";
  }
  ```

  This means that in trans(Π_1) there is a value $fracQ$[r] that corresponds to k. We also know that in trans(Π_1) we have `packedQ[r]==false`. Since we devised $fracQ$[r] and `packedQ[r]` to mean exactly the value of the fraction and the packing state corresponding to the object proposition *r*@k *Q* `(`$\bar{\mathrm{t}}$`)`, we have all the ingredients showing that `unpacked(`*r*@k *Q* `(`$\bar{\mathrm{t}}$`))` holds in LL.

- **Case** $P4$

  To prove: If trans($\Gamma;\Pi_0;\Pi_1$) $\vdash^{FOL}$ trans(P) then $\Gamma;\Pi_0;\Pi_1 \vdash^{LL}$ P, where:

  1. R = P = t binop t
  2. trans(P) = t trans(binop) t

  Proof:
  For all but the `modulo` binary operator, it is straightforward to see that if the translation of the binary expression holds in FOL, then the original expression holds in LL. Because division and modulo are defined differently in different source languages, Boogie provides syntax for the operators and $\%$ but gives them no meaning. Instead, the meaning of these operators can be axiomatized according to their desired meaning, as can be seen below.

We have taken the definition of the `modulo` operator from the official Boogie manual [50] available online at `http://research.microsoft.com/~leino/papers.html`, and one can see that if the axioms below are holding in FOL, then $x\%y == modulo(x, y)$.

```
 function modulo(x:int, y:int) returns (int);
 axiom (forall x:int, y:int ::  {modulo(x,y)}
  modulo(x,y) == x - x/y * y
 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)));
```

- **Case** $R\otimes$
  To prove: If trans($\Gamma; \Pi_0; \Pi_1$) $\vdash^{FOL}$ trans(R) then $\Gamma; \Pi_0; \Pi_1 \vdash^{LL}$ R, where:
  1. R = R1 $\otimes$ R2
  2. trans(R) = translateAnd(R1,R2)

  Proof:
  The definition of the translateAnd() function in Figure 4.18 first rewrites the initial formula R1 $\otimes$ R2 into disjunctive normal form (DNF). Since we have a formula in DNF, we will have a number of conjunctions combined with `||` operators. Within each conjunction the `coalesce()` function in the translation adds the fractions in all the appearances of `ri@_ Qi(ti)`. Whether `ri@_ Qi(ti)` is packed or unpacked, the corresponding fractions are summed up into `ri@ki Qi(ti)`. After this rewrite, the translation of each conjunction has the form
  trans(`r1@(k1+k'1) Q1(t1)`) && trans(`r2@(k2+k'2) Q2(t2)`) && .. &&
  trans(`rn@(kn+k'n) Qn(tn)`) && trans(`R1leftover`) && trans(`R2leftover`).
  By the induction hypothesis we get that the component formulas hold in LL, i.e., the formulas `r1@(k1+k'1) Q1(t1)`, `r2@(k2+k'2) Q2(t2)`, .., `rn@(kn+k'n) Qn(tn)`, `R1leftover`, `R2leftover` simultaneously hold in LL. The induction hypothesis above considers each disjunct separately. The only case where a conjunction would not hold in LL when it holds in FOL is because the same resource is used twice; and the coalescing done in the translation rule ensures that the resources used in each part of the conjunction are different.
  Finally, the formula
  `r1@(k1+k'1) Q1(t1)` $\otimes$ `r2@(k2+k'2) Q2(t2)` $\otimes$ .. $\otimes$ `rn@(kn+k'n) Qn(tn)` $\otimes$ `R1leftover` $\otimes$ `R2leftover` holds in LL, i.e., using the $\otimes$ linear theory rule we obtain that the formula R1 $\otimes$ R2 holds in LL.
- **Case** $R\oplus$
  To prove: If trans($\Gamma; \Pi_0; \Pi_1$) $\vdash^{FOL}$ trans(R) then $\Gamma; \Pi_0; \Pi_1 \vdash^{LL}$ R, where:
  1. R = R1 $\oplus$ R2
  2. trans(R) = trans(R1) || trans(R2)

  Proof:

If trans(R1) || trans(R2) holds in FOL, it means that trans(R1) or trans(R2) holds in FOL. Let's assume that trans(R1) holds in FOL, since the proof is similar if trans(R2) holds. By the induction hypothesis, we know that R1 holds in LL. By the $\oplus_L$ linear theory rule, this means that R1 $\oplus$ R2 holds in LL.

- **Case** $R\exists_1$
  To prove: If trans($\Gamma; \Pi_0; \Pi_1$) $\vdash^{FOL}$ trans(R) then $\Gamma; \Pi_0; \Pi_1 \vdash^{LL}$ R, where:
  1. R = ∃x:T.R1
  2. trans(R) = trans(R1); `addExistential`(trans(R1), x:trans(T))

  Proof:
  We know that trans(R1); `addExistential`(trans(R1), x:trans(T)) holds in FOL. This means that the formula R1 is first translated and then the translation function `addExistential()`, for which we present the definition below, adds the parameter x:trans(T) to the signature of the method or predicate in which R1 is found. Adding the existentially quantified variable x to the surrounding method has the effect, from the point of view of the formula, of extending the context with x. From the induction hypothesis we know that R1, that contains the x variable, holds in LL. The `addExistential()` writes the type of the variable x and it bounds it to the enclosing method or predicate. This does not change the semantics of the R1 formula, which still holds in LL. Thus we obtain that ∃x:T.R1 holds in LL, i.e., $\Gamma; \Pi_0; \Pi_1 \vdash^{LL}$ R.

```
function addExistential(trans(R1), x:trans(T))
{
    MethodOrPredDeclaration result;
    let methodOrPred(params) be the method
    or predicate in the body of which R is found;
    update methodOrPred(params) to be methodOrPred(params, t:
       trans(T));
    return trans(R);
}
```

- **Case** $R\exists_2$
  To prove: If trans($\Gamma; \Pi_0; \Pi_1$) $\vdash^{FOL}$ trans(R) then $\Gamma; \Pi_0; \Pi_1 \vdash^{LL}$ R, where:
  1. R = ∃z:double.R
  2. trans(R) = trans(R); `addExistentialFrac`(trans(R), z:trans(double))

  Proof:
  We know that trans(R); `addExistentialFrac`(trans(R), z:trans(double)) holds in FOL. This means that the formula R is first translated and then the function `addExistentialFrac()`, for which we present the definition below, adds the parameter `k:real` to the enclosing method or predicate signature. By the induction hypothesis we obtain that R holds in LL. We then see that the value $fracQ[r]$ is the witness for z needed by the formula ∃z:double.R in LL. Thus this formula holds in LL.

```
function addExistentialFrac(trans(R), k:trans(double))
{
    MethodOrPredDeclaration result;
    let methodOrPred(params) be the method
    or predicate in the body of which R is found;
    update methodOrPred(params) to be methodOrPred(params, k:real
       );
    return trans(R);
}
```

- **Case** $R\exists_3$
  To prove: If trans($\Gamma; \Pi_0; \Pi_1$) $\vdash^{FOL}$ trans(R) then $\Gamma; \Pi_0; \Pi_1 \vdash^{LL}$ R, where:
  1. R = ∃z:double.z binop t ⇒ R
  2. trans(R) = trans(R);
     `addExistentialFracBinop`(trans(R), z:trans(double), `z binop t)`

  Proof:
  The proof in this case is very similar to the proof in the $R\exists_1$ case, except that we have the extra condition `fracQ[r]` binop t on the value `fracQ[r]`. The definition of the translation function addExistentialFracBinop() is given below, and it shows that the only difference between this function and the translation function in the $R\exists_1$ case is the extra condition on `k`.

```
function addExistentialFracBinop(trans(R), k:trans(double), k
   binop t)
{
    MethodOrPredDeclaration result;
    let methodOrPred(params) be the method
    or predicate in the body of which R is found;
    update methodOrPred(params) to be methodOrPred(params, k:real
       ) and
    add "assert k binop t;"(or "k binop t") to body of
       methodOrPred;
    return trans(R);
}
```

- **Case** $R\forall_1$
  To prove: If trans($\Gamma; \Pi_0; \Pi_1$) $\vdash^{FOL}$ trans(R) then $\Gamma; \Pi_0; \Pi_1 \vdash^{LL}$ R, where:
  1. R = ∀t:T.R
  2. trans(R) = $var\ t$:trans(T); trans(R)

  Proof:
  We know that trans(R) holds in FOL, for every t :trans(T). Using the induction hypothesis, we obtain that R holds in LL for all $t : T$. This means that ∀t:T.R holds in LL.
- **Case** $R\forall_2$
  To prove: If trans($\Gamma; \Pi_0; \Pi_1$) $\vdash^{FOL}$ trans(R) then $\Gamma; \Pi_0; \Pi_1 \vdash^{LL}$ R, where:
  1. R = ∀z:double.R
  2. trans(R)$= var\ z : real$; trans(R)

  Proof:

We know that trans(R) holds in FOL, for every $z : real$. Using the induction hypothesis, we obtain that R holds in LL for all $z : double$. This means that ∀z:double.R holds in LL.

- **Case** $R\forall_3$
  To prove: If trans($\Gamma; \Pi_0; \Pi_1$) $\vdash^{FOL}$ trans(R) then $\Gamma; \Pi_0; \Pi_1 \vdash^{LL}$ R, where:
  1. R = ∀z:double.z binop t ⇒ R
  2. trans(R) = $var\ z : real$; z trans(binop) t ==> trans(R)

  Proof:
  We know that trans(R) holds in FOL, for every $z : real$, for which z trans(binop) t holds. Using the induction hypothesis, we obtain that R holds in LL for all $z : double$ for which z binop t holds. This means that ∀z:double.z binop t ⇒ R holds in LL.
- **Case** $R \Rightarrow$
  To prove: If trans($\Gamma; \Pi_0; \Pi_1$) $\vdash^{FOL}$ trans(R) then $\Gamma; \Pi_0; \Pi_1 \vdash^{LL}$ R, where:
  1. R = t1 binop t2 ⇒ R
  2. trans(R) = t1 trans(binop) t2 ==> trans(R)

  Proof:
  We know that trans(R) holds in FOL when t1 trans(binop) t2 holds. Using the induction hypothesis, we obtain that R holds in LL when t1 binop t2 holds. This means that t1 binop t2 ⇒ R holds in LL.

□

Theorem 3 (Completeness Theorem)**.** *For a formula R that is written in our linear theory and parses according to the grammar in Section 3.1 and assuming that the right witnesses are chosen for the existentially quantified variables in R and knowing the aliasing relations in R, i.e., which references present in R point to the same object, if $\Gamma; \Pi_0; \Pi_1 \vdash^{LL} R$ then trans$(\Gamma; \Pi_0; \Pi_1) \vdash^{FOL}$ trans(R).*

Proof. The proof will be done by induction on the derivation that R is true in our linear theory. For this we use the rules in Figures 4.22 and 4.23. For each formula R we have to prove that if R is true in the linear theory then trans(R) is true in first order logic.

- **Case** $OPack_1$
  To prove: If $\Gamma; \Pi_0; \Pi_1 \vdash^{LL}$ P then trans($\Gamma; \Pi_0; \Pi_1$) $\vdash^{FOL}$ trans(P) where:
  1. P = r@k Q ($\bar{\mathsf{t}}$)
  2. trans(P) = `packedQ[r]` && translateObjectProposition(`r@k Q(`$\bar{t}$`)`) and
  3. $$\frac{\Gamma; \Pi_0; \Pi_1 \vdash\ r@\mathsf{k/2}\ Q\,(\bar{\mathsf{t}}) \otimes r@\mathsf{k/2}\ Q\,(\bar{\mathsf{t}})}{\Gamma; \Pi_0; \Pi_1 \vdash\ r@\mathsf{k}\ Q\,(\bar{\mathsf{t}})}\ (OPack_1)$$

  Proof:
  From the $OPack_1$ rule we know that
  if $\Gamma; \Pi_0; \Pi_1 \vdash^{LL}$ r@k Q ($\bar{\mathsf{t}}$) then $\Gamma; \Pi_0; \Pi_1 \vdash^{LL}$ r@k/2 Q ($\bar{\mathsf{t}}$) $\otimes$ r@k/2 Q ($\bar{\mathsf{t}}$) . Using the induction hypothesis, we know that trans($\Gamma; \Pi_0; \Pi_1$) $\vdash^{FOL}$ trans(r@k/2 Q ($\bar{\mathsf{t}}$) $\otimes$ r@k/2 Q ($\bar{\mathsf{t}}$)). Using the definition of the translateAnd() function from Figure 4.18, we obtains that trans($\Gamma; \Pi_0; \Pi_1$) $\vdash^{FOL}$ trans(r@(k/2 + k/2) Q ($\bar{\mathsf{t}}$)), i.e., trans($\Gamma; \Pi_0; \Pi_1$) $\vdash^{FOL}$ trans(r@(k) Q ($\bar{\mathsf{t}}$)). This means that if P holds in LL then trans(P) holds in FOL. Q.E.D.

- **Case** $OPack_2$

  To prove: If $\Gamma; \Pi_0; \Pi_1 \vdash^{LL}$ P then trans($\Gamma; \Pi_0; \Pi_1$) $\vdash^{FOL}$ trans(P) where:

  1. P = r@k Q ($\bar{\mathsf{t}}$)
  2. trans(P) = `packedQ[r]` && translateObjectProposition(`r@k Q(`$\bar{t}$`)`) and
  3. $$\frac{\Gamma; \Pi_0; \Pi_1 \vdash\ r@\mathsf{k*2}\ Q\ (\bar{\mathsf{t}})}{\Gamma; \Pi_0; \Pi_1 \vdash\ r@\mathsf{k}\ Q\ (\bar{\mathsf{t}})\ \otimes r@\mathsf{k}\ Q\ (\bar{\mathsf{t}})}\ (OPack_2)$$

  Proof: From the $OPack_2$ rule we know that
  if $\Gamma; \Pi_0; \Pi_1 \vdash^{LL}\ r@\mathsf{k}\ Q\ (\bar{\mathsf{t}})\ \otimes r@\mathsf{k}\ Q\ (\bar{\mathsf{t}})$ then $\Gamma; \Pi_0; \Pi_1 \vdash^{LL}\ r@\mathsf{k*2}\ Q\ (\bar{\mathsf{t}})$.
  Using the definition of the translateAnd() function from Figure 4.18, we obtain that trans(r@k Q ($\bar{\mathsf{t}}$) $\otimes r$@k Q ($\bar{\mathsf{t}}$)) is equal to trans(r@(k+k) Q ($\bar{\mathsf{t}}$)), which is equal to trans(r@(k*2) Q ($\bar{\mathsf{t}}$)).
  We now have to prove that trans($\Gamma; \Pi_0; \Pi_1$) $\vdash$ trans(r@k*2 Q ($\bar{\mathsf{t}}$)).
  Using the induction hypothesis we obtain that trans($\Gamma; \Pi_0; \Pi_1$) $\vdash^{FOL}$ trans(r@k*2 Q ($\bar{\mathsf{t}}$)), which is exactly what we needed to prove. Thus if P holds in LL then trans(P) holds in FOL. Q.E.D.

- **Case** $OUnpack_1$

  To prove: If $\Gamma; \Pi_0; \Pi_1 \vdash^{LL}$ P then trans($\Gamma; \Pi_0; \Pi_1$) $\vdash^{FOL}$ trans(P) where:

  1. P = `unpacked(`r@k Q ($\bar{\mathsf{t}}$) `)`
  2. trans(P) = `(packedQ[r]==false)` &&
     translateObjectProposition(`r@k Q(`$\bar{t}$`)`) and
  3. $$\frac{\Gamma; \Pi_0; \Pi_1 \vdash\ r@\mathsf{k/2}\ Q\ (\bar{\mathsf{t}})\ \otimes \mathtt{unpacked}(r@\mathsf{k/2}\ Q\ (\bar{\mathsf{t}})\ )}{\Gamma; \Pi_0; \Pi_1 \vdash\ \mathtt{unpacked}(r@\mathsf{k}\ Q\ (\bar{\mathsf{t}})\ )}\ (OUnpack_1)$$

  Proof:
  From the $OUnpack_1$ rule we know that
  if $\Gamma; \Pi_0; \Pi_1 \vdash^{LL}$ `unpacked(`r@k Q ($\bar{\mathsf{t}}$) `)` then
  $\Gamma; \Pi_0; \Pi_1 \vdash^{LL}\ r$@k/2 Q ($\bar{\mathsf{t}}$) $\otimes$ `unpacked(`r@k/2 Q ($\bar{\mathsf{t}}$) `)` .
  Using the induction hypothesis we obtain that trans($\Gamma; \Pi_0; \Pi_1$) $\vdash^{FOL}$ trans(r@k/2 Q ($\bar{\mathsf{t}}$) $\otimes$ `unpacked(`r@k/2 Q ($\bar{\mathsf{t}}$) `)`)). Using the definition of the translateAnd() function from Figure 4.18, we obtain that trans(r@k/2 Q ($\bar{\mathsf{t}}$) $\otimes$ `unpacked(`r@k/2 Q ($\bar{\mathsf{t}}$) `)`) is equal to trans(`unpacked(`r@k Q ($\bar{\mathsf{t}}$) `)`). This means that trans($\Gamma; \Pi_0; \Pi_1$) $\vdash^{FOL}$ trans(`unpacked(`r@k Q ($\bar{\mathsf{t}}$) `)`).
  Thus if P holds in LL then trans(P) holds in FOL. Q.E.D.

- **Case** $\otimes$

  To prove: If $\Gamma; \Pi_0; \Pi_1 \vdash^{LL}$ R then trans($\Gamma; \Pi_0; \Pi_1$) $\vdash^{FOL}$ trans(R) where:

  1. R = R1 $\otimes$ R2
  2. trans(R) = translateAnd(R1,R2)
  3. $$\frac{\Gamma; \Pi_0; \Pi_1 \vdash\ \mathsf{R1} \quad \Gamma; \Pi_0; \Pi_1' \vdash\ \mathsf{R2}}{\Gamma; \Pi_0; \Pi_1, \Pi_1' \vdash\ \mathsf{R1} \otimes \mathsf{R2}}\ (\otimes)$$

  Proof:
  From the $\otimes$ rule we know that $\Gamma; \Pi_0; \Pi_1 \vdash$ R1 $\quad \Gamma; \Pi_0; \Pi_1' \vdash$ R2 . By induction and using the translation rules we know that trans($\Gamma; \Pi_0; \Pi_1$) $\vdash$ trans(R1) and trans($\Gamma; \Pi_0; \Pi_1'$) $\vdash$ trans(R2). Using the rules of FOL we obtain that trans($\Gamma; \Pi_0; \Pi_1, \Pi_1'$) $\vdash$ trans(R1) && trans(R2). If R1 and R2 both represent object propositions about the same reference and same predicate (also the same parameters for that predicate), the fractions of these two

object propositions will be added. This is the point in the proof where we need to know the aliasing relations in R1 and R2, i.e., which references present in R1 and R2 point to the same object. If R1 and R2 are general formulas they can be rewritten in their disjunctive normal form and only then the corresponding object propositions in each conjunction will have their fractions added. Thus the formula trans(R1) && trans(R2) is equivalent to the formula translateAnd(R1,R2), except that in translateAnd(R1,R2) the fractions are added. But we know that $\Pi_1 \vdash$ R1 and $\Pi_1' \vdash$ R2, which means that the fractions related to R1 are in Π_1 and the fractions related to R2 are in Π_1'. When Π_1 and Π_1' are put together in Π_1, Π_1', the fractions of the object propositions are added, which means that Π_1, Π_1' will contained the summed fractions necessary to prove translateAnd(R1,R2). So if R holds in LL then trans(R) holds in FOL. Q.E.D.

- **Case** $\oplus_L$
  To prove: If $\Gamma; \Pi_0; \Pi_1 \vdash^{LL}$ R then trans($\Gamma; \Pi_0; \Pi_1$) $\vdash^{FOL}$ trans(R) where:
  1. R = R1 $\oplus$ R2
  2. trans(R) = trans(R1) || trans(R2)
  3. $\dfrac{\Gamma; \Pi_0; \Pi_1 \vdash \text{ R1}}{\Gamma; \Pi_0; \Pi_1 \vdash \text{ R1} \oplus \text{R2}} \ (\oplus_L)$

  Proof: If the additive disjunction R1 $\oplus$ R2 holds in LL then R1 holds in LL, from the ($\oplus_L$) rule. Using the induction hypothesis, we obtain that trans($\Gamma; \Pi_0; \Pi_1$) $\vdash^{FOL}$ trans(R1). Using the || rule from FOL we obtain that trans($\Gamma; \Pi_0; \Pi_1$) $\vdash^{FOL}$ trans(R1) || trans(R2). This is exactly what we needed to prove. Q.E.D.
- **Case** $\oplus_R$
  To prove: If $\Gamma; \Pi_0; \Pi_1 \vdash^{LL}$ R then trans($\Gamma; \Pi_0; \Pi_1$) $\vdash^{FOL}$ trans(R) where:
  1. R = R1 $\oplus$ R2
  2. trans(R) = trans(R1) || trans(R2)
  3. $\dfrac{\Gamma; \Pi_0; \Pi_1 \vdash \text{ R2}}{\Gamma; \Pi_0; \Pi_1 \vdash \text{ R1} \oplus \text{R2}} \ (\oplus_R)$

  Proof: If the additive disjunction holds in the linear theory then the translation into the first order logic disjunction holds. The proof in this case is very similar to the $\oplus_L$ case.
- **Case** $\exists_1 R$
  To prove: If $\Gamma; \Pi_0; \Pi_1 \vdash^{LL}$ R then trans($\Gamma; \Pi_0; \Pi_1$) $\vdash^{FOL}$ trans(R) where:
  1. R = ∃t:T.R
  2. trans(R) = trans(R1); `addExistential(`trans(R1), t:trans(T)`)`
  3. $\dfrac{\Gamma \vdash \text{ M:T} \quad \Gamma; \Pi_0; \Pi_1 \vdash \text{ R1\{M/t\}}}{\Gamma; \Pi_0; \Pi_1 \vdash \ \exists\text{t:T.R1}} \ (\exists_1 R)$

  Proof: Whenever there is an existential formula in our linear theory, the way we translate it in first order logic in Boogie is by adding the existentially quantified variable to the variables of the enclosing method or predicate where R1 is found. When that method or the packing or unpacking procedures of that predicate are called, the programmer will have to give a witness for the variable t. This is because the performance of the Boogie tool is extremely poor when it has to instantiate an existentially quantified variable on its own.
  If the existential formula is true in the linear theory then according to the rule $\exists_1 R$, there exists a witness M:T for it. By induction, there exists a witness trans(M):trans(T) for the

translated formula in FOL. The statement of the Completeness Theorem assumes that the witness is chosen correctly by the programmer. If the witness is chosen incorrectly by the programmer, the formula will be true in the linear theory but not in first order logic. By applying the existential rule in FOL, we obtain that if the existentially quantified translated first order logic formula is true then the existentially quantified linear theory formula is true.

- **Case** $\exists_1 L$

  To prove: If $\Gamma;\Pi_0;\Pi_1 \vdash^{LL}$ R then trans($\Gamma;\Pi_0;\Pi_1$) $\vdash^{FOL}$ trans(R) where:
  1. R = ∃t:T.R
  2. trans(R) = trans(R1); `addExistential`(trans(R1), t:trans(T))
  3. $\dfrac{\Gamma, \text{M:T}; \Pi_0; \Pi_1, \text{R\{M/x\}} \vdash \text{ C}}{\Gamma; \Pi_0; \Pi_1, \exists\text{x:T.R} \vdash \text{ C}}\ (\exists_1 L)$

  Proof: Whenever there is an existential formula in our linear theory, the way we translate it in first order logic in Boogie is by adding the existentially quantified variable to the variables of the enclosing method or predicate where R1 is found. When that method or the packing or unpacking procedures of that predicate are called, the programmer will have to give a witness for the variable t. This is because the performance of the Boogie tool is extremely poor when it has to instantiate an existentially quantified variable on its own.
  If the existential formula is true in the linear theory then according to the rule $\exists_1 L$ C will be true and the same C will be true when we have R{M/x} instead of the existential formula. We can choose the witness M:T for the existential formula ∃x:T.R. The statement of the Completeness Theorem assumes that the witness is chosen correctly by the programmer. If the witness is chosen incorrectly by the programmer, the formula will be true in the linear theory but not in first order logic. By induction, there exists a witness trans(M):trans(T) for the translated formula in FOL. By applying the existential rule in FOL, we obtain that if the existentially quantified translated first order logic formula is true then the existentially quantified linear theory formula is true.

- **Case** $\exists_2 R$

  To prove: If $\Gamma;\Pi_0;\Pi_1 \vdash^{LL}$ R then trans($\Gamma;\Pi_0;\Pi_1$) $\vdash^{FOL}$ trans(R) where:
  1. R= ∃z:double.R1
  2. trans(R) = trans(R1); `addExistentialFrac`(trans(R1), z:trans(double))
  3. $\dfrac{\Gamma \vdash \text{ F:double} \quad \Pi_0 \vdash \text{ F} > 0 \quad \Gamma;\Pi_0;\Pi_1 \vdash \text{ R1\{F/z\}}}{\Gamma;\Pi_0;\Pi_1 \vdash \text{ }\exists\text{z:double.R1}}\ (\exists_2 R)$

  Proof: This case is similar to the one above. If no restriction is given on the fraction z we assume that it can be any fraction that is greater than zero. If R1 uses the fraction z, the translation of R1 will instead state that the value of `frac` for the corresponding predicate and reference is greater than zero, for all the occurrences of z in R1.
  The inductive proof is similar to the proof for case $\exists_1 R$.

- **Case** $\exists_2 L$

  To prove: If $\Gamma;\Pi_0;\Pi_1 \vdash^{LL}$ R then trans($\Gamma;\Pi_0;\Pi_1$) $\vdash^{FOL}$ trans(R) where:
  1. R= ∃z:double.R1
  2. trans(R) = trans(R1); `addExistentialFrac`(trans(R1), z:trans(double))

3. $\dfrac{\Gamma, \textsf{F:double}; \Pi_0, \textsf{F} > 0; \Pi_1, \textsf{R\{F/z\}} \vdash C}{\Gamma; \Pi_0; \Pi_1, \exists \textsf{z:double.R} \vdash C}\ (\exists_2 L)$

Proof: This case is similar to the $\exists_1 L$ case above. If no restriction is given on the fraction z we assume that it can be any fraction that is greater than zero. If R1 uses the fraction z, the translation of R1 will instead state that the value of `frac` for the corresponding predicate and reference is greater than zero, for all the occurrences of z in R1.
The inductive proof is similar to the proof for case $\exists_1 L$.

- **Case** $\exists_3 R$
  To prove: If $\Gamma; \Pi_0; \Pi_1 \vdash^{LL}$ R then trans($\Gamma; \Pi_0; \Pi_1$) $\vdash^{FOL}$ trans(R) where:
  1. R= $\exists$z:double.z binop t $\Rightarrow$ R1
  2. trans(R) = trans(R1);
     `addExistentialFracBinop`(trans(R1), z:trans(double), `z binop t`)
  3. $\dfrac{\Gamma \vdash \textsf{F:double} \quad \Pi_0 \vdash \textsf{F binop t} \quad \Gamma; \Pi_0; \Pi_1 \vdash \textsf{R1\{F/z\}}}{\Gamma; \Pi_0; \Pi_1 \vdash \exists \textsf{z:double.z binop t} \Rightarrow \textsf{R1}}\ (\exists_3 R)$

  Proof: Similarly to the case above, the proof for this case also relies on replacing all the occurrences of z in R1 with a concrete formula, for which we have `z binop t` in this case. As opposed to the case above where we were implicitly assuming that z > 0, here we have a specific condition that has to hold for z: z binop t .
  The inductive proof is similar to the proof for case $\exists_1 R$.

- **Case** $\exists_3 L$
  To prove: If $\Gamma; \Pi_0; \Pi_1 \vdash^{LL}$ R then trans($\Gamma; \Pi_0; \Pi_1$) $\vdash^{FOL}$ trans(R) where:
  1. R= $\exists$z:double.z binop t $\Rightarrow$ R1
  2. trans(R) = trans(R1);
     `addExistentialFracBinop`(trans(R1), z:trans(double), `z binop t`)
  3. $\dfrac{\Gamma, \textsf{F:double}; \Pi_0, \textsf{F binop t}; \Pi_1, \textsf{R\{F/z\}} \vdash C}{\Gamma; \Pi_0; \Pi_1, \exists \textsf{z:double.z binop t} \Rightarrow \textsf{R} \vdash C}\ (\exists_3 L)$

  Proof: Similarly to the $\exists_1 L$ case above, the proof for this case also relies on replacing all the occurrences of z in R1 with a concrete formula, for which we have `z binop t` in this case. As opposed to the case above where we were implicitly assuming that z > 0, here we have a specific condition that has to hold for z: z binop t .
  The inductive proof is similar to the proof for case $\exists_1 L$.

- **Case** $\forall_1 R$
  To prove: If $\Gamma; \Pi_0; \Pi_1 \vdash^{LL}$ R then trans($\Gamma; \Pi_0; \Pi_1$) $\vdash^{FOL}$ trans(R) where:
  1. R = $\forall$t:T.R1
  2. trans(R) $=$ $var\ t : T$; trans(R1)
  3. $\dfrac{\Gamma, \textsf{m:T}; \Pi_0; \Pi_1 \vdash \textsf{R1\{m/t\}}}{\Gamma; \Pi_0; \Pi_1 \vdash \forall \textsf{t:T.R1}}\ (\forall_1 R)$

  Proof: This case together with the following cases use the universal quantifier. The way the universal quantifier is translated into Boogie is by declaring the universally quantified variable as a variable in the program. From the $\forall_1 R$ rule we know that if the formula $\forall$t:T.R1 is true in LL then R1{m/t} will be true, with m being declared of type T. After the declaration of variable t, if the formula R is true then the FOL translation of the formula R1 is true, following our inductive proof. Thus, trans($\Gamma; \Pi_0; \Pi_1$) $\vdash^{FOL}$ trans(R).

- **Case** $\forall_1 L$

  To prove: If $\Gamma; \Pi_0; \Pi_1 \vdash^{LL}$ R then trans($\Gamma; \Pi_0; \Pi_1$) $\vdash^{FOL}$ trans(R) where:
  1. R $=$ ∀t:T.R1
  2. trans(R) $=$ $var\ t : T$; trans(R1)
  3. $\dfrac{\Gamma \vdash \text{m:T}; \Gamma; \Pi_0; \Pi_1, \text{R\{m/x\}} \vdash C}{\Gamma; \Pi_0; \Pi_1, \forall\text{x:T.R} \vdash C}\ (\forall_1 L)$

  Proof: The way the universal quantifier is translated into Boogie is by declaring the universally quantified variable as a variable in the program. Using the $\forall_1 L$ rule we know that when ∀t:T.R1 is true in LL, C will be true and m:T can be obtained from Γ which will make R{m/x} true. By renaming the general variable m to t and following our inductive proof we obtain that after the declaration of variable t, if the formula R is true then the FOL translation of the formula R1 is true. Thus, trans($\Gamma; \Pi_0; \Pi_1$) $\vdash^{FOL}$ trans(R).

- **Case** $\forall_2 R$

  To prove: If $\Gamma; \Pi_0; \Pi_1 \vdash^{LL}$ R then trans($\Gamma; \Pi_0; \Pi_1$) $\vdash^{FOL}$ trans(R) where:
  1. R= ∀z:double.R1
  2. trans(R)= $var\ z : real$; trans(R1)
  3. $\dfrac{\Gamma, \text{f:double}; \Pi_0, \text{f > 0}; \Pi_1 \vdash \text{R\{f/z\}}}{\Gamma; \Pi_0; \Pi_1 \vdash \forall\text{z:double.R}}\ (\forall_2 R)$

  Proof: This case is very similar to the $\forall_1 R$ case above, but instead of the variable t we have the variable z that designates fractions. The only difference is that for a fraction, we declare it as having type $real$ and we assume that it is greater than zero. If the formula R is true then the translation of the inner formula R1 is true.
  The inductive proof is similar to the proof for case $\forall_1 R$.

- **Case** $\forall_2 L$

  To prove: If $\Gamma; \Pi_0; \Pi_1 \vdash^{LL}$ R then trans($\Gamma; \Pi_0; \Pi_1$) $\vdash^{FOL}$ trans(R) where:
  1. R= ∀z:double.R1
  2. trans(R)= $var\ z : real$; trans(R1)
  3. $\dfrac{\Gamma \vdash \text{f:double}; \Gamma, \Pi_0, \text{f > 0}; \Pi_1, \text{R\{f/z\}} \vdash C}{\Gamma; \Pi_0; \Pi_1, \forall\text{z:double.R} \vdash C}\ (\forall_2 L)$

  Proof: This case is very similar to the $\forall_1 R$ case above, but instead of the variable t we have the variable z that designates fractions. The only difference is that for a fraction, we declare it as having type $real$ and we assume that it is greater than zero. If the formula R is true then the translation of the inner formula R1 is true.
  The inductive proof is similar to the proof for case $\forall_1 R$.

- **Case** $\forall_3 R$

  To prove: If $\Gamma; \Pi_0; \Pi_1 \vdash^{LL}$ R then trans($\Gamma; \Pi_0; \Pi_1$) $\vdash^{FOL}$ trans(R) where:
  1. R= ∀z:double.z binop t ⇒ R1
  2. trans(R)= $var\ z : real$; z binop t ==> trans(R1)
  3. $\dfrac{\Gamma, \text{f:double}; \Pi_0, \text{f binop t}; \Pi_1 \vdash \text{R1\{f/z\}}}{\Gamma; \Pi_0; \Pi_1 \vdash \forall\text{z:double.z binop t} \Rightarrow R1}\ (\forall_3 R)$

  Proof: This case is very similar to the previous cases $\forall_1 R$ and $\forall_2 R$, the only difference being that now we have an additional condition on the fraction z, which is z binop t.
  The inductive proof is similar to the proof for case $\forall_1 R$.

- **Case** $\forall_3 L$
  To prove: If $\Gamma;\Pi_0;\Pi_1 \vdash^{LL}$ R then trans($\Gamma;\Pi_0;\Pi_1$) $\vdash^{FOL}$ trans(R) where:
  1. R= $\forall$z:double.z binop t $\Rightarrow$ R1
  2. trans(R)= $var\ z : real;$ z binop t ==> trans(R1)
  3. $\dfrac{\Gamma \vdash \text{ f:double}; \Gamma, \Pi_0, \text{f binop t}; \Pi_1 \text{R\{f/z\}} \vdash C}{\Gamma;\Pi_0;\Pi_1, \forall\text{z:double.z binop t} \Rightarrow R \vdash C}\ (\forall_3 L)$

  Proof: This case is very similar to the previous cases $\forall_1 L$ and $\forall_2 L$, the only difference being that now we have an additional condition on the fraction z, which is z binop t. The inductive proof is similar to the proof for case $\forall_1 L$.
- **Case** $tbint$
  To prove: If $\Gamma;\Pi_0;\Pi_1 \vdash^{LL}$ R then trans($\Gamma;\Pi_0;\Pi_1$) $\vdash^{FOL}$ trans(R) where:
  1. R = t binop t $\Rightarrow$ R1
  2. trans(R) = t binop t ==> trans(R1)
  3. $\dfrac{\Gamma;\Pi_0;\Pi_1 \vdash \text{ t1 binop t2} \Rightarrow \text{R1}}{\Gamma;\Pi_0, \text{t1 binop t2}; \Pi_1 \vdash \text{ R1}}\ (tbint)$

  Proof: If the linear logic implication will be true then the first order logic implication will be true. The reasoning in this case is straightforward because on the left side of the implication we have very simple formulas that do not require a complicated translation. More importantly, the formulas on the left are not using resources and thus we do not have to worry about the left side using up a resource that the right side needs.
- **Case** id
  To prove: If $\Gamma;\Pi_0;\Pi_1 \vdash^{LL}$ R then trans($\Gamma;\Pi_0;\Pi_1$) $\vdash^{FOL}$ trans(R) where:
  1. $\dfrac{}{\Gamma;\Pi_0;\Pi_1, \text{R} \vdash \text{ R}}\ (id)$

  Proof: From the (id) rule we know that $\Gamma;\Pi_0;\Pi_1$, R $\vdash$ R and we need to prove that trans($\Gamma;\Pi_0;\Pi_1$, R) $\vdash$ R, i.e., we need to prove that trans($\Gamma;\Pi_0$);trans(Π_1),trans(R) $\vdash$ trans(R), which holds according to the identity rule from FOL.

Note that the main caveat to completeness (other than the need for witnesses, which is a theorem prover limitation) is aliasing. Although the point of linear logic is to signal aliasing by having an object proposition with a fraction less than 1, in practice there is a need to specify if two references are pointing to the same object. This relaxes our system: two object propositions, with fractions less than 1, can coexist in the same formula. This can be advantageous, but may result in incompleteness in the prover if the aliasing relationships are not properly specified.

□

4.6 Informal Soundness Argument

In the following paragraphs we give an intuitive explanation of why our translation is sound beyond the formulas. Throughout the explanation, we use the Composite example as a reference.

When we have an object proposition containing a fraction less than 1 inside the definition of a predicate, we translate it as a `requires` statement in the corresponding `Pack` procedure, and as an `ensures` statement in the `Unpack` procedure. For example, we have the object proposition `ol#0.5 count(lc)` inside the definition of the predicate `left` from Section 5.4, Figure 5.39. Moreover, the object proposition is preceded by the condition `ol != null`. In this case our translation will be `(left[this] != null) ==> ((fracCount[ol] >= 0.5) && (count[ol] == lc))`, preceded either by the `requires` or `ensures` keyword, depending if we are inside the `Pack` procedure or in the `Unpack` procedure. Note that the translation does not have the bit `packedCount[ol]` in it. This is because the fraction is less than 1 and we cannot be sure that at this point in time the inner object proposition is packed. We do know for sure that we have a fraction of at least 0.5 to it. Our theory ensures us that although the inner object proposition might not be packed at this moment, it will be packed at some point in the future, most likely the boundary of methods.

As opposed to the body of predicates where we do not mention `packedPred` when referring to a packed object proposition, we do mention whether a predicate is `packed` or `unpacked` in the pre- and post-conditions of methods. This is because the boundaries of methods are places where we should know for sure whether an object proposition is packed or not.

Our theory states that in the precondition of a method, all the unpacked object propositions need to be mentioned explicitly. All the others are assumed to be packed. This holds only for the beginning of a procedure; in the body of the procedure some object propositions can be unpacked and then packed back again to the same predicate; this is why we are writing in the first paragraph above that we cannot always be sure that an object proposition is packed. In order to infuse the Boogie translation with this knowledge, we have to write what we call `requires forall` statements. These are statements universally quantified, such as the following `requires (forall x:Ref :: ((x!=this) && (x!=op) ==> packedCount[x]))`, taken as an example from the preconditions of the procedure `updateCount`. These `requires forall` statements are generated automatically by looking at the preconditions that the programmer specifies.

The `modifies` statement that Boogie requires of each method is very strong: for the global maps representing the fields in a class, these `modifies` statements act as `havoc` statements, thus removing all prior knowledge of the global maps that we had before calling the respective method. This is why we need statements that we call `ensures forall`, generated automatically as postconditions to methods. Similarly to the `requires forall` statements, they are generated by looking only at the pre- and postconditions that the programmer specified for that specific method, with a single exception: we do need to differentiate between methods that call other methods in their bodies or not. The specific rules that we follow to infer the `ensures forall` statements are straightforward.

To gather the information to generate them, we only look at pre- and post-conditions. We only try to infer the `ensures forall` for the `packed` or `frac` maps if there is a mention of them in the postcondition. More specifically, for all the objects `obj` for which there was a `packedQ[obj]` in the precondition, `packedQ[obj]` must also be mentioned in the postcondition. If `packedQ[obj]` is not mentioned in the postcondition for all objects `obj` for which `packedQ[obj]` was mentioned in the precondition, it means that the programmer does not know what happens to `packedQ` for certain objects at the end of this method. The same reasoning applies to the global variable `frac`. In these cases we cannot infer the `ensures forall` statement. As an insight, the programmer would have to be able to write all the `ensures forall` statements manually. If they don't know what happens to a particular packedQ[obj], they would not be able to write the ensures forall. One can see that if the programmer is not able to write the ensures forall statement, our Oprop tool should not be able to infer this statement.

There are two kinds of ensures forall that can be inferred for fractions:

ensures (forall y:Ref :: ((y!=this) ==> (fracLeft[y] == old(fracLeft[y])))) and

ensures (forall y:Ref :: (old(fracParent[y]) > 0.0) ==> (fracParent[y] > 0.0)).

The second one is more relaxed. If there are calls to any methods in the body of this method, then we infer the second ensures forall. To generate this ensures forall we check the specifications of the method to get the objects for which the fractions were > 0.0 in the precondition but ≥ 0.0 in the postcondition. Note that the user can specify a fraction to be ≥ 0.0 in the postcondition if he knows that all that fraction was consumed in the body of this method. That could happen if a method that is called inside the body uses up all the fraction that was specified in the object proposition in the precondition.

The first one can be inferred for methods that have no calls to other methods inside their bodies. We gather the objects that were not modified inside the body of this method and only for those objects we ensure that the old fraction which was given in the precondition is equal to the fraction that we have in the postcondition. Note that the generation of the ensures forall statement can be made more insightful by parsing the body of the method and seeing if there are cases when even though a fraction was modified, the resulting fraction is the same as the fraction that we got in the precondition. This optimization of the implementation is left as future work.

We do need to get minimal help from parsing the body of a method to infer these ensures forall related to fractions for that method and this is a limitation of our Oprop tool.

For the ensures forall statements for packed, if there is an unpacked object proposition in the postcondition then we infer the ensures forall that compares the value of the map (whether that map is packed or frac) to the old value, i.e., the value of the variable in the beginning of the method. If there is no unpacked object proposition in the postcondition then we infer the ensures forall statement that is similar to ensures forall y:: packedPred[y].

After calling UnpackPred or PackPred in the body of a method, we write the fraction manipulation statements corresponding to each one. These fraction manipulations are either additions or subtractions from the current values of the fractions that are mentioned in the body of the predicates that are inside the predicates that are being packed or unpacked. The Boogie semantics does not allowed us to use the keyword old to infer ensures or requires for the procedures UnpackPred or PackPred because these procedures do not have a body or a modifies clause. That is why we have to write fraction manipulations after invoking these procedures, but we do not have to use fraction manipulations after invoking other methods (actual methods that the programmer wrote and we translated into Boogie).

Chapter 5

Validation and Evaluation

The purpose of this chapter is to show that the object proposition methodology can be used in the formal verification of general design patterns, thus highlighting the maturity and generality of our approach. We evaluate the usefulness of object propositions by applying them to verify a number of small Java programs and also a few more complex programs implementing the following design patterns: observer, state, proxy, flyweight and composite. The names of the small programs are *SimpleCell*, *Link*, *DoubleCount* and *Share*; we describe them in detail later in this chapter. The purpose of the small examples is to illustrate how our methodology handles particular verification idioms.

So far, we have described in Section 2.2 how we manually verified an instance of the observer pattern and in Section 5.4 we are going to show the verification of an instance of the composite pattern, that we have automatically verified using the Oprop tool. In this section we describe the verification of three instances of the state design pattern [9], proxy [8] and flyweight pattern [3], thus showing that object propositions can be used to verify a variety of design patterns. Note that for these three design patterns, we have manually translated the programs and their Oprop annotations into the Boogie language, and then verified the translated programs in the Boogie tool. While we have implemented all the features necessary for our Oprop tool to automatically verify an instance of the composite design pattern, our tool does not implement all the necessary features to automatically verify these last three design patterns, such as inheritance, base classes or derived classes. These patterns are slightly different than the composite pattern, and while it would not be very difficult to implement these features in Oprop, we leave this as future work.

We chose to verify these design patterns because they represent general coding patterns and this is the first step in showing that our methodology is general. We chose the state pattern because it shows how object propositions can be used for the verification of objects whose internal state changes. Verifying programs showcasing this kind of rapidly changing states highlights the differences between object propositions and the classical invariant technique. By verifying the state pattern, we also demonstrate how we can verify programs containing interfaces and their implementations. By adding this capability to our system, we would be able to verify other programs that use interfaces. Many improvements can be brought to the Oprop tool that would allow the formal verification of very diverse design patterns and programs, but we leave that as future work.

The proxy design pattern is widely used, in a variety of situations (to add access control to an existing object, to provide an interface for remote resources, to coordinate expensive operations on remote resources, etc.). By verifying an instance of this pattern, we are able to say that our methodology can be used for the verification of many programs, which indicates the applicability of our approach to many practical programs.

The similarity between the state and proxy patterns reinforces the idea that some patterns resemble each other, with small variations. For example, the strategy pattern is also similar to the state pattern. There is also a connection between the proxy and flyweight patterns: in situations where multiple copies of a complex object must exist, the proxy pattern can be adapted to incorporate the flyweight pattern to implement a reference counting pointer object. To better understand this idea, consider a complex object and a proxy object that accesses the complex object, as an instance of the proxy design pattern. If there is a need for multiple copies of the complex object, we can add the flyweight pattern to have multiple references to the complex object, instead of making actual copies of the complex object. The three design patterns that we chose show how our work improves the state of the art in formal verification. When choosing to verify the state pattern, the proxy and the flyweight patterns, we stayed away from patterns that were using inheritance. The support for inheritance in the object propositions methodology is left as future work.

By analyzing these three design patterns, which are not too simple and not too complex,we show the differences between our approach and previous ones. It was interesting to verify the design patterns because it gave us more insight into how the theory of object propositions can be extended to incorporate more features of the Java language and it gave us the opportunity to apply our verification system to creative instances of the design patterns.

5.1 Verifying an Instance of the State Pattern

The state pattern allows an object to alter its behavior when its internal state changes. When this happens, the object will appear to change its class. Programs that include instances of this pattern have an interface that is implemented by different classes, representing the different states. There is also a context object that acts as a controller and whose behavior varies as its state object changes.

5.1.1 Example

In Figure 5.1, inspired from the Wikipedia figure [11], we present the UML diagram of the general state pattern . There is a *State* interface and two classes *ConcreteStateA* and *ConcreteStateB* that implement that interface. The *Context* class acts as a controller which has a *request()* method in which it invokes the method *handle()* of its field *state*. Depending on the actual class of the *state* field, the method *handle()* of class *ConcreteStateA* is called or the method *handle()* from class *ConcreteStateB*. The class *StateClient* instantiates a field *s* of type *Context*. In the formal verification of our instance of the state pattern, we check that the properties about *s* stated in the client code hold. The state pattern implements a state machine using the features of object oriented code and shows how behavior can vary for the same object based on its internal state. The state pattern is important because it can be used by an object to change its behavior at runtime without resorting to large monolithic conditional or *switch* statements and thus improve maintainability.

In the remainder of this subsection we give an intuitive explanation of our instance of the state pattern, we present the annotated Oprop classes that make up our example and for each class we detail the more interesting parts.

The Oprop classes used in the state pattern example are: IntCell in Figure 5.3, interface Statelike in Figure 5.4, StateLive in Figures 5.5 and 5.6, StateSleep in Figures 5.7 and 5.8, StateLimbo in Figures 5.9 and 5.10, StateContext in Figures 5.11 and 5.12 and StateClient in Figures 5.13, 5.14 and 5.15. One can think of this system as a machine controller accepting requests during the day, while being in the StateLive state, transitioning for a short while to the StateLimbo state and finally being in the StateSleep state during the night when it is not accepting requests. The interface Statelike represents objects that have an integer field that is a multiple of 3, or a multiple of 2, while the actual classes StateLive, StateLimbo and StateSleep implement their own versions of the predicates in the interface Statelike. An object of type Statelike can be in the three concrete states StateLive, StateLimbo or StateSleep, and it can switch back and forth between these three states.

The way this example works is that when a method is called on an object of type `StateContext`, the call is forwarded to the `Statelike` object, which can actually be of type `StateLive, StateSleep` or `StateLimbo`.
In addition to doing the computation specified in `computeResult()` or `computeResult2()` - depending on which method of `StateContext` was called - the context is also updated with the next state that the `Statelike` object should have.

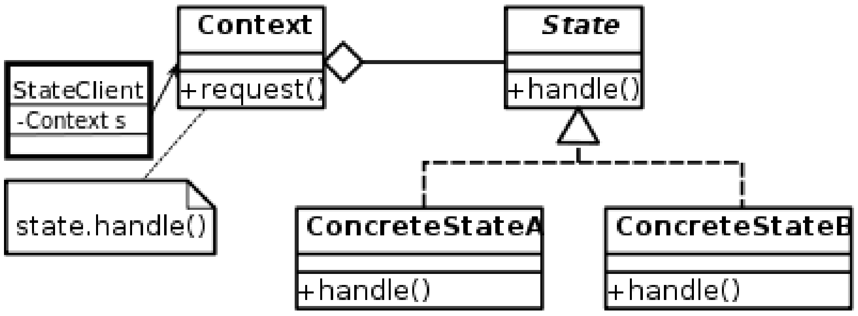

Figure 5.1: UML Diagram of State Pattern

In Figure 5.2 we show the UML diagram of the classes in our example implementing the state design pattern. The interface `Statelike` declares two methods `computeResult` and `computeResult2`, which not only calculate a result, but are also the ones to make the transitions between the `Live`, `Limbo` and `Sleep` of a `Statelike` object. The classes `StateLive`, `StateLimbo` and `StateSleep` implement the interface `Statelike` and they represent each of the three states that a machine controlled Statelike object can be in. Object propositions - by modeling the properties that an object can satisfy and the idea that even though an object is shared, it can be modified as long as an invariant property is satisfied - are perfect for verifiying programs implementing the state design pattern. The `main1()` and `main2()` methods can be found in the class `StateClient` and they show the ways in which the verification us-

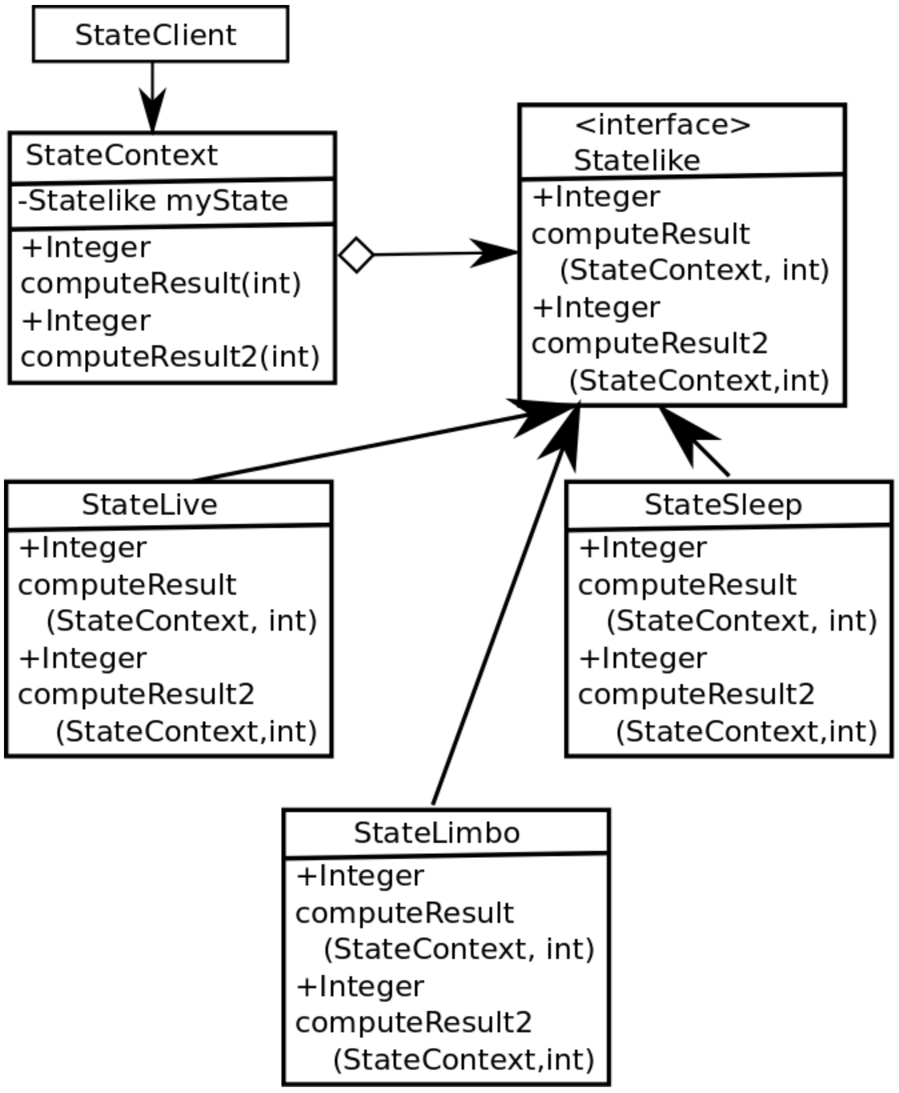
StateClient
StateContext
-Statelike myState
+Integer computeResult(int)
+Integer computeResult2(int)
<interface> Statelike
+Integer computeResult (StateContext, int)
+Integer computeResult2 (StateContext,int)
StateLive
+Integer computeResult (StateContext, int)
+Integer computeResult2 (StateContext,int)
StateSleep
+Integer computeResult (StateContext, int)
+Integer computeResult2 (StateContext,int)
StateLimbo
+Integer computeResult (StateContext, int)
+Integer computeResult2 (StateContext,int)

Figure 5.2: UML Diagram of Our Example of State Pattern

ing object propositions is different than the classical invariants verification or than verification done using separation logic. When the method `computeResult()` is called on an object of type `StateContext`, it is forwarded to the `Statelike` object which is a field of the class `StateContext`.

The *IntCell* class 5.3 has two fields *divider* and *value*. The predicate *BasicIntCell* is there merely to give us access to the two fields without imposing any properties on them. The predicate *MultipleOf* is used to ensure that the field *value* is a multiple of *divider* and it is used in our *State* design pattern example.

The *Statelike* interface 5.4 defines the predicates *StateMultipleOf3* and *StateMultipleOf2* that need to be defined in each of the classes implementing this interface, as well as the methods *computeResult()*, *computeResult2()*, *checkMod3()* and *checkMod2()* that need to be implemented by classes *StateLive* , *StateLimbo* and *StateSleep*. Moreover the annotations of the methods

```
class IntCell {
  int divider;
  int value;

  predicate BasicIntCell(int val, int divi) =
      this.divider -> divi && this.value -> val

  predicate MultipleOf(int a) = exists int v :
      this.divider -> a && this.value -> v &&
      ( (v - int(v/a)*a ) == 0 )

  IntCell(int divider1, int value1)
    ensures this#1.0 BasicIntCell(value1, divider1)
  {
    this.value  = value1;
    this.divider = divider1;
  }

  int getValueInt()
  ~double k:
  int v, int d:
    requires this#k BasicIntCell(v, d)
    ensures this#k BasicIntCell(v, d)
  {
    unpack(this#k BasicIntCell(v, d));
    int temp = this.value;
    pack(this#k BasicIntCell(v, d));
    return temp;
  }
}
```

Figure 5.3: State pattern example - IntCell class

```
interface Statelike {
  predicate StateMultipleOf3();
  predicate StateMultipleOf2();

  Statelike computeResult(StateContext context, int num);
  ~double k, k2:
    requires (context#k stateContextMultiple3())
    ensures (context#k stateContextMultiple3())

  Statelike computeResult2(StateContext context, int num);
  ~double k, k2:
    requires (context#k stateContextMultiple2())
    ensures (context#k stateContextMultiple2())

  boolean checkMod3();
  ~double k:
    requires this#k StateMultipleOf3()
    ensures this#k StateMultipleOf3()

boolean checkMod2();
  ~double k:
    requires this#k StateMultipleOf2()
    ensures this#k StateMultipleOf2()
}
```

Figure 5.4: State pattern example - Statelike interface

defined in the interface *Statelike* need to be implied by the annotations of the methods with the same names in the implementing classes. The methods in the implementing classes can have stronger annotations but they need to be at least as strong as the corresponding annotations from the interface.

The class *StateLive* 5.5 implements the predicates *StateMultipleOf3* and *StateMultipleOf2* in a concrete manner: whenever the field in an object of type *StateLive* should be a multiple of 3, it will be a multiple of 21 (which is indeed a multiple of 3). Similarly, whenever the field in an object of type *StateLive* should be a multiple of 2, it will be a multiple of 4. The function *computeResult()* in class *StateLive* has the same precondition as method *computeResult()* from interface *Statelike*, as seen on line 23 in Figure 5.5. Class *StateLive* represents one of the three states that the machine controlled object can be in. The postcondition of the same function in class *StateLive* ensures more properties than the postcondition written for the function in interface *Statelike*, as seen on lines 24-25 in Figure 5.5. It is important for the postconditions of the methods in the classes implementing the methods declared in the interface *Statelike* to ensure at least the properties written in the postconditions of those methods in the interface. The function *computeResult2()* is similar conceptually to the function *computeResult()*, the only difference being that it refers to the predicate *StateContextMultiple2* instead of the predicate *StateContext-Multiple3*. The functions *computeResult()* and *computeResult2()* control the state of the object inside the *Context* parameter and they change it to *StateLimbo* and *StateSleep* respectively, according to the change of state that we envisioned for the automaton that this example describes.

Method *checkMod3()* from Figure 5.6 ensures that the caller of this method satisfies the predicate *StateMultipleOf3*, as defined in the class *StateLive*. Similarly, method *checkMod2()* ensures that the caller of this method satisfies the predicate *StateMultipleOf2*, as defined in the class *StateLive*.

Similarly to class *StateLive*, the class *StateSleep* 5.7 implements the predicates *StateMultipleOf3* and *StateMultipleOf2* in a concrete manner: whenever the field in an object of type *StateSleep* should be a multiple of 3, it will be a multiple of 15 (which is indeed a multiple of 3). Similarly, whenever the field in an object of type *StateSleep* should be a multiple of 2, it will be a multiple of 16. Class *StateSleep* represents one of the three states that the machine controlled object can be in. The function *computeResult()* in class *StateSleep* also has the same precondition as method *computeResult()* from interface *Statelike*, as seen on line 23 in Figure 5.7. The postcondition of the same function in class *StateSleep* ensures more properties than the postcondition written for the function in interface *Statelike*, as seen on line 24 in Figure 5.7. The function *computeResult2()* is similar conceptually to the function *computeResult()*, the only difference being that it refers to the predicate *StateContextMultiple2* instead of the predicate *StateContextMultiple3*. The functions *computeResult()* and *computeResult2()* control the state of the object inside the *Context* parameter and they change it to *StateLive* and *StateLimbo* respectively, according to the change of state that we envisioned for the automaton that this example describes.

Like classes *StateLive* and *StateSleep*, the class *StateLimbo* 5.9 implements the predicates *StateMultipleOf3* and *StateMultipleOf2* in a concrete manner: whenever the field in an object of type *StateLimbo* should be a multiple of 3, it will be a multiple of 33 (which is indeed a multiple of 3). Similarly, whenever the field in an object of type *StateLimbo* should be a multiple of 2, it will be a multiple of 14. Class *StateLimbo* represents another state that the machine controlled object can be in. The function *computeResult()* in class *StateLimbo* also has the same

```
class StateLive implements Statelike {
  IntCell cell;

  predicate StateMultipleOf3() = exists IntCell c, double k :
      this.cell -> c && (c#k MultipleOf(21))
  predicate StateMultipleOf2() = exists IntCell c, double k :
      this.cell -> c && (c#k MultipleOf(4))

  StateLive()
  {
    IntCell temp = new IntCell(0);
    this = new StateLive(temp);
  }

  StateLive(IntCell c)
  ensures this.cell == c;
  {
    this.cell = c;
  }

  Statelike computeResult(StateContext context, int num)
  ~double k, k2:
    requires (context#k StateContextMultiple3())
    ensures (context#k StateContextMultiple3())
  {
    IntCell i1 = new IntCell(MultipleOf(33)[num*33])(33, num*33);
    StateLike r = new StateLimbo(StateMultipleOf3()[i1])(i1);
    context.setState3(r);
    return r;
  }

  Statelike computeResult2(StateContext context, int num)
  ~double k, k2:
    requires (context#k StateContextMultiple2())
    ensures (context#k StateContextMultiple2())
  {
    IntCell i1 = new IntCell(MultipleOf(16)[num*16])(16, num*16);
    StateLike r = new StateSleep(StateMultipleOf2()[i1])(i1);
    context.setState2(r);
    return r;
  }
```

Figure 5.5: State pattern example - Statelive class

```
  boolean checkMod3()
  ~double k:
    requires this#k StateMultipleOf3()
    ensures this#k StateMultipleOf3()
  {
unpack(this#k StateMultipleOf3());
return (this.cell.getValueInt() % 3 == 0);
pack(this#k StateMultipleOf3());
  }

  boolean checkMod2()
  ~double k:
    requires this#k StateMultipleOf2()
    ensures this#k StateMultipleOf2()
  {
    unpack(this#k StateMultipleOf2());
    boolean temp = (this.cell.getValueInt() % 2 == 0);
    pack(this#k StateMultipleOf2());
    return temp;
  }
}
```

Figure 5.6: State pattern example - StateLive class continuation

```
class StateSleep implements Statelike {
  IntCell cell;

  predicate StateMultipleOf3() = exists IntCell c, double k :
      this.cell -> c && (c#k MultipleOf(15))
  predicate StateMultipleOf2() = exists IntCell c, double k :
      this.cell -> c && (c#k MultipleOf(16))

  StateSleep()
  {
    IntCell temp = new IntCell(0);
    this.cell = temp;
  }

  StateSleep(IntCell c)
  ensures this.cell == c;
  {
    this.cell = c;
  }

  Statelike computeResult(StateContext context, int num)
  ~double k, k2:
    requires (context#k StateContextMultiple3())
    ensures (context#k StateContextMultiple3())
  {
    IntCell i1 = new IntCell(MultipleOf(21)[num*21])(21, num*21);
    StateLike r = new StateLive(StateMultipleOf3()[i1])(i1);
    context.setState3(s);
    return r;
  }

  Statelike computeResult2(StateContext context, int num)
  ~double k, k2:
    requires (context#k StateContextMultiple2())
    ensures (context#k StateContextMultiple2())
  {
    IntCell i1 = new IntCell(MultipleOf(14)[num*14])(14, num*14);
    StateLike r = new StateLimbo(StateMultipleOf2()[i1])(i1);
    context.setState2(r);
    return r;
  }
```

Figure 5.7: State pattern example - StateSleep class

```
  boolean checkMod3()
  ~double k:
    requires this#k StateMultipleOf3()
    ensures this#k StateMultipleOf3()
  {
    unpack(this#k StateMultipleOf3());
    boolean temp = (this.cell.getValueInt() % 3 == 0);
    pack(this#k StateMultipleOf3());
    return temp;
  }

  boolean checkMod2()
  ~double k:
    requires this#k StateMultipleOf2()
    ensures this#k StateMultipleOf2()
  {
    unpack(this#k StateMultipleOf2());
    boolean temp = (this.cell.getValueInt() % 2 == 0);
    pack(this#k StateMultipleOf2());
    return temp;
  }
}
```

Figure 5.8: State pattern example - StateSleep class continuation

```
class StateLimbo implements Statelike {
  IntCell cell;

  predicate StateMultipleOf3() = exists IntCell c, double k :
      this.cell -> c && (c#k MultipleOf(33))
  predicate StateMultipleOf2() = exists IntCell c, double k :
      this.cell -> c && (c#k MultipleOf(14))

  StateLimbo()
  {
    IntCell temp = new IntCell(0);
    this.cell = temp;
  }

  StateLimbo(IntCell c)
  ensures this.cell == c;
  {
    this.cell = c;
  }

  Statelike computeResult(StateContext context, int num)
  ~double k, k2:
    requires (context#k stateContextMultiple3())
    ensures (context#k stateContextMultiple3())
  {
    IntCell i1 = new IntCell(MultipleOf(15)[num*15])(15, num*15);
    StateLike r = new StateSleep(StateMultipleOf3()[i1])(i1);
    context.setState3(s);
    return r;
  }

  Statelike computeResult2(StateContext context, int num)
  ~double k, k2:
    requires (context#k stateContextMultiple2())
    ensures (context#k stateContextMultiple2())
  {
    IntCell i1 = new IntCell(MultipleOf(4)[num*4])(4, num*4);
    StateLike r = new StateLive(StateMultipleOf2()[i1])(i1);
    context.setState2(r);
    return r;
  }
```

Figure 5.9: State pattern example - StateLimbo class

```
    boolean checkMod3()
    ~double k:
      requires this#k StateMultipleOf3()
      ensures this#k StateMultipleOf3()
    {
      unpack(this#k StateMultipleOf3());
      boolean temp = (this.cell.getValueInt() % 3 == 0);
      pack(this#k StateMultipleOf3());
      return temp;
    }

    boolean checkMod2()
    ~double k:
      requires this#k StateMultipleOf2()
      ensures this#k StateMultipleOf2()
    {
      unpack(this#k StateMultipleOf2());
      boolean temp = (this.cell.getValueInt() % 2 == 0);
      pack(this#k StateMultipleOf2());
      return temp;
    }
}
```

Figure 5.10: State pattern example - StateLimbo class continuation

precondition as method *computeResult()* from interface *Statelike*, as seen on line 23 in Figure 5.9. The postcondition of the same function in class *StateSleep* ensures more properties than the postcondition written for the function in interface *Statelike*, as seen on lines 24-25 in Figure 5.9. The function *computeResult2()* is similar conceptually to the function *computeResult()*, the only difference being that it refers to the predicate *StateContextMultiple2* instead of the predicate *StateContextMultiple3*. The functions *computeResult()* and *computeResult2()* control the state of the object inside the *Context* parameter and they change it to *StateSleep* and *StateLive* respectively, according to the change of state that we envisioned for the automaton that this example describes.

The *StateContext* class is the controller of the *Statelike* object in the example, to which it has a reference, and it is used to change the state of that object between *StateLive*, *StateLimbo* and *StateSleep*. On lines 4-9 in Figure 5.11 we see the definitions of predicates *StateLive*, *StateSleep*, *StateLimbo*, *StateContextMultiple2* and *StateContextMultiple3*. Note that predicates such as *StateLive* and *StateContextMultiple3* referring to the same object *obj* of type *StateContext* can exist in the same pre-condition (or post-condition) because they refer to different fields of the object *obj*. We keep track of the actual class of the object *obj*, that implements the interface *Statelike*, by declaring a field `var instanceof : [Ref]int` for the objects of type *StateLive*, *StateLimbo* and *StateSleep*. This field will be equal to 1 if the actual class of *obj* is *StateLive*, 2 if it is *StateLimbo* and 3 if it is *StateSleep*.

```
class StateContext {
  Statelike myState;

  predicate StateContextMultiple2() = exists StateLike m, double k :
      this.myState -> m && (m#k StateMultipleOf2())
  predicate StateContextMultiple3() = exists StateLike m, double k :
      this.myState -> m && (m#k StateMultipleOf3())

  StateContext(Statelike newState)
  ensures this.myState == newState;
  {
    this.myState = newState;
  }

  void setState2(Statelike newState)
  ~double k1, k2:
    requires this#k1 StateContextMultiple2()
    requires newState#k2 StateMultipleOf2()
    ensures this#k1 StateContextMultiple2()[newState]
  {
    unpack(this#k1 StateContextMultiple2());
    this.myState = newState;
    pack(this#k1 StateContextMultiple2())[newState];
  }

  void setState3(Statelike newState)
  ~double k1, k2:
    requires this#k1 StateContextMultiple3()
    requires newState#k2 StateMultipleOf3()
    ensures this#k1 StateContextMultiple3()[newState]
  {
    unpack(this#k1 StateContextMultiple3());
    this.myState = newState;
    pack(this#k1 StateContextMultiple3())[newState];
  }
```

Figure 5.11: State pattern example - StateContext class

```
   Statelike computeResultSC(int num)
   ~double k1, k2:
     requires (this#k1 StateContextMultiple3())
     ensures (this#k1 StateContextMultiple3())
   {
     unpack(this#k1 stateClientMultiple3());
     Statelike temp = this.myState.computeResult(this, num);
     pack(this#k1 stateClientMultiple3());
     return temp;
   }

   Statelike computeResult2SC(int num)
   ~double k1, k2:
     requires (this#k1 StateContextMultiple2())
     ensures (this#k1 StateContextMultiple2())
   {
     unpack(this#k1 stateClientMultiple2());
     Statelike temp = this.myState.computeResult2(this, num);
     pack(this#k1 stateClientMultiple2());
     return temp;
   }

   boolean stateContextCheckMultiplicity3()
   ~double k:
     requires this#k StateContextMultiple3()
     ensures this#k StateContextMultiple3()
   {
     unpack(this#k StateContextMultiple3());
     boolean temp = this.myState.checkMod3();
     pack(this#k StateContextMultiple3());
     return temp;
   }

   boolean stateContextCheckMultiplicity2()
   ~double k:
     requires this#k StateContextMultiple2()
     ensures this#k StateContextMultiple2()
   {
     unpack(this#k StateContextMultiple2());
     boolean temp = this.myState.checkMod2();
     pack(this#k StateContextMultiple2());
     return temp;
   }
}
```

Figure 5.12: State pattern example - StateContext class continuation

One can say that the predicates in class StateContext from Figures 5.11 and 5.12 are of two forms: either `stateX` - such as `stateLive, stateLimbo, stateSleep` or of the form `stateContextMultipleX` - such as `stateContextMultiple2`, `stateContextMultiple3`. Although both of these predicates refer to the field `myState`, they refer to different parts, different fields, of `myState`. Even though we only have one field permission to `myState`, both of these kinds of predicates can exist at the same time because they refer to different fields of `myState`.

There are two invariants that an object of type Statelike can satisfy: being a multiple of 3 or being a multiple of 2 (as the interface promises). Similarly to an automaton, there are two paths that the machine controller is programmer to be on: it can be in StateLive, then switch to StateLimbo and finally to StateSleep - and repeat this process indefinitely, or it can start in the StateLive state, transition to the StateSleep state and then be go to the StateLimbo state. Notice that we removed the argument `String[] args` from the `main1()` and `main2()` functions in Figures 5.14 and 5.15 of all the design pattern examples because the Oprop methodology and tool do not currently offer support for Strings or arrays. We leave those features as interesting future work.

The `main1()` method shows the object `scontext1`, that transitions between having as a field a reference to a Statelike object. Although it changes state, it is always a multiple of 3, which is the invariant that is ensured by the `computeResult()` method, or a multiple of 2, which is the invariant that is ensured by the `computeResult2()` method. Thus if we had 2 pointers to the `scontext1` object that start off as satisfying the multiple of 3 invariant, each having a fraction of half, both pointers would be able to rely on the fact that the reference inside of `scontext1` is a multiple of 3, while being able to change its state between StateLive, StateLimbo and StateSleep, that each has its own implementation of what the predicate *multiple of 3* represents. This is the primary difference between object propositions and separation logic in this example - the ability of a client to change the state of an object as long as the invariant that was initially stated is preserved even if that client only has a fractional permission to that object.

The `main2()` method showcases another scenario where the object `scontext2`, that transitions between the three states StateLive, StateSleep and StateLimbo, has as a field a reference to a Statelike object that is a multiple of 2. The invariant that the object Statelike is a multiple of 2 is enforced by the specifications of the method `computeResult2()`. For objects of the Statelike class to be able to satisfy 2 invariants, being either a multiple of 3 or a multiple of 2, shows the difference between object propositions and classical invariants (or considerate reasoning [68]).

The state design pattern introduces the $interface$ and $implements$ keywords and ideas to Oprop. We have manually translated the Oprop classes from figures 5.3, 5.4, 5.5, 5.6 5.7, 5.9, 5.10, 5.11, 5.12, 5.14 and 5.15.

Boogie only accepts a single file as input, so we have concatenated the translated files into the file
https://github.com/ligianistor/boogie/blob/master/statelatest.bpl that can be verified in Boogie.

The idea is that if a predicate is declared in the interface (only by the name of the predicate, since the interface will not have the fields declared), then the body of the predicate will be declared in the classes that implement that interface. Since the input to the Boogie verifier has to be a single Boogie program, each method name or predicate name from the interface `Statelike`

```
class StateClient {
  StateContext scon;

  predicate StateClientMultiple2() =
      exists double k, StateContext s :: this.scon->s && (s#k
         StateContextMultiple2())
  predicate StateClientMultiple3() =
      exists double k, StateContext s :: this.scon->s && (s#k
         StateContextMultiple3())

  StateClient(StateContext s)
  ensures this.scon == s
  {
    this.scon = s;
  }

  boolean stateClientCheckMultiplicity3()
  ~double k:
    requires this#k StateClientMultiple3()
    ensures this#k StateClientMultiple3()
  {
    unpack(this#k StateClientMultiple3())[this.scon]
    boolean temp = this.scon.stateContextCheckMultiplicity3();
    pack(this#k StateClientMultiple3())[this.scon]
    return temp;
  }

  boolean stateClientCheckMultiplicity2()
  ~double k:
    requires this#k StateClientMultiple2()
    ensures this#k StateClientMultiple2()
  {
    unpack(this#k StateClientMultiple2())[this.scon]
    boolean temp = this.scon.stateContextCheckMultiplicity2();
    pack(this#k StateClientMultiple2())[this.scon]
    return temp;
  }
```

Figure 5.13: State pattern example - StateClient class

```
void main1()
~double k:
{
  IntCell i1 = new IntCell(MultipleOf(21))(21);
  Statelike st1 = new StateLive(StateMultipleOf3())(i1);
  StateContext scontext1 = new StateContext(stateContextMultiple3()
     [])(st1);
  StateClient sclient1 = new StateClient(stateClientMultiple3()[])(
     scontext1);
  StateClient sclient2 = new StateClient(stateClientMultiple3()[])(
     scontext1);
  scontext1.computeResultSC(1);
  sclient1.stateClientCheckMultiplicity3();
  scontext1.computeResultSC(2);
  sclient2.stateClientCheckMultiplicity3();
  scontext1.computeResultSC(3);
  sclient1.stateClientCheckMultiplicity3();
}
```

Figure 5.14: State pattern example - main1() function

```
void main2()
~double k:
{
  IntCell i2 = new IntCell(MultipleOf(4))(4);
  Statelike st2 = new StateLive(StateMultipleOf2())(i2);
  StateContext scontext2 = new StateContext(stateContextMultiple2()
     [])(st2);
  StateClient sclient3 = new StateClient(stateClientMultiple2()[])(
     scontext2);
  StateClient sclient4 = new StateClient(stateClientMultiple2()[])(
     scontext2);
  scontext2.computeResult2SC(1);
  sclient3.stateClientCheckMultiplicity2();
  scontext2.computeResult2SC(2);
  sclient4.stateClientCheckMultiplicity2();
  scontext2.computeResult2SC(3);
  sclient3.stateClientCheckMultiplicity2();
}
}
```

Figure 5.15: State pattern example - main2() function

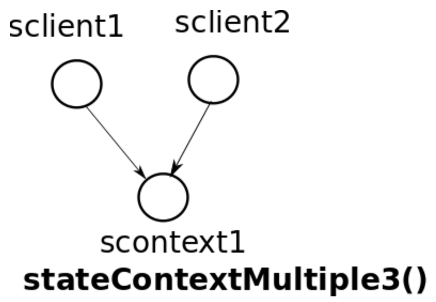

Figure 5.16: Diagram of main1() for state pattern

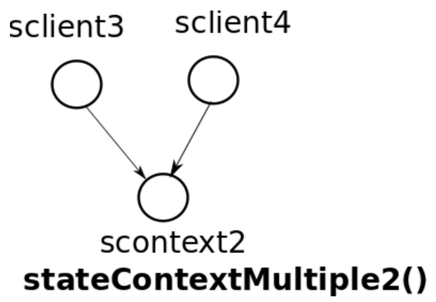

Figure 5.17: Diagram of main2() for state pattern

that is implemented in the classes `StateLive, StateSleep` or `StateLimbo` has to be suffixed with the name of the implementing class, as to not create name duplication. Each time an object of interface type `Statelike` calls a method, that call has to be preceded by an `if` statement that decides which actual implementation is called depending on the actual class of the calling object. In the Boogie translation files there is a map `instanceof` (inspired by the *instanceof* keyword from Java that is used to check whether an object is an instance of a specified type) that points to 1, 2 or 3 for each object of type `Statelike`, depending on the actual type of the current object. If we do not know the actual type of an object, we must assume that it can be any of the three types and we must make sure that the verificaton succeeds in all three cases.

We write an extended grammar to incorporate the notions specific to interfaces. We start from the grammar in Section 3.1 and make the necessary changes. Below we present only the rules that have changed.

$$\begin{array}{rcl}
\mathsf{Prog} & ::= & \overline{\mathsf{InterfDecl}}\ \overline{\mathsf{ClDecl}}\ \mathsf{e} \\
\mathsf{InterfDecl} & ::= & \texttt{interface}\ I\ \texttt{\{}\ \overline{\mathsf{InterfPredDecl}}\ \overline{\mathsf{InterfMthDecl}}\ \texttt{\}} \\
\mathsf{InterfPredDecl} & ::= & \texttt{predicate}\ Q\ \texttt{(}\overline{\mathsf{T\ x}}\texttt{)} \\
\mathsf{InterfMthDecl} & ::= & \mathsf{T}\ m\ \texttt{(}\overline{\mathsf{T\ x}}\texttt{)}\ \mathsf{MthSpec} \\
\mathsf{ClDecl} & ::= & \texttt{class}\ C\ \texttt{(implements I)?}\ \texttt{\{}\ \overline{\mathsf{FldDecl}}\ \overline{\mathsf{PredDecl}}\ \overline{\mathsf{MthDecl}}\ \texttt{\}}
\end{array}$$

An interface will contain the names of predicates and the specifications of the methods inside it, but not the definitions of those predicates or the bodies of those methods. The definitions of the predicates will have to be given in each class that implements that interface and likewise the implementations of the methods will have to satisfy the specifications given in the interface for those methods. Note that a class might or might not implement an interface and that is why we added the text (`implements I`) to the grammar, followed by the question mark that makes this text optional.

The modified judgments for a well formed program and class are given below and they incorporate the addition of the interface concept.

$$\frac{\vdash \overline{CL} \quad \vdash \overline{Interf}}{\vdash \langle \overline{Interf}, \overline{CL}, e\rangle}\ \textsc{Program}$$

$$\frac{\begin{array}{c}\vdash_C \overline{M} \quad \vdash_C \texttt{interface I}\{\overline{Q(\overline{x})}; \overline{MDecl}\} \\ \overline{M = MDecl1; MBody} \quad MDecl1\ implies\ MDecl\end{array}}{\vdash \texttt{class}\ C\ \texttt{implements I \{}\ \overline{F}\ \overline{Q(\overline{x}) = R}\ \overline{M}\,\texttt{\}}}\ \textsc{Class}$$

5.2 Verifying an Instance of the Proxy Pattern

The second design pattern, proxy, is used when we need a representative object (the proxy) that controls access to another object, which may be remote, expensive to create or compute, or in need of security. Programs that are instances of this pattern have an interface which is implemented by two classes: the proxy class and the "real" class. The proxy object can act as a cache for holding on to data that only needs to be computed once and the "real" object can be thought of as a server that only does computation that is considered expensive on demand. This pattern is similar to the state pattern because it also has an interface that is implemented by multiple classes - two in our example. The difference is that the proxy object can hide information about the real object from the client. This means that one can modify the real object without the client knowing it.

The Oprop classes used in the proxy pattern example are: Sum in Figure 5.20, RealSum in Figures 5.21 and 5.22, ProxySum in Figures 5.23 and 5.24 and ClientSum in Figures 5.25 and 5.26.

5.2.1 Example

In Figure 5.18, inspired from the Wikipedia figure [10], we present the UML diagram of the general proxy pattern. There is a *Subject* interface containing the method *DoAction()* that has to be coded by all classes implementing it. The classes *Proxy* and *RealSubject* implement the *Subject* interface. When a client class, represented in the figure by the class *Client*, asks a *Subject* object to perform an action by calling the method *DoAction()* on it, the proxy object will try to perform that action or resort to its cache if the action was performed earlier and the results are stored in its cache. If the proxy is not able to do the computation it will delegate the action to the object of type *RealSubject* to which it has a reference to. The *RealSubject* will perform the action, which typically is a resource intensive operation, and return the result to the proxy object, which will report the answer to the client. The formal verification of an instance of this pattern will check that the returned answer is correct whether the proxy object computes it or the real subject computes it.

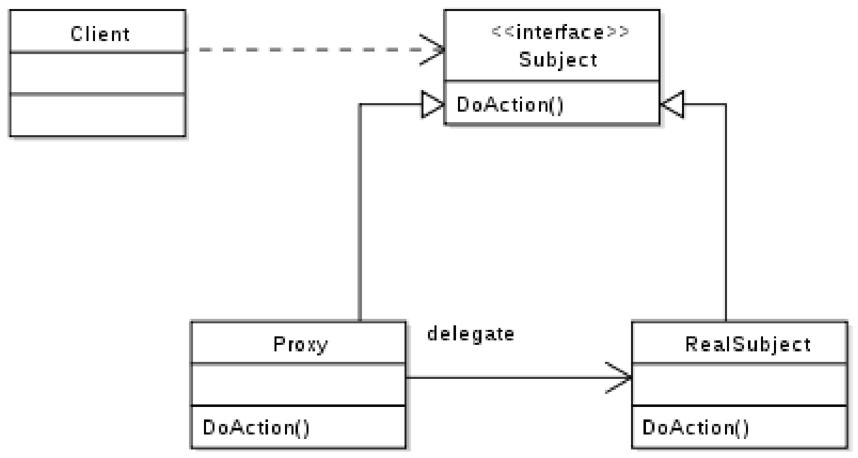

Figure 5.18: UML Diagram of Proxy Pattern

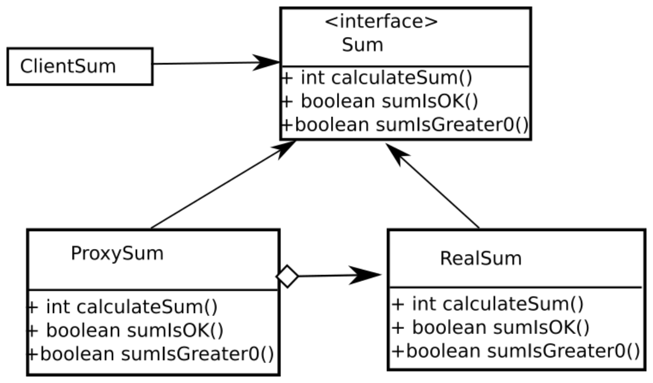

Figure 5.19: UML Diagram of Our Example of Proxy Pattern

In the remainder of this subsection we give an intuitive explanation of our instance of the proxy pattern, we present the annotated Oprop classes that make up our example and for each class we detail the more interesting parts.

In Figure 5.19 we show the UML diagram of the classes in our example implementing the proxy design pattern. The interface `Sum` is implemented by the classes `ProxySum` and `RealSum`. When the method `calculateSum()` is called on an object of type `RealSum`, the sum of the first `n` natural numbers is always calculated; when `calculateSum()` is called on an object of type `ProxySum`, if the sum has already been calculated then the value is taken from the object of type `ProxySum`, otherwise the call is forwarded to the `RealSum` object which calculates the value. The class `ClientSum` shows different snippets of code - where we start with an object of type `ProxySum` - that showcase the differences between verification using object propositions versus using classical invariants or separation logic.

Along the same lines, the method `addOneToSum()` is declared in the interface `Sum` and implemented by the classes `ProxySum` and `RealSum`. When this method is called on an object of type `RealSum`, the sum of the first `n` natural numbers is always calculated by the method

```
interface Sum {
  predicate BasicFields(int n1, double s1);
  predicate SumOK();
  predicate SumGreater0();

  double calculateSum();
  ~double k:
  int n1, double s1:
    requires this#k BasicFields(n1, s1)
    ensures this#k SumOK()

  double addOneToSum();
  ~double k:
  int n1, double s1:
    requires this#k BasicFields(n1, s1)
    ensures this#k SumGreater0()

  boolean sumIsOK();
  ~double k:
    requires this#k SumOK()
    ensures this#k SumOK()

  boolean sumIsGreater0();
  ~double k:
    requires this#k SumGreater0()
    ensures this#k SumGreater0()
}
```

Figure 5.20: Proxy pattern example - interface Sum

`calculateSum` from the class `RealSum` being called, which is then increased by 1; when `addOneToSum()` is called on an object of type `ProxySum`, if the sum has already been calculated then the value is taken from the object of type `ProxySum`, otherwise the call is forwarded to the `RealSum` object which calculates the value. In the class `ClientSum` we show a snippet of code where we start with an object of type `ProxySum` that will satisfy the invariant *sumGreater0* throughout the `main2()` method and thus other objects referencing this object will be able to rely on this invariant.

The `calculateSum()` from class `RealSum` method calculates the sum of the first `n` positive integers. This method can be transformed into the iterative version of the sum, instead of the recursive version, when the Oprop tool implements loop features (such as *while*, *for*).

The interface *Sum* 5.20 declares the predicates *BasicFields* (which is only defined so that when unpacked, the caller gets access to the fields of the object of type *Sum*), *SumOK* and *SumGreater0*. All these predicates are implemented in the classes *RealSum* and *ProxySum*, where they could have different implementations. The interface *Sum* also declares methods *calculate-*

Sum(), *sumIsOK()*, *addOneToSum()* and *sumIsGreater0()* that are implemented in the classes *RealSum* and *ProxySum*.

In Figure 5.21 the method `calculateSum()` in the class `RealSum` could be replaced with the iterative computation of the sum of the first `n` natural numbers and the client or the `Proxy` class would not know about that change. The client does not know (or care) that it is using a proxy rather than the real object. In the state pattern, on the other hand, each state and its behavior is visible, the client knows about all the states that implement the common interface.

In Figure 5.23, in the pre-condition of the method *calculateSum()* the predicate *BasicFields()* refers to the object *this* and the predicate *SumOK()* refers to the object *this.realSum*. They do not refer to the same field object and so they can appear in the pre-condition at the same time. Moreover, the predicate *BasicFields()* needs to be unpacked so that we have access to the object *this.realSum* in the same pre-condition and we can write an object proposition about it.

The method *addOneToSum()* defined in class *ProxySum* in Figure 5.23 can have any implementation but its specification will have to be at least as strong as the specification of the declaration of the same method in the interface *Sum*. In this way, if the method *addOneToSum()* is called on an object *obj* of type *Sum*, we can rely on the specifications written in the interface for this method, although we do not know the concrete class of the object *obj* - whether it is an object of type *RealSum* or *ProxySum*. Note that all the annotations - definitions of predicates, pre- and postconditions, the calls to `unpack` in order to get access to the fields of an object, the calls to `pack` that are needed to pack back to a predicate that is satisfied - have to be written by the programmer. He is the one that decides all the annotations, while the object proposition methodology and our Oprop tool will decide whether those specifications are satisfied.

In the `main1()` and `main2()` methods in Figure 5.26 (we have two such methods in order to differentiate the two usages in client code) we have an object `s` of type `Sum`. The first time when we call `calculateSum()`, the result is coming from class `RealSum`. We have a `ClientSum` object that has as a field the object `s`, on which it calls the checking method `checkSum()`. The second time when the method `calculateSum()` is called on the object `s`, we do not resort to `RealSum` to calculate the sum, but instead use the cached copy of the sum that we have in the object `Proxysum`. All the times when we call `checkSum()`, the code verifies successfully because the invariant `sumOK` holds whether the sum is obtained using the `ProxySum` or using `RealSum`.

Note that if the two classes `ProxySum` and `RealSum` were not entitled as they are, and instead they were named `ClassA` and `ClassB` to distinguish them more, it would be more meaningful to the reader to know that the invariant `sumOK` holds whether we go to `ClassA` or `ClassB` to get our result.

Objects that point to the object `s` can rely on this invariant and change the object `s` while preserving the invariant, even if they have only a half fraction to `s`, as can be seen in this example and others throughout this thesis. This is the crux of the difference between the verification using normal separation logic and object propositions.

We have added the predicate `sumGreater0` to show that two objects of the class `Sum` can have different invariants. This idea is the difference between object propositions and classical invariants, where all objects of the same class have to respect the same invariant.

The Oprop classes of the proxy example that can be seen in Figures 5.21, 5.22, 5.23, 5.24, 5.25 and 5.26 are translated into Boogie. We have concatenated the translated files into the file

```
class RealSum implements Sum {
  int n;
  double sum;

  predicate BasicFields(int n1, double s1) =
      this.n -> n1 && this.sum -> s1 && n1>0 && s1>=0
  predicate SumOK() = exists double k, int n1 :
      this#k BasicFields(n1, (n1 * (n1+1) / 2))
  predicate SumGreater0() = exists double k, int n1, double s1 :
      this#k BasicFields(n1, s1) && s1 > 0

  RealSum(int n1)
    requires n1 > 0
    ensures this#1.0 SumOK()
  {
    this.n = n1;
    this.sum = 0;
    pack(this#1 BasicFields(n1, 0));
    this.calculateSum();
  }

  double getRealSum()
  ~double k:
   requires this#k SumOK()
   ensures this#k SumOK()
  {
    unpack(this#k SumOK());
    boolean temp = this.sum;
    pack(this#k SumOK());
    return temp;
  }

  double addOneToSum()
  ~double k:
  int n1, double s1:
    requires this#k BasicFields(n1, s1)
    ensures this#k SumGreater0()
    ensures result > 0
  {
    unpack(this#k BasicFields(n1, s1));
    this.sum = this.sum+1;
    double temp = this.sum;
    pack(this#k BasicFields(n1, s1));
    pack(this#k SumGreater0())[n1, this.sum];
    return temp;
  }
```

Figure 5.21: Proxy pattern example - class RealSum

```
  double calculateSum()
  ~double k:
  int n1, double s1:
    requires this#k BasicFields(n1, s1)
    ensures this#k SumOK()
    ensures result > 0
  {
    unpack(this#k BasicFields(n1, s1));
    this.sum = (n1 * (n1+1) / 2);
    double temp = this.sum;
    pack(this#k BasicFields(n1, n1 * (n1+1) / 2));
    pack(this#k SumOK())[n1];
    return temp;
  }

  boolean sumIsOK()
  ~double k:
    requires this#k SumOK()
    ensures this#k SumOK()
  {
    unpack(this#k SumOK());
    boolean temp = (this.sum == (this.n * (this.n+1) / 2));
    pack(this#k SumOK())[this.n];
    return temp;
  }

  boolean sumIsGreater0()
  ~double k:
    requires this#k SumGreater0()
    ensures this#k SumGreater0()
  {
    unpack(this#k SumGreater0());
    boolean temp = (this.sum > 0);
    pack(this#k SumGreater0())[this.sum];
    return temp;
  }
}
```

Figure 5.22: Proxy pattern example - class RealSum continuation

```
class ProxySum implements Sum {
  RealSum realSum;
  double sum;
  int n;

  predicate BasicFields(int n1, double s1) =
      exists RealSum rs, double k2:
      this.realSum -> rs && this.sum -> s1 && this.n -> n1
     && n1>0 && && s1 >= 0 && ((rs!=null) ~=> rs#k2 SumOK())
  predicate SumOK() = exists double k, int n1 :
      this#k BasicFields(n1, (n1 * (n1+1) / 2))
  predicate SumGreater0() = exists double k, int n1, double s1 :
      this#k BasicFields(n1, s1) && s1 > 0

  ProxySum(int n1)
    ensures this#1.0 BasicFields(n1, 0)[null]
  {
    this.n = n1;
    this.sum = 0;
    this.realSum = null;
  }

  double calculateSum()
  ~double k1, k2:
  int n1, double s1, RealSum ob:
    requires this#k1 BasicFields(n1, s1)[ob]
    ensures this#k1 SumOK()
  {
    unpack(this#k1 BasicFields(n1, s1))[ob];
    if (ob == null) {
      ob = new RealSum(SumOK()[n1])(n1);
    }
    this.sum = ob.getRealSum();
    double temp = this.sum;
    pack(this#k1 BasicFields(this.n, this.sum))[ob];
    unpack(ob#k2 SumOK());
    pack(this#k1 SumOK());
    return temp;
  }
```

Figure 5.23: Proxy pattern example - class ProxySum

```
   double addOneToSum()
   ~double k1, k2:
   int n1, double s1, RealSum ob:
     requires this#k1 BasicFields(n1, s1)[ob]
     ensures (this#k1 SumGreater0())
   {
     unpack(this#k1 BasicFields(n1, s1))[ob];
     if (ob == null) {
       ob = new RealSum(SumOK()[n1])(n1);
       unpack(ob#1.0 SumOK())[n1]
     }
     this.sum = ob.addOneToSum();
     double temp = this.sum;
     pack(this#k1 BasicFields(n1, temp))[ob];
     pack(this#k1 SumGreater0());
     return temp;
   }

   boolean sumIsOK()
   ~double k:
     requires this#k SumOK()
     ensures this#k SumOK()
   {
     unpack(this#k SumOK());
     boolean temp = (this.sum == (this.n * (this.n + 1) / 2));
     pack(this#k SumOK())[this.n];
     return temp;
   }

   boolean sumIsGreater0()
   ~double k:
     requires this#k SumGreater0()
     ensures this#k SumGreater0()
   {
     unpack(this#k SumGreater0());
     boolean temp = (this.sum > 0);
     pack(this#k SumGreater0())[this.sum];
     return temp;
   }
}
```

Figure 5.24: Proxy pattern example - class ProxySum continuation

```
class ClientSum {
  Sum sumClient;

  predicate ClientSumOK() =
      exists double k, Sum s :: this.sumClient->s && (s#k SumOK())

  predicate ClientSumGreater0() =
      exists double k, Sum s :: this.sumClient->s && (s#k SumGreater0
         ())

  ClientSum(Sum sum1)
    ensures (this.sumClient -> sum1);
  {
    this.sumClient = sum1;
  }

  boolean checkSumIsOK()
  ~double k1, k2:
    requires (this#k1 ClientSumOK())
    ensures (this#k1 ClientSumOK())
  {
    unpack(this#k1 ClientSumOK());
    boolean temp = this.sumClient.sumIsOK();
    pack(this#k1 ClientSumOK());
    return temp;
  }

  boolean checkSumGreater0()
  ~double k1, k2:
    requires (this#k1 ClientSumGreater0())
    ensures (this#k1 ClientSumGreater0())
  {
    unpack(this#k1 ClientSumGreater0());
    boolean temp = this.sumClient.sumIsGreater0();
    pack(this#k1 ClientSumGreater0());
    return temp;
  }
```

Figure 5.25: Proxy pattern - class ClientSum

```
    void main1()
    ~double k:
    {
      Sum s = new ProxySum(BasicFields(5, 0))(5);
      s.calculateSum();
      ClientSum client1 = new ClientSum(ClientSumOK()[])(s);
      ClientSum client2 = new ClientSum(ClientSumOK()[])(s);
      client1.checkSumIsOK();
      unpack(s#k SumOK())[5];
      s.calculateSum();
      client2.checkSumIsOK();
    }

    void main2()
    ~double k:
    {
      Sum s2 = new ProxySum(BasicFields(7, 0))(7);
      s2.addOneToSum();
      ClientSum client3 = new ClientSum(ClientSumGreater0()[])(s2);
      ClientSum client4 = new ClientSum(ClientSumGreater0()[])(s2);
      client3.checkSumGreater0();
      unpack(s2#k SumGreater0())[7, 29];
      s2.addOneToSum();
      client4.checkSumGreater0();
    }
}
```

Figure 5.26: Proxy pattern - class ClientSum continuation

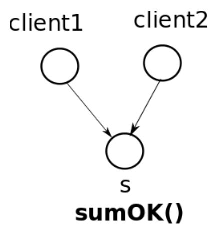

Figure 5.27: Diagram of main1() for proxy pattern

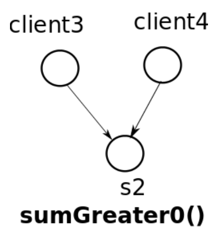

Figure 5.28: Diagram of main2() for proxy pattern - cont.

https://github.com/ligianistor/boogie/blob/master/proxylatest.bpl that can be verified in Boogie.

5.3 Verifying an Instance of the Flyweight Pattern

The third pattern, flyweight, represents programs that have an immutable object that a large number of client objects point to. In the example below we show how the verification works when a new client object is added. We have a comparison to the verification of this example using the invariant technique - we will come back to this comparison later in this section, and also to the verification using separation logic with fractions technique described in the paper "Permission accounting in separation logic" by R. Bornat *et al.* [24]. Although the referenced paper is written in the context of concurrency, if we think of each thread that has access to some common variable as a reference pointing to that common variable, we can apply the separation logic with fractions methodology in the sequential setting. The difference is that when multiple threads have access to a common variable with a fraction less than 1, they only have read access in the separation logic with fractions scenario, while in the object propositions methodology we allow read/write access to this common object as long as the initial predicate is preserved.

Since the flyweight object is immutable, for the flyweight example it is not important to have read/write access to the shared data. One could perform the same verification using separation logic and fractions that one can do with object propositions in the case of the flyweight. The paper describes their approach for concurrent programs; since sequential verification is a special case of concurrent verification, their approach would automatically work for sequential programs, but the reader would have to fill in the details of specializing sequential verification from concurrent verification. The purpose of us verifying an instance of the flyweight pattern is not to show our unique characteristics but that we can solve a problem that prior approaches have; the other examples in this thesis show the unique benefits of our approach. Both scenarios are important: we want our approach to have unique advantages but not be worse for cases that other systems already solve.

5.3.1 Example

Each “flyweight” object is divided into two pieces: the state-dependent (extrinsic) part, and the state-independent (intrinsic) part. Intrinsic state is stored (shared) in the Flyweight object. Extrinsic state is computed when client objects give the necessary information. This necessary information is passed to the Flyweight object when its relevant methods are invoked.

The UML diagram of the general flyweight pattern can be seen in Figure 5.29, inspired from the website [4]. There is a *Factory* class that acts as a repository and stores objects of type *Flyweight*. The client does not create Flyweights directly, and requests them from the Factory. Each Flyweight cannot stand on its own. Any particular attributes that would make sharing impossible must be supplied by the client whenever a request is made of the Flyweight. The Flyweight pattern is useful in cases when the context lends itself to *economy of scale* (i.e., the client can easily compute or look-up the necessary attributes).

In the remainder of this subsection we give an intuitive explanation of our instance of the flyweight pattern, we present the annotated Oprop classes that make up our example and for

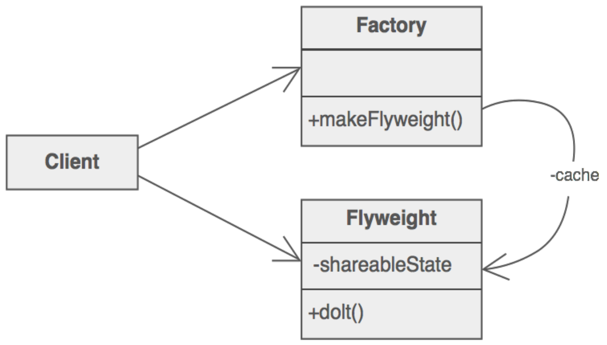

Figure 5.29: UML Diagram of Flyweight Pattern

each class we detail the more interesting parts.

In Figure 5.30 we present the UML diagram of the classes of our example implementing the flyweight pattern. The `College` objects are the flyweights, which have both an immutable, intrinsic state, represented by the simple `collegeNumber` field and by the more complicated field `endowment`, and a mutable, extrinsic state that is given to the flyweights through the `getNumberFacilities()` method called on them. The number of facilities `numFacilities` is extrinsic state for a `College` object.
When objects of class `StudentApplication` are created using the constructor `StudentApplication(college, campusNumber)`, the external information `campusNumber` will determine how many facilities this particular college has. The map data structure `MapCollege` acts as a cache where multiple colleges can be stored and accessed through the `lookup` method. We assume that we can use maps in our logic, such as the `MapCollege` data structure. The caching is needed because of the `endowment` field, which represents a field more difficult to calculate and needs to be stored in the cache. The class `ApplicationWebsite` is where the `main()` method of this example is and where the client code that can be seen in action.

A `College` flyweight can satisfy the invariant `collegeFacilitiesFew` throughout the program, or it can satisfy the invariant `collegeFacilitiesMany`. Each invariant is interesting because it is partly over intrinsic state, and partly over extrinsic state which is passed to the invariant itself as an argument. By offering two different invariants that a flyweight object can satisfy, we are differentiating the verification using object propositions from the verification using classical invariants. While a `College` object satisfies the invariant `collegeFacilitiesFew`, client code can call the method
`changeApplicationFew` that can change the `campusNumber` that a particular student wants to apply to - and this change the number of facilities of that college, while still satisfying the
`collegeFacilitiesFew` invariant. The intuition is that once a student has decided to apply to a college that has few facilities, the student can change the actual college campus that he or she applies to as long as it is a college with few facilities.

Flyweight objects are stored in a factory's repository. The client restrains herself from creating Flyweights directly, and requests them from the factory. If the context lends itself to "econ-

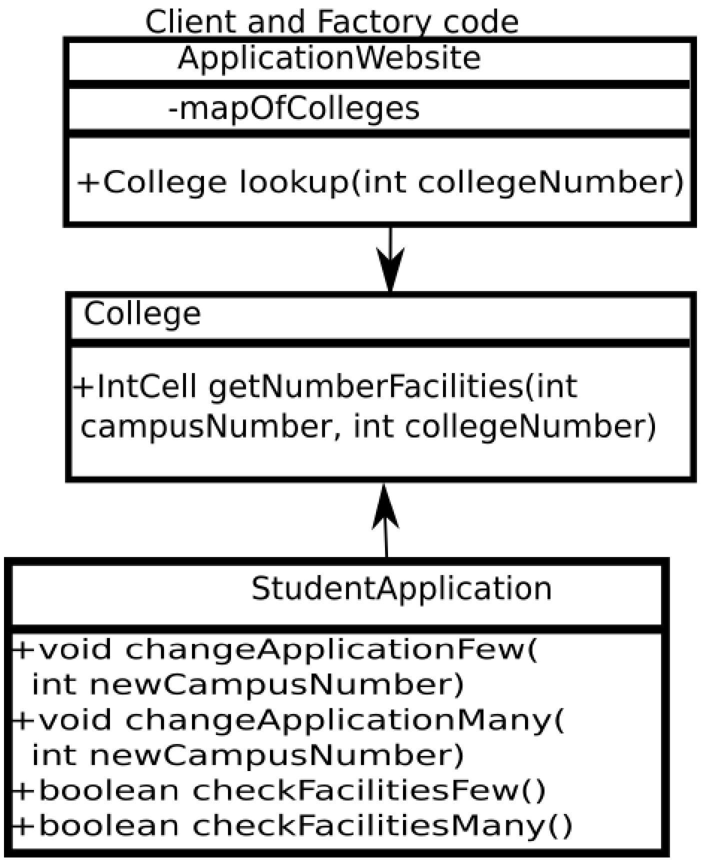
Client and Factory code
ApplicationWebsite
-mapOfColleges
+College lookup(int collegeNumber)
College
+IntCell getNumberFacilities(int campusNumber, int collegeNumber)
StudentApplication
+void changeApplicationFew(int newCampusNumber)
+void changeApplicationMany(int newCampusNumber)
+boolean checkFacilitiesFew()
+boolean checkFacilitiesMany()

Figure 5.30: UML Diagram of Our Example of Flyweight Pattern

omy of scale" (i.e., the client can easily compute or look-up the necessary attributes), then the Flyweight pattern offers appropriate leverage.

In Figure 5.31, instances of the `College` class will be the Flyweights. The example only uses numbers because they are readily implemented in our Oprop tool, but future work can easily add support for Strings or other Java data types. The data structure `MapCollege` acts as a factory and cache for `College` flyweight objects.

The Oprop classes used in the flyweight pattern example are: College in Figure 5.31, StudentApplication in Figures 5.32, 5.33 and 5.34 and ApplicationWebsite in Figures 5.35, 5.36, 5.37 and 5.38. The invariants that a `Flyweight` object, i.e., an object of type `College` can satisfy are
`collegeFacilitiesMany` or `collegeFacilitiesFew`. Each of these two invariants depends on both the intrinsic state of a College and its extrinsic state.

Class *College* in Figure 5.31 represents the flyweight class. We define three predicates, *CollegeNumberField*, which simply gives access to the field *collegeNumber* when unpacked, and *CollegeFacilitiesMany* and *CollegeFacilitiesFew*, which state whether this college has many or few facilties, depending on the parameter *numFacilities* that gets passed to these predicates. The constructor computes the *endowment* value; this value only needs to be computed once, when a College object is instantiated. All the objects referencing this flyweight object can take advantage of the fact that the value of the endowment was already calculated. The method *getNumberFacilities()* has the intrinsic value *collegeNumber* (the parameter *colNum* is provided only to check that it is equal to the value of *collegeNumber*) and the extrinsic value *campNum*, which is used for calculating the return value for the number of facilities.

In Figure 5.33 we can see the two methods *changeApplicationFew()* and *changeApplicationMany()* that preserve the predicates *StudentAppFacilitiesFew* and *StudentAppFacilitiesMany*, which are wrapper predicates around the predicates *CollegeFacilitiesFew* and *CollegeFacilitiesMany*. While the methods *changeApplicationFew()* and *changeApplicationMany()* make changes to the *facilities* field of class *StudentApplication*, in Figure 5.34 one can see the methods *checkFacilitiesFew()* and *checkFacilitiesMany()* that simply check that the predicates *StudentAppFacilitiesFew* and *StudentAppFacilitiesMany* hold.

Class *ApplicationWebsite* in Figure 5.35 contains a field *mapOfColleges*, which acts as a cache for colleges. This field is put in this class because each website in our example has its own list of colleges that users can apply to. The method *makeMapNull()* is used in the constructor of *ApplicationWebsite* in order to initialize the field *mapOfColleges*. Note that on lines 31-39 in Figure 5.35 and lines 6-12 and 19-25 in Figure 5.36 the code is commented out; this is because these functions are standard and coming from external libraries. We initially implemented them to get a better idea of the pre- and post-conditions that should hold, but their implementations are usually black boxes and when using them in client code, one should only rely on the specifications provided by the authors of these functions.

The method *lookup()* from Figure 5.36 returns a reference to the college object from *mapOfColleges* for which *collegeNumber* is equal to *colNum*. As seen in the body of this method, field *mapOfColleges* acts as a cache, according to the general idea of the flyweight design pattern.

In the `main1()` and `main2()` methods in Figures 5.37 and 5.38, the first part in each method highlights the differences between the classical invariants verification methodology and object propositions, while the second part shows the differences between separation logic and

```
class College {
  int collegeNumber;
  int endowment;

  predicate CollegeNumberField(int c) =
      this.collegeNumber -> c && (c>0)
  predicate CollegeFacilitiesMany(int numFacilities) =
      exists double k, int c : this#k CollegeNumberField(c) &&
      (numFacilities >= 10 * c)
  predicate CollegeFacilitiesFew(int numFacilities) =
    exists double k, int c : this#k CollegeNumberField(c) &&
    (numFacilities <= 4 * c)

  College(int number)
    ensures this#1.0 CollegeNumberField(number);
  {
    this.collegeNumber = number;
    this.endowment = (number *1000) - 5;
  }

  int getCollegeNumber()
  ~double k:
  int c:
    requires this#k CollegeNumberField(c)
    ensures this#k CollegeNumberField(c)
  {
    unpack(this#k CollegeNumberField(c));
    int temp = this.collegeNumber;
    pack(this#k CollegeNumberField(c));
    return temp;
  }

  int getNumberFacilities(int campNum, int colNum)
  ~double k:
    requires this#k CollegeNumberField(colNum)
    requires campNum > 0;
    ensures this#k CollegeNumberField(colNum)
    ensures result == colNum * campNum;
    ensures result > 0;
  {
    unpack(this#k CollegeNumberField(colNum));
    int temp = colNum * campNum;
    pack(this#k CollegeNumberField(colNum));
    return temp;
  }
}
```

Figure 5.31: Flyweight example - College class

```
class StudentApplication {
  College college;
  int campusNumber;
  int facilities;

  predicate StudentApplicationFields(College c, int campNum) =
      (this.college -> c) && (this.campusNumber -> campNum)

  predicate StudentAppFacilitiesMany() =
      exists double k, double k1, College col, int campNum,
      int fa : this.facilities -> fa &&
      this#k1 StudentApplicationFields(col, campNum) &&
      (col#k CollegeFacilitiesMany(fa)[col.number])

  predicate StudentAppFacilitiesFew() =
      exists double k, double k1, College col, int campNum,
      int fa : this.facilities -> fa &&
      this#k1 StudentApplicationFields(col, campNum) &&
      (col#k CollegeFacilitiesFew(fa)[col.number])

  StudentApplication(College col, int campusNum)
  ~double k:
  int colcolNumber;
    requires campusNum > 0;
    requires (col#k CollegeNumberField(colcolNumber));
    ensures (this#1.0 StudentApplicationFields(col, campusNum));
    ensures ( (campusNum <= 4) && (campusNum > 0) ) ~=>
      (col#k CollegeFacilitiesFew(this.facilities));
    ensures (campusNum >= 10) ~=>
      (col#k CollegeFacilitiesMany(this.facilities));
  {
    this.college = col;
    this.facilities = col.getNumberFacilities(campusNum, colcolNumber)
        ;
    this.campusNumber = campusNum;
    if (0 < campusNum  && campusNum <= 4) {
      pack(col#k CollegeFacilitiesFew(this.facilities));
    } else if (campusNum >= 10) {
      pack(col#k CollegeFacilitiesMany(this.facilities));
    }
}
```

Figure 5.32: Flyweight example - StudentApplication class

```
void changeApplicationFew(int newCampusNumber)
~double k, k2:
  requires newCampusNumber > 0;
  requires this#k StudentAppFacilitiesFew()
  ensures this#k StudentAppFacilitiesFew()
{
  unpack(this#k StudentAppFacilitiesFew());
  this.campusNumber = newCampusNumber % 4;
  unpack(this.college#k2 CollegeFacilitiesFew(this.facilities));
  this.facilities = this.college.getNumberFacilities(this.
     campusNumber);
  pack(this.college#k2 CollegeFacilitiesFew(this.facilities));
  pack(this#k StudentAppFacilitiesFew());
}

void changeApplicationMany(int newCampusNumber)
~double k, k2:
  requires newCampusNumber > 0;
  requires this#k StudentAppFacilitiesMany()
  ensures this#k StudentAppFacilitiesMany()
{
  unpack(this#k StudentAppFacilitiesMany());
  this.campusNumber = newCampusNumber * 10 + 1;
  unpack(this.college#k2 CollegeFacilitiesMany(this.facilities));
  this.facilities = this.college.getNumberFacilities(this.
     campusNumber);
  pack(this.college#k2 CollegeFacilitiesMany(this.facilities));
  pack(this#k StudentAppFacilitiesMany());
}
```

Figure 5.33: Flyweight example - StudentApplication class continuation 1

```
  boolean checkFacilitiesFew()
  ~double k:
    requires this#k StudentAppFacilitiesFew()
    ensures this#k StudentAppFacilitiesFew()
  {
    unpack(this#k StudentAppFacilitiesFew());
    boolean temp = (this.facilities <= 4 * this.campusNumber);
    pack(this#k StudentAppFacilitiesFew())
    return temp;
  }

  boolean checkFacilitiesMany()
  ~double k:
    requires this#k StudentAppFacilitiesMany()
    ensures this#k StudentAppFacilitiesMany()
  {
    unpack(this#k StudentAppFacilitiesMany());
    boolean temp = (this.facilities >= 10 * this.campusNumber);
    pack(this#k StudentAppFacilitiesMany());
    return temp;
  }
}
```

Figure 5.34: Flyweight example - StudentApplication class continuation 2

```
class ApplicationWebsite {
  map<int, College> mapOfColleges;
  int maxSize;

  predicate MapOfCollegesField(map<int, College> m) =
      this.mapOfColleges -> m

  predicate KeyValuePair(int key, College value) =
      exists double k1, double k2, map<int, College> m ::
      this#k1 MapOfCollegesField(m) && (m[key] == value) &&
      (value!=null) ~=> value#k2 CollegeNumberField(key)

  void makeMapNull(int i, map<int, College> ma)
  ~ double k, k1, k2, k3:
    requires (i >= 0);
    requires this#k1 MapOfCollegesField(ma);
    requires (forall int j :: ((j<=i) && (j>=0)) => exists College c::
        this#1.0 KeyValuePair(j, c));
    ensures (forall int j :: ((j<=i) && (j>=0)) => this#1.0
       KeyValuePair(j, null));
    ensures this#k3 MapOfCollegesField(ma);
  {
    if (i==0) {
      unpack(this#k2 KeyValuePair(i, ma[i]));
      unpack(this#k3 MapOfCollegesField(ma));
      this.mapOfColleges[i] = null;
      pack(this#k3 MapOfCollegesField(ma));
      pack(this#1.0 KeyValuePair(i, null));
    } else {
      this.makeMapNull(i-1);
      unpack(this#k2 KeyValuePair(i, ma[i]));
      unpack(this#k3 MapOfCollegesField(ma));
      this.mapOfColleges[i] = null;
      pack(this#k3 MapOfCollegesField(ma));
      pack(this#1.0 KeyValuePair(i, null));
    }
  }
```

Figure 5.35: Flyweight example - ApplicationWebsite class

```
boolean  containsKey(int key1)
~double k, k2:
  requires exists c1:College ==> (this#k2 KeyValuePair(key1, c1));
  ensures (result == true) ==> (exists c:College ==> (this#k2
     KeyValuePair(key1, c))
  ensures (result == false) ==> (this#k2 KeyValuePair(key1, null))
{
  boolean b = true;
  unpack(this#k2 KeyValuePair(key1, c1));
  if (this.mapOfColleges[key1] == null) {
    b = false;
  }
  pack(this#k2 KeyValuePair(key1, null));
  return b;
}

College lookup(int colNum)
~double k:
  requires (0 < colNum);
  requires (exists College cinit :: this#k KeyValuePair(colNum,
     cinit))
  ensures this#k KeyValuePair(colNum, result)
{
  boolean b = this.containsKey(colNum);
  if (!b) {
    College c = new College(CollegeNumberField(colNum))(colNum);
    unpack(this#k KeyValuePair(colNum, cinit));
    this.mapOfColleges[colNum] := c;
    pack(this#k KeyValuePair(colNum, c));
    return c;
  } else {
    unpack(this#k KeyValuePair(colNum, cinit));
    College temp = cinit;
    pack(this#k KeyValuePair(colNum, cinit));
    return temp;
  }
}
```

Figure 5.36: Flyweight example - ApplicationWebsite class continuation 1

```
ApplicationWebsite(map<int, College> ma)
ensures this.mapOfcolleges -> ma;
{
    this.mapOfcolleges = ma;
}

void InitializeApplicationWebsite(int maxSize1, map<int, College> ma
    )
~double k:
  requires (this#k MapOfCollegesField(ma))
  requires (maxSize1>=0);
  requires
  ensures (forall int j : ((j<=maxSize1) && (j>=0)) ~=>
    this#1.0 KeyValuePair(j, null));
  ensures this.maxSize -> maxSize1;
  ensures (this#k MapOfCollegesField(ma))
{
  this.maxSize = maxSize1;
  this.makeMapNull(maxSize1);
}

void main1()
~double k1, k2:
{
  map<int, College> ma;
  ApplicationWebsite website = new ApplicationWebsite(
      MapOfCollegesField(ma))(ma);
  website.InitializeApplicationWebsite(100, ma);
  College college = website.lookup(2);
  unpack(website#k1 KeyValuePair(2, ma[2])[ma]);
  StudentApplication app1 = new StudentApplication(
      StudentAppFacilitiesFew())(college, 3)[2];
  StudentApplication app2 = new StudentApplication(
      StudentAppFacilitiesFew())(college, 2)[2];
  app1.checkFacilitiesFew();
  app1.changeApplicationFew(34);
  app2.checkFacilitiesFew();
}
```

Figure 5.37: Flyweight example - ApplicationWebsite class continuation 2

```
  void main2()
  ~double k1, k2:
  {
    map<int, College> ma;
    ApplicationWebsite website = new ApplicationWebsite(
        MapOfCollegesField(ma))(ma);
    website.InitializeApplicationWebsite(100, ma);
    College college2 = website.lookup(56);
    unpack(website#k1 KeyValuePair(56, ma[56])[ma]);
    StudentApplication app3 = new StudentApplication(
        StudentAppFacilitiesMany())(college2, 45)[56];
    StudentApplication app4 = new StudentApplication(
        StudentAppFacilitiesMany())(college2, 97)[56];
    app3.checkFacilitiesMany();
    app3.changeApplicationMany(78);
    app4.checkFacilitiesMany();
  }
}
```

Figure 5.38: Flyweight example - ApplicationWebsite class continuation 3

object propositions. In the first part we have two objects of type `college`, with one having `collegeBuildingsFew()` as invariant and the other having `collegeBuildingsMany` as invariant. The two instances of the same class having two different invariants throughout the program is the main difference between Oprop and classical invariants.

In the `main1()` method from Figure 5.37 we construct an object `college`, that becomes a field in the `StudentApplication` objects `app1` and `app2`. When constructing `app1` and `app2`, each adds the extrinsic information to the intrinsic information that `college` already has. The intrinsic and extrinsic information are both used in the definition of the predicate `CollegeFacilitiesFew`, which holds for both `app1` and `app2`. This invariant predicate `CollegeFacilitiesFew` holds for both of these objects, because if relies on the intrinsic information of the immutable college and also it relies in a fine way on the extrinsic information. Thus, any reference pointing to `app1` or `app2`, even if that reference has only half a permission to these objects, can rely on the invariant `CollegeFacilitiesFew`, and it can also modify the objects `app1` and `app2` as long as this invariant holds at the end of the modification.

The Oprop classes of the flyweight pattern example that can be seen in Figures 5.31, 5.32, 5.33, 5.34, 5.35, 5.36 and 5.37 are translated into Boogie.

We have concatenated the translated files into the file *https://github.com/ligianistor/boogie/blob/master/flyweightlatest.bpl* that can be verified in Boogie.

5.4 Composite

The Composite design pattern [34] 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 [46], 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. [20, 40]). 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.* [40] 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 are able to solve a practical problem that was proposed as a challenge problem by Leavens *et al.* [46]: the specification and verification of an instance of the composite pattern. As a downside, the specification of the composite is verbose– we have four predicates that are recursive and depend on each other. The source of this verbosity comes from the the fact that the composite example itself is complicated and thus necessitates a complicated specification and verification. Our specification and verification of the composite pattern allows clients to directly mutate any place in the tree, using predicates that reason about one object in the composite at a time and treat other objects in the composite abstractly. Note that a simpler specification is possible in our system but would limit mutation to the root of the tree.

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.

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

  public Composite
   {
    this.count = 1;
    this.left = null;
    this.right = null;
    this.parent = 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)
    else
      newc = newc + left.count;
    if (this.right == null)
    else
      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 [20]. 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 [2]. Our verification proves partial correctness of this version of the Composite pattern.

The Composite pattern has been verified using many different techniques: there were 10 papers dedicated to this problem in the 2008 workshop 'Specification and Verification of Component-Based Systems', Summers *et al.* proposed a solution in VMCAI 2010 [68] and Rosenberg *et al.* did so in 2010 [66]. We do not claim to be the first ones to verify the Composite pattern, but we want to show that our methodology can be used to verify the Composite pattern.

5.5 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 5.39.

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 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!

Although in the beginning stages of the implementation there will be $pack/unpack$ specifications in the code, we do not intend to have them in the final version of our implementation. A predicate of an object has to be unpacked when there are statements (such as assignments) that modify (or read) from the fields of that object. When we finish accessing those fields, we can pack back the predicate. Thus, there is a clear way of inferring when to pack or unpack a

$$\text{predicate } \textit{count}\,(\text{int c}) \equiv \exists \text{ol, or, lc, rc. this.count} \rightarrow \text{c} \otimes$$
$$\text{c} = \text{lc} + \text{rc} + 1 \;\otimes \text{this@}\frac{1}{2}\, left(\text{ol, lc})$$
$$\otimes\, this@\frac{1}{2}\, right(\text{or, rc})$$

$$\text{predicate } \textit{left}\,(\text{Composite ol}, \text{int lc}) \equiv \text{this.left} \rightarrow \text{ol} \otimes$$
$$\big((\text{ol} \neq \text{null} \;\multimap \text{ol@}\frac{1}{2}\, count(\text{lc}))$$
$$\oplus\, (\text{ol} = \text{null} \;\multimap\; \text{lc} = 0)\big)$$

$$\text{predicate } \textit{right}\,(\text{Composite or}, \text{int rc}) \equiv \text{this.right} \rightarrow \text{or} \otimes$$
$$\big((\text{or} \neq \text{null} \;\multimap \text{or@}\frac{1}{2}\, count(\text{rc}))$$
$$\oplus\, (\text{or} = \text{null} \;\multimap\; \text{rc} = 0)\big)$$

$$\text{predicate } \textit{parent}\,() \equiv \exists \text{op, c, k . k} < \frac{1}{2} \Rightarrow\; \text{this.parent} \rightarrow \text{op} \otimes$$
$$\text{op} \neq \text{this} \;\otimes\; \text{this@}\frac{1}{2}\, count(\text{c}) \otimes$$
$$\Big((\text{op} \neq \text{null} \;\multimap \text{op@k}\, parent() \otimes$$
$$(\text{op@}\frac{1}{2}\, left(\text{this, c})$$
$$\oplus\, \text{op@}\frac{1}{2}\, right(\text{this, c})))\oplus$$
$$(\text{op} = \text{null} \;\multimap\; this@\frac{1}{2}\, count(\text{c}))\Big)$$

Figure 5.39: Predicates for Composite

predicate and we do not to explicitly state the $pack/unpack$ expressions.

The proof of partial correctness of the Composite pattern is presented in the supplemental material [2]. We had initially written the proof by hand, but we were finally able to automatically verify our instance of the Composite pattern using our Oprop tool. Our tool can be found online at *lowcost-env.ynzf2j4byc.us-west-2.elasticbeanstalk.com* as a very user friendly web application. The source code of Oprop is open-source and can be found at
https://github.com/ligianistor/Oprop. The Composite.java input file, annotated with the Oprop specifications, can be found in the following pages of this thesis, but it is also included in the examples.zip folder that can be downloaded from the first page of the Oprop web application. In order to see the formal verification of the Composite.java example, the user should input *1* in the input textarea on the first page of the web application, then upload the Composite.java file found in the examples.zip folder (that the user has to unzip) and then press *Verify* on the next page. On the final page of the web application, the user will see the link *inputboogie.bpl* which contains the translation of the Oprop file Composite.java into the Boogie language. The *result.txt* file contains the result of the Boogie verification that uses *inputboogie.bpl* as input, and the final answer to the question 'Does our Composite.java file satisfy its specifications?'.

The full `Composite.java` file, including the Oprop annotations, is given in Figures 5.40, 5.41, 5.42, 5.43 and 5.44.

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.

5.6 Description of Smaller Examples

We have formally verified a number of small examples, each meant to showcase a feature of Oprop: the *DoubleCount.java* example illustrates invariants, *SimpleCell.java* is about manipulating fractions, *Link.java* shows how we handle predicates that have parameters and *Share.java* uses objects that have a reference to a shared common object. Note that the *Link* example was previously mentioned in Section 1.4 and Chapter 2.

The class *DoubleCount.java* is given in given in Figure 5.45. The DoubleCount example has been briefly covered in the AI4FM'14 workshop [57]. There we only presented a sketch of a manual translation while here the example is translated automatically by our Oprop tool. The class

```
package testcases.cager.jexpr;

class Composite {
int count;
Composite left;
Composite right;
Composite parent;

predicate left(Composite ol, int lc) =
  exists Composite op :
  this.parent -> op &&
  this.left -> ol &&
  (ol== null ~=> (lc == 0) ) &&
  (ol != null ~=> ((ol#0.5 count(lc)) && (ol!=this) && (ol!=op)))

predicate right(Composite or, int rc) =
  exists Composite op :
  this.right -> or &&
  this.parent -> op &&
  (or== null ~=> (rc == 0) ) &&
  (or != null ~=> ((or#0.5 count(rc)) && (or!=this) && (or!=op)))

predicate count(int c) =
  exists Composite ol, Composite or,
  int lc, int rc :
  this.count -> c &&
  (c == lc + rc + 1)
 && (this#0.5 left(ol, lc))
 && (this#0.5 right(or, rc))

predicate parent() =
  exists Composite op, int c, double k:
  this.parent -> op &&
  (op != this) &&
  (this#0.5 count(c)) &&
  (op != null ~=> (op#k parent())) &&
  (((op != null) && (op.left == this)) ~=>
    op#0.5 left(this, c)) &&
  (((op != null) && (op.right == this)) ~=>
    op#0.5 right(this, c)) &&
  (op == null ~=> (this#0.5 count(c)))
```

Figure 5.40: Composite.java

```
Composite(int c, Composite l, Composite r, Composite p)
ensures (this.count == c) && (this.left == l)
  && (this.right == r) && (this.parent == p)
{
   this.count = c;
   this.left = l;
   this.right = r;
   this.parent = p;
}

void updateCount()
~double k, double k1, double k2:
int c, Composite ol, Composite or, Composite op,
int c1, int c2, int c3:
requires ((op!=null) ~=>
   (unpacked(op#k1 left(op.left, op.left.count)) ||
   unpacked(op#k2 right(op.right, op.right.count)))) &&
   this.parent -> op &&
   unpacked(this#1.0 count(c)) &&
   (this#0.5 left(ol, c1)) &&
   (this#0.5 right(or, c2)) &&
   (op!=null ~=> unpacked(op#k count(c3)))
ensures (this#1.0 count(c1+c2+1)) &&
   ((op!=null) ~=> (unpacked(op#k1 left(op.left,op.left.count)) ||
   unpacked(op#k2 right(op.right, op.right.count)) ) ) &&
   (op!=null ~=> unpacked(op#k count(c3)))  &&
   (this#0.0 left(this.left, this.left.count)) &&
   (this#0.0 right(this.right, this.right.count))
{
int newc;

newc = 1;
unpack(this#0.5 left(ol, c1))[op];
if (this.left != null) {
   unpack(ol#0.5 count(c1))[ol.left, ol.right,
      ol.left.count, ol.right.count];
   newc = newc + this.left.count;
```

Figure 5.41: Composite.java - cont. 1

```
  pack(ol#0.5 count(c1))[ol.left, ol.right,
    ol.left.count, ol.right.count];
}
pack(this#0.5 left(ol, c1))[op];

unpack(this#0.5 right(or, c2))[op];
if (this.right != null) {
  unpack(or#0.5 count(c2))[or.left, or.right,
    or.left.count, or.right.count];
  newc = newc + this.right.count;
  pack(or#0.5 count(c2))[or.left, or.right,
    or.left.count, or.right.count];
}
pack(this#0.5 right(or, c2))[op];
this.count = newc;
pack(this#1.0 count(newc))[ol, or, c1, c2];
}


void updateCountRec()
~ double k1, double k, double k2, double k3:
Composite opp, int lcc,
Composite ol, Composite or,
int lc, int rc:
requires unpacked(this#k1 parent()) &&
  this.parent -> opp &&
  (opp != this) &&
  ( (opp != null) ~=> (opp#k parent()) ) &&
  ( ( (opp != null) && (opp.left == this)) ~=>
    (opp#0.5 left(this, lcc))) &&
  ( ( (opp != null) && (opp.right == this)) ~=>
    (opp#0.5 right(this, lcc)))
  &&
  ((opp == null) ~=> (unpacked(this#0.5 count(lcc))))
  &&
  unpacked(this#0.5 count(lcc)) &&
  (this#0.5 left(ol, lc)) &&
  (this#0.5 right(or, rc))
ensures (this#k2 parent()) &&
  ( (opp != null) ~=> (opp#k3 parent()) )
{
  if (this.parent != null) {
    splitFrac(opp#k parent(), 2);
    unpack(opp#k/2 parent())[opp.parent, opp.count];
    unpack(opp#0.5 count(opp.count))[opp.left, opp.right,
      opp.left.count, opp.right.count];
```

Figure 5.42: Composite.java - cont. 2

```
      if (this == this.parent.right) {
        addFrac(opp#0.5 right(this, lcc),
          opp#0.5 right(opp.right, opp.right.count));
        unpack(opp#1.0 right(this, lcc))[opp.parent];
        addFrac(unpacked(this#0.5 count(lcc)),
          this#0.5 count(lcc));

        this.updateCount()[lcc, ol, or, opp, lc, rc, opp.count];
        pack(this#k2 parent())[opp, lc + rc + 1];
        pack(opp#1.0 right(this, lc + rc + 1))[opp.parent];

        this.parent.updateCountRec()[opp.parent, opp.count, opp.left,
          this, opp.left.count, lc + rc + 1];
      } else {
        addFrac(opp#0.5 left(this, lcc),
          opp#0.5 left(opp.left, opp.left.count));
        unpack(opp#1.0 left(this, lcc))[opp.parent];
        addFrac(unpacked(this#0.5 count(lcc)),
          this#0.5 count(lcc));

        this.updateCount()[lcc, ol, or, opp, lc, rc, opp.count];
        pack(this#k2 parent())[opp, lc + rc + 1];
        pack(opp#1.0 left(this, lc + rc + 1))[opp.parent];

        this.parent.updateCountRec()[opp.parent, opp.count,
          this, opp.right, lc + rc + 1, opp.right.count];
      }
    } else {
      addFrac(this#0.5 count(lcc), unpacked(this#0.5 count(lcc)));
      this.updateCount()[lcc, ol, or, opp, lc, rc, opp.count];
      splitFrac(unpacked(this#1.0 count(lcc)), 2);
      pack(this#k2 parent())[this.parent, lc + rc + 1];
    }
  }

  void setLeft(Composite l)
  ~ double k1, double k2, double k, double k3:
  requires (this != l) && (l != null) &&
    (this.left != this.parent) &&
    (l != this.parent) && (l != this.right) &&
    (this != this.right) && (this != this.left) &&
    (this#k1 parent()) && (l#k2 parent())
  ensures (this#k parent()) && (l#k3 parent())
  {
    unpack(l#k2 parent())[l.parent, l.count];
```

Figure 5.43: Composite.java - cont. 3

```
    if (l.parent==null) {
      l.parent = this;
      unpack(this#k1 parent())[this.parent, this.count];
      unpack(this#0.5 count(this.count))[null, this.right, 0,
        this.right.count];
      addFrac(this#0.5 left(null, 0), this#0.5 left(null, 0));
      unpack(this#1.0 left(null, 0))[this.parent];
      this.left = l;
      this.left.count = l.left.count;
      pack(this#1.0 left(l, l.left.count))[this.parent];
      splitFrac(this#1.0 left(l, l.left.count));
      pack(l#k2 parent())[l.parent, l.left.count];
      this.updateCountRec()[this.parent, this.count, l, this.right,
        l.left.count, this.right.count];
    } else {
      pack(l#k2 parent())[l.parent, l.count];
    }
  }

}
```

Figure 5.44: Composite.java - cont. 4

DoubleCount represents objects which have a field *val* and a field *dbl*, such that *dbl==2*val*. This property represents the invariant of objects of type Doublecount. We want to verify that this invariant is maintained by the method *increment*. The tilde sign in the specification of the `increment` method in Figure 5.45 is there to differentiate between variables `k` used for fractions and other variables that are used as parameters to predicates.

The Boogie translation of DoubleCount.java is given in Figure 5.46. We have global map variables for the fields of the class, for keeping track of which objects are packed and for the fraction corresponding to each object proposition. We have the translation of the constructor `ConstructDoubleCount`, procedures `PackOK` and UnpackOK that are being called when an object has to be packed or unpacked. In the specification of procedure `increment`, we require that all objects are packed on the entrance to the procedure. In our methodology, all the objects that are not explicitly unpacked are considered packed. The `ensures forall` and `requires forall` specifications act as frame conditions and restrict the number of objects that the Boogie verifier assumes have changed at method boundaries. We need this restriction because of the `modifies` that Boogie needs for each method: the `modifies` is very general and assumes that all the global variables have been changed, thus nullifying any previous (old) properties about those variables.

Examples SimpleCell.java, Link.java and Share.java can be seen on the GitHub website under the *src/testcases/cager/jexpr* directory. The current version of the code behind Oprop can be found on GitHub [5]. We created the SimpleCell example to illustrate how we add fractions in the cases when we need a larger fraction to an object proposition than we currently have. This

```
class DoubleCount {
  int val; int dbl;
  predicate OK() = exists int v, int d :
    this.val -> v && this.dbl -> d && d == 2*v
  void increment()
    ~double k:
    requires this#k OK()
    ensures this#k OK()
      { unpack(this#k OK());
      this.val = this.val + 1;
      this.dbl = this.dbl + 2;
      pack(this#k OK()); } }
```

Figure 5.45: DoubleCount.java

```
type Ref;
const null:  Ref; var val:  [Ref]int; var dbl:  [Ref]int;
var packedOK: [Ref] bool; var fracOK: [Ref] real;
procedure ConstructDoubleCount
      (val1 :int, dbl1 :int, this:  Ref);
  ensures (val[this] == val1) && (dbl[this] == dbl1);
procedure PackOK(this:Ref);
  requires packedOK[this]==false &&
      ((dbl[this]==(2*val[this])));
procedure UnpackOK(this:Ref);
  requires packedOK[this] &&(fracOK[this] > 0.0);
  ensures ((dbl[this]==(2*val[this])));
procedure increment(this:Ref)
  modifies dbl,packedOK,val;
  requires packedOK[this] && (fracOK[this] > 0.0);
  ensures packedOK[this] && (fracOK[this] > 0.0);
  requires (forall x:Ref ::  packedOK[x]);
  ensures (forall x:Ref::
      (packedOK[x] == old(packedOK[x])));
    { call UnpackOK(this); packedOK[this] := false;
    val[this]:=val[this]+1; dbl[this]:=dbl[this]+2;
    call PackOK(this); packedOK[this] := true;}
```

Figure 5.46: doublecount.bpl

example is explained in detail in Section 4.3.

The example Link illustrates how we deal with predicates that have parameters. This is where we implemented the `Range` predicate which has two parameters. When constructing an object of type `Link` we write the following line:

```
Link l1 = new Link(Range(0,10)[3, null])(3, null);
```

When the object `l1` is created, we have to specify the predicate that holds for it in case the object becomes shared (or aliased) in the future. Since the predicate `Range` has two existentially quantified variables and Boogie cannot successfully instantiate existential variables, we give the witnesses `3` and `null` for the two variables `int v, Link o` existentially quantified in the body of the predicate `Range`.

We created the Share example to exemplify objects that have a reference to a *shared* common object. In the `main()` method of this example there are two `Share` objects and each has a fraction of 0.1 to a common object `dc0` of type `DoubleCount`. Since the `Share.java` file uses the file `DoubleCount.java`, we specify to Oprop that there are 2 files that need to be translated into Boogie and we give the names of the files one after the other in the list of arguments; the arguments for Oprop when running this example would be:
`2 DoubleCount.java Share.java`. Oprop will produce a single output file because Boogie needs all its input code to be in a single file in order for it to do the verification.

The result of the automatic translation of these examples can also be seen on GitHub under the same directory *src/testcases/cager/jexpr*. In the same directory one can see the **.interm* file corresponding to each *.bpl file, which is the intermediate representation of the initial Oprop program into an abstract syntax tree, annotated with the type of each node.

Chapter 6

Related Work

This chapter first does a general comparison to existing relevant related work. In Section 6.1 we go into detail about the differences between our object proposition verification versus considerate reasoning and concurrent abstract predicates.

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

Bierhoff and Aldrich [19] 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 [30] system 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 [25] 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. Similarly, in [24] the fractional permissions are treated as in Boyland's work: when a fraction is 1 there is write access, but when a fraction is less than 1 there is only read access to the shared resource. This is very different from our work because we allow multiple clients to write to a common resource even if the fractional permission is less than 1. The trick is that those clients can only write to the resource if they maintain an invariant on the resource.

Boogie [16] 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.* [44] 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 specific for their library. Our methodology is a general one that can potentially be used for verifying any object-oriented program.

Nanevski *et al.* [56] developed Hoare Type Theory (HTT), which combines a dependently typed, higher-order language with stateful computations. While the HTT language is meant to be an expressive and explicitly annotated internal language which offers a semantic framework for elaborating more practical external languages and verification tools, our work targets Java-like languages and does not have the cognitive complexity overhead of higher-order logic. The more difficult a verification system is to understand, the higher the entry barrier is for the programmers that want to use it.

Summers and Drossopoulou [68] 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 [16], facilitating the automatic verification of the Composite pattern specification. Despite the fundamental differences in underlying methodology (visible-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.* [28] 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 [61] and Dinsdale-Young *et al.* [31]. 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 [40].

There exists a set of verification methodologies for object-oriented programs in a concurrent setting [31, 39, 42, 47]. 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 single-threaded setting, there is no such lock. Even in multi-threaded 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 that we are considering.

There are a number of tools [13], [51], [26] that verify concurrent programs using model checking techniques. They prove that a program running on a relaxed (or weak) memory model is sequentially consistent, meaning that there is a total order on the instructions of all processes and the per-process program order. A weak memory model would be a model where not all of the following orderings are preserved: write-to-read order (meaning that store instructions are not reordered after load instructions), write-to-write order, read-to-read/write order and write atomicity (meaning that all writes to a location should appear to all processors to have occurred in the same order). Model checking has the drawback that state explosion can happen during the verification. Nevertheless, it is a technique that is orthogonal to the static verification described in this thesis and is used successfully in many scenarios.

The work by Kassios and Kritikos [43] is done in a concurrent setting, by using invariants and permissions. Their idea is to extend separation logic by adding backpointer conditions (state conditions that involve heap objects that point to the reachable part of the heap). They tackle the problem of observational disjointness: 'the structure pretends that it supports mutually disjoint mutable sequences of integers, even though it uses data sharing under the hood, to enhance performance'. By extending separation logic with backpointer conditions and a reference counting mechanism, the result is similar to abstract predicates: the ability to hide shared data and enhance modularity (whether it is at the level of methods as in our work, or at the thread level, as in their work). While we use fractional permissions to 'loosen the heap disjointness requirement', they use counting permissions. ('A counting permission is a natural number n, or -1. At any given execution time, there is one thread that holds a non-negative counting permission n to a heap location and n threads that hold a -1 counting permission.') Although the work of Kassios and Kritikos is done for concurrent programs, their solution is orthogonal to our object propositions.

Peter Müller *et al.* [53] created the Viper toolchain, that also uses Boogie and Z3 as a back-end, and can reason about persistent mutable state, about method permissions or ownership, but they also need a full permission to modify shared data. There are other formal verification tools for object oriented programs, such as KeY [12], VCC [28] or Dafny [48], that implement other methodologies.

6.1 Detailed Comparison to Related Work

In Figure 6.1 we present the code and object propositions specifications for a class $Link$ that is used to create a linked list. We want all the elements of the list to be integers in the range [0,10]. The method $addModulo11$ adds an integer to each element of the list, but it makes sure to take the modulo 11 of the result, such that the elements of the list remain in the range [0,10]. In the following two sections we present a detailed comparison of object propositions versus considerate reasoning and, of object propositions versus concurrent abstract predicates.

6.1.1 Considerate Reasoning

A solution that uses considerate reasoning has been given to the verification of the Composite pattern in [68]. The considerate-style specification in Java is given in Figure 6.2.

The solutions for the Composite pattern using object propositions or considerate reasoning are similar because for both the burden lies in coming up with the appropriate specifications. When using object propositions, the programmer has to come up with exactly the right predicates. When using considerate reasoning, the programmer has to come up with the right invariants, to

```
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]

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

Figure 6.1: Link class and range predicates using object propositions

declare certain invariants as structural and to define the *broken* declarations along with method specifications. Note the analogy between the broken invariants in considerate reasoning and unpacked predicates in our system. In considerate reasoning the broken invariants have to be restored by the end of the method $addToTotal$ in Figure 6.2; in our system unpacked predicates have to be made to hold again before they can be packed back up. Both broken invariants in considerate reasoning, and unpacked predicates in our system, allow the specifier to express which are the predicates that cannot be relied on at a certain moment because their fields are being updated. This is essential to verifying the Composite pattern because by showing which predicates (invariants) are broken, the specification is kept consistent with the implementation when a Composite object is being updated.

Structural invariants in considerate reasoning are very similar to object proposition invariants. 'Structural invariants can only be violated within unreliable blocks which explicitly declare that they might be. Structural invariants may be depended on to accurately predict the concerns of a field update only outside the scope of such blocks. Furthermore, any structural invariants violated within an unreliable block should be re-established by the end of the block'. In the same way object proposition invariants can be broken inside methods, but they have to be re-established by the end of the methods and before they are packed back again. The difference between structural invariants and other invariants that can be broken in considerate reasoning is that structural invariants can be broken only for a handful of consecutive statements at a time, while other invariants can be broken for arbitrarily long code fragments (so long as no method boundaries are reached). Intuitively these 'structural invariants' are broken just while the necessary field updates can all be made, to modify the intended parent-components relation in the Composite example. There is an interesting relationship between structural invariants and our use of a resource-based linear theory: structural invariants are broken when the implementation updates the resources (the fields). Thus structural invariants are mirrors of the state of resources

```
class Composite {
      private Composite parent;
      private Composite[] comps;
      private int count=0;
      private int total=1;

// Inv1(o):  $1 \leq o.total \wedge 0 \leq o.count$
// Inv2(o):  $o.total = 1 + \sum_{0 \leq i \leq o.count} o.comps[i].total$
// Inv3(o):  $\forall 0 \leq i < o.count : o.comps[i].parent = o$
// Inv4(o):  $o.parent \neq null \Rightarrow \exists 0 \leq i < o.parent.count : o.parent.comps[i] = o$
// Inv5(o):  $\forall 0 \leq i \neq i < o.count : o.comps[i] \neq o.comps[j]$

// requires:  $c \neq null;$
// requires:  $c.parent = null;$

  public void add(Composite c) {
    comps[count] = c;
    count++;
    c.parent = this;
    addToTotal(c.total);
    }

// broken:  $Inv2(this)$
// requires:  $this.total + p = 1 + \sum_{0 \leq i \leq o.count} o.comps[i].total$
  public void addToTotal(int p) {
    total+=p;
    if (parent!=null) parent.addTotal(p);
    }

 }
```

Figure 6.2: Composite example using considerate reasoning

in the program.

While considerate reasoning is capable of giving a solution to the verification of the composite pattern, there are example programs that it cannot verify. This is because the invariants of the class have to hold for every instance of the class, the invariants are not optional and thus the considerate reasoning methodology becomes more rigid than we would want.

Below we present the Link class and Range predicates using considerate reasoning. Considerate reasoning does not have abstract predicates and it is not as flexible as one would want. In the example below $Inv1$ could be extended in the following way: $Inv1(o, x, y) : val \geq x \wedge val \leq y$. This would extend considerate reasoning and give it more flexibility, but it would also pose problems because there would be a family of abstract predicates that should hold for the objects of class $Link$. This goes against the principles of considerate reasoning where the same invariant has to hold for all objects of a certain class.

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

Inv1(o):  $val \geq 0 \wedge val \leq 10$

  void addModulo11(int x)
    {val = (val + x)% 11;
     if (next!=null) {next.addModulo11(x);}
    }
 }
```

The problem is that the invariant $Inv1$ has to hold for every instance of the class $Link$ and thus it constrains the instances of class ***Link*** more than we would want. Considerate reasoning wouldn't be able to specify the example in Section 2.3. This is because in that example there are two objects of type $Link$, one depicting a simulator for a queue of jobs containing large jobs (size>10) and another one depicting a simulator containing small jobs (size<11). The class $Link$ would need two invariants $I1(o) : val \geq 0 \wedge val \leq 10$ and $I2(o) : val \geq 11 \wedge val \leq 100$ such that some objects satisfy $I1$ while others satisfy $I2$. This is not expressible in considerate reasoning because all objects would have to satisfy a single invariant, either $I1$ or $I2$. Object propositions are able to specify this example in Subsection 2.5.2 because different object propositions and different invariants can be defined for objects of the same class.

6.2 Alternate Verification of Producer-Consumer Example Using Considerate Reasoning

One idea is to incorporate the $Range$ predicate in the $Control$ class of the simulator of queues of jobs example (which would specify the range of the contents of each of the links) in order to make that example verifiable using considerate reasoning.

The invariant in the considerate reasoning variant would be

```
Inv1(control):
```

$$\begin{aligned}&control.prod1.startSmallJobs.val > 0 \wedge\\ &control.prod1.startSmallJobs.val < 10 \wedge\\ &control.prod2.startLargeJobs.val > 10 \wedge\\ &control.prod2.startLargeJobs.val < 100 \wedge\\ &control.cons1.startJobs.val > 0 \wedge\\ &control.cons1.startJobs.val < 10 \wedge\\ &control.cons2.startJobs.val > 10 \wedge\\ &control.cons2.startJobs.val < 100\end{aligned}$$

This would be the invariant of the `Control` class, which would have to be satisfied both in the pre- and postcondition of the method `makeActive`.

One can see that although we are able to write the invariant `Inv1` in the considerate reasoning style, this invariant is not as modular as the predicates written using object propositions. When changes will be made to the `Control` class, the invariant `Inv1` above will also have to be completely rewritten, which is a major inconvenience.

6.2.1 Concurrent Abstract Predicates

In [31], Parkinson *et al.* 'present a program logic for reasoning abstractly about data structures that provides a fiction of disjointness and permits compositional reasoning.' By 'fiction of disjointness', Parkinson *et al.* mean that the predicates can be used as if each abstract predicate represents disjoint resources, whereas in fact resources are shared between predicates. 'The internal details of a module are completely hidden from the client by concurrent abstract predicates (CAP). They reason about a module's implementation using separation logic with permissions, and provide abstract specifications for use by client programs using concurrent abstract predicates'.

The great benefit of CAP is that it allows fine-grained abstraction in a concurrent environment. Their strength comes from combining three lines of research: abstract predicates for abstracting the internal details of a module or class; deny-guarantee for reasoning about concurrent programs; and context logic for fine-grained reasoning at the module level.

The specification of the $Link$ example using CAP is given below.

Range(x, l,u) $\equiv\ \exists r, k.$

$$\boxed{\exists v, n.(x.val \to v \wedge l \le v \le u) * x.next \to n \wedge n \neq null \Rightarrow Range(n, l, u)}^{r}_{I(x,l,u)}$$

$* \ [UPD]^r_k * [TAIL]^r_k$

I(x,l,u) $\equiv$ UPD: $x.val \to \_ \rightsquigarrow x.val \to v \wedge l \le v \le u$

TAIL: $x.next \to \_ \rightsquigarrow x.next \to n * Range(n, l, u)$

In this example concurrent abstract predicates are able to accomplish what object propositions do by defining the shared region assertion

$$\boxed{\exists v, n.(x.val \to v \wedge l \le v \le u) * x.next \to n \wedge n \neq null \Rightarrow Range(n, l, u)}^{r}_{I(x,l,u)}$$

and the permission assertions $[UPD]^r_k$ and $[TAIL]^r_k$. CAP works in a concurrent setting; thus the shared state is indivisible so that all threads maintain a consistent view of it. Although object propositions are defined for single threaded programs, one of our main concerns is modularity.

Achieving modular verification in one thread is analogous to achieving correct verification in an environment where multiple threads interact.

The shared region assertion specifies that there is a shared region of memory, identified by label r, and that the entire shared region satisfies $\exists v, n.(x.val \rightarrow v \wedge l \leq v \leq u) * x.next \rightarrow n \wedge n \neq null \Rightarrow Range(n, l, u)$. The possible actions on the state are declared by the environment $I(x, l, u)$.

The permission assertions $[UPD]_k^r$ and $[TAIL]_k^r$ specify that the thread has permission k to perform actions UPD and $TAIL$ over region r, provided the action is declared in the environment. The permission k can be the fractional permission $k \in (0, 1)$, denoting that both the thread and the environment can do the action, or the full permission, $k = 1$, denoting that the thread can do the action but the environment cannot. This is very similar to the way we use fractional permissions in our system.

Concurrent abstract predicates can indeed do the work that object propositions can do, but they are more difficult to automate than object propositions. This is because the logical framework of CAP is quite complicated: the programmer would have to understand and implement (in the program specifications) abstract predicates, deny-guarantee logics and context logics. Another reason why automation of CAP is difficult is the use of resource permissions: permissions must ensure that a predicate is self-stable (that is, immune from interference from the surrounding environment). To automatically show that a predicate is self-stable seems to be a difficult task.

In [31], Parkinson *et al.* first present an implementation of a concurrent set using a single global lock. This meant that only a single thread at once could access the entire set. They go on to refine the locking strategy by presenting the verification of a set implementation that uses a sorted list with one lock per each node in the list. This allows many threads to access the list at once, ensuring that the threads do not violate the safety properties of other threads. Since object propositions work in a more well-behaved setting where we do not have concurrency, we are able to call Unpack (the equivalent of Unlock from [31]) twice on the same object, as long as the two predicates that we are unpacking at once refer to disjoint fields of the object that we are referring to. In [31] a thread can only release a lock once without locking it again and this restricts the verification scenarios that CAP can be applied to, while object propositions have more flexibility because they do not deal with concurrency.

Chapter 7

Limitations of the Object Propositions Verification System

The limitations of our system are the following:

1. The programmers that want to use object propositions need to become familiar with linear logic and with the semantics of Oprop. Although we do not think it is difficult to understand the theory of object propositions, programmers might benefit from tutorials or interactive presentations if we want them to use it. This is a minor limitation that slightly affects the practicality claim of the thesis because there is a hurdle that programmers need to overcome in order to use our verification system, but it is the same kind of hurdle that they would have to surpass when they need to learn a new programming language.
2. Programmers might need to add many annotations in their programs in order to prove their desired properties. For example, they need to specify in the Java code the right instantiation value whenever there is a variable that is existentially quantified in the Oprop annotations. Usually it is not difficult to know what witness to chose for a quantified variable, but it does require extra effort. This limitation appears in our Oprop tool because the Boogie tool (which in turn uses the SMT solver Z3) that we use as a back end is not capable of reasoning when presented with existentially quantified variables. This is a problem for all SMT solvers, but as the research progresses in that field, the need for the annotations in Oprop will decrease. This is a limitation that is intrinsic to our approach because it is tightly connected to the tools that we chose as a base for our implementation and to the fact that our formalism uses existentially quantified variables.
3. We only allow multiple object propositions to be unpacked at the same time if the objects are not aliased, or if the unpacked propositions cover disjoint fields of a single object. We do this to avoid inconsistencies and preserve the soundness of our system, even if it means that it makes the theory of object propositions more complicated. This restriction is a fundamental part of the approach that might be considered a major shortcoming because it limits the expressivity of the verification language, but it is a necessary restriction for soundness.
4. The specifications can become too verbose in some cases, depending on the properties that we want to prove. Usually if the properties are more difficult to prove, the specifications

will be more verbose. We think it is natural that the specifications become more complicated as the properties that the programmer wants to verify become more difficult, but we still see it as a drawback for the adoption of Oprop in industry. Although this verbosity does not affect the claims of the thesis, if we ever wanted to commercialize Oprop the first step would be to reduce the number of annotations that the programmer has to write. The Oprop tool would need to be able to infer more annotations at each point of the program, based on the existing annotations and the surrounding program statements. We think this is a worthwhile effort, but a major one nonetheless.

5. Since Java is a garbage collected language, we have to be able to remove unrequired assertions from the linear context. The solution is to add an annotation statement *release(r@1 Pred($\bar{t}$))*, that the programmer can write as a program statement. Only object propositions that have a fraction of 1 should be removed from the system, when the programmer knows that an object is not needed anymore and it will be garbage collected. This relies on the programmer being sure that an object proposition is not needed from that moment on. If the programmer is wrong and later on that object proposition is needed, then the verification will fail at the point where the discarded object proposition is actually needed. The objects in our examples are not garbage collected until the end of the programs and this scenario does not affect us. We leave the implementation of this feature as future work. The Oprop implementation of this feature requires a moderate amount of work. Nevertheless it is important because it strenghtens the modularity claims of the thesis: object propositions that are not needed in the verification are discarded and thus the linear contexts do not carry all the object propositions that were used so far, making sure that this information is hidden in the client code and not accessed by mistake.

6. We have not added to Oprop features such as inheritance, casting or dynamic dispatch that are important but are handled by orthogonal techniques. Hence a limitation of our system is that it cannot verify programs that use these features. For example, if we wanted to support inheritance we would define some predicates in the base class, and the same predicates in all the derived classes would be redefined knowing that they have to be at least as strong as the predicates in the base class. We would need to define the proof rules to support inheritance, to show that the system is still sound, implement the translation into Boogie that would allow the programmers to reason about programs using inheritance, and finally verify an example program that uses inheritance. Although this is a major limitation of our work, it can be addressed with additional work. The fact that not all features of the Java language are supported by Oprop marginally affects the claims of the thesis, since our verification system is not as complex and general as it could be. We believe that it would be the work of at least another Ph.D. thesis to be able to support all the features of the Java language.

   A discussion is warranted here: the need to support inheritance is debatable. The *Design Patterns: Elements of Reusable Object-Oriented Software* book by Ralph Johnson *et al.* [41] discusses at length replacing implementation inheritance (*extends*) with interface inheritance (*implements*). One of the problems with inheritance is that explicit use of concrete class names locks the programmer into specific implementations, making future changes unnecessarily difficult. We have already added theoretical support for interfaces and we have also manually translated examples that use interfaces, such as our program implementing the state design pattern. Adding full support for interfaces is the natural next step that would improve the Oprop tool.

7. Our Oprop tool does not implement all the necessary features to automatically verify the state, proxy and flyweight design paterns, such as interfaces, base classes or derived classes. Nonetheless, we have manually translated our examples for each of these patterns into Boogie and verified them with the Boogie tool. An interesting piece of future work could be to implement these features into Oprop and automatically verify the instances of the three design patterns. This limitation is minor and can be addressed by extending the implementation of Oprop to support interfaces.

Chapter 8

Future Work

Future work can be divided into two categories: improvements that are natural extensions of this thesis and projects that would require significantly more work.

In the first category we have the work needed to augment the features of the Oprop language, which would lead to the improvement of the Oprop tool. Now Oprop is a variant of Featherweight Java. In order for the Oprop tool to be used in industry, Oprop would have to be as close to Java as possible.

In order to be able to verify a number of practical programs, we have added the necessary features in the implementation of our tool. In the future we would like other contributors to be able to add features to the target language Oprop. If someone wanted to add a new feature to Oprop, they would first have to change the lexer and parser parts of the tool located in the JExpr.java file. The next step would be to add the necessary methods to the class ContextVisitor.java, which traverses the generated abstract syntax tree in order to add types. The last step would be to modify the class BoogieVisitor.java that represents the implementation of the translation rules for each node in the AST. We have written many comments in the code of the Oprop tool that we believe will be very useful for future contributors. As mentioned in the previous chapter, supporting the automatical verification of the three design patterns state, proxy and flyweight is a natural continuation of our work.

When verifying a program using Oprop, if a property is not satisfied it could be because the specifications are not quite right or because there is an actual error in the programmer's code. In order to help the programmer debug the problem, we would like to add an *assert* statement to the language Oprop. The Boogie language already has an *assert* statement that can be used for debugging and we would like to allow the programmers to access that statement through the *assert* in the Oprop language. This would allow the programmer to test assertions at any point in their program and see if they hold. This is a minor implementation effort and we think would improve the usability of the Oprop tool.

In the second category there is the work needed for Oprop to be extended for multi-threaded programs. Oprop can only verify single-threaded programs, but we hope that it will be a starting point for writing a tool that verifies multi-threaded Java programs. There are numerous programs that can be verified in the Oprop system, but one particularly interesting area is privacy and security applications. The modularity focused approach of Oprop can be especially valuable for those applications, since it provides the information hiding needed for these applications. Some particularly fascinating questions are: how can the verification be done more automatically, how can we reduce the number of annotations that the programmers have to write and how can the tool infer these annotations? These are questions that require significant work in order to be answered.

A possible approach for the extension to concurrency is for us to add *synchronized* methods for the times when an object is unpacked until it is packed again. We would have a different synchronized method for the unpacking/packing of each predicate. Such a method would be similar to the following code: `public synchronized void unpackAndPackPredA(double k){ unpack(this@k PredA()); ...;pack(this@k PredA());}`. We would make the synchronization be reentrant because if there are two predicates that are packed for the same object, but each predicate refers to different fields of the object, we would like to be able to unpack both predicates at the same time if the need arises. We would not want to unpack the same predicate twice for the same object, but that would not be possible because by the time we want to unpack the same object the second time, it would already be unpacked and the conditions of unpacking it (i.e., that it needs to be packed) would not be satisfied.

This thesis sets the stage for future work in verification by presenting a modular way of verifying single threaded object-oriented programs. Our work can be used as the basis for verifying security and privacy applications that benefit from the hiding of information in specifications. The computing of the future is cloud computing, automation/robotics on a large scale, increased networking, Internet of Things. Whether we need to verify that the software as a service that was sold to a customer actually performs as expected, that an Amazon robot interacts well with the other robots, or that the smart devices in our homes respect our wishes, object propositions (or future systems inspired by them) could be used to achieve our formal verification goals.

Chapter 9

Conclusion

In conclusion, we created a verification system that is used for single-threaded object-oriented programs. The object proposition methodology uses abstract predicates and fractional permissions. Developers will first annotate their programs using object propositions and then use the Oprop tool to automatically verify their annotated programs. We present the theory of object propositions in Section 3.1 and the proof that our system is sound in Section 3.5.

In Chapter 4 we showed the strategy that we used in the implementation of our Oprop tool, how we encoded our Oprop language into first order logic and why this encoding is sound. We hope that by using Oprop developers will be able to prove that their programs satisfy their specifications.

We have studied how instances of a number of design patterns can be verified using object propositions: the composite design pattern in Section 5.4, the producer-consumer in Section 2.3, the subject-observer design pattern in Section 2.5.1, and the state, proxy and flyweight design patterns in Chapter 5. Throughout the thesis, all these examples were compared to existing approaches such as the classical invariant technique, verification using separation logic, ramified frame properties, and we showed that object propositions provide a more modular verification in some cases. Each instance program of these design patterns had particular properties that we were able to verify using object propositions. This demonstrates that our methodology is widely applicable and general, and that it has enough features to verify various properties of a number of very different design patterns. We look forward to other people potentially using object propositions to verify instances of other design patterns.

Bibliography

[1] https://en.wikipedia.org/wiki/Composite_pattern. . 2.6

[2] http://www.cs.cmu.edu/~lnistor/composite-partial.pdf. . 5.4, 5.5

[3] https://en.wikipedia.org/wiki/Flyweight_pattern. . 5

[4] https://sourcemaking.com/design_patterns/flyweight. . 5.3.1

[5] https://github.com/ligianistor/Oprop. 5.6

[6] http://www.java.com/en/about/. . 1

[7] http://code.google.com/p/syper/. . 4, 4.4

[8] https://en.wikipedia.org/wiki/Proxy_pattern. 5

[9] https://en.wikipedia.org/wiki/State_pattern. 5

[10] https://en.wikipedia.org/wiki/Proxy_pattern{#}/media/File: Proxy_pattern_diagram.svg. . 5.2.1

[11] https://en.wikipedia.org/wiki/State_pattern{#}/media/File: State_Design_Pattern_UML_Class_Diagram.svg. . 5.1.1

[12] Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H. Schmitt, and Mattias Ulbrich, editors. *Deductive Software Verification - The KeY Book - From Theory to Practice*, volume 10001 of *Lecture Notes in Computer Science*. Springer, 2016. ISBN 978-3-319-49811-9. doi: 10.1007/978-3-319-49812-6. URL http://dx.doi.org/10.1007/978-3-319-49812-6. 1.4, 6

[13] Jade Alglave, Daniel Kroening, Vincent Nimal, and Michael Tautschnig. Software verification for weak memory via program transformation. In *ESOP*, pages 512–532, 2013. URL http://arxiv.org/abs/1207.7264. arXiv version available. 6

[14] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *AFIPS '67 (Spring)*, pages 483–485, New York, NY, USA, 1967. ACM. doi: 10.1145/1465482.1465560. URL http://doi.acm.org/10.1145/1465482.1465560. 1

[15] 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. 2.1

[16] 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. 1.3, 2.5.1, 4.2, 6

[17] Yves Bertot and Pierre Castran. *Interactive Theorem Proving and Program Development: Coq'Art The Calculus of Inductive Constructions*. Springer Publishing Company, Incorporated, 1st edition, 2010. ISBN 3642058809, 9783642058806. 1.2.2

[18] Kevin Bierhoff. Api protocol compliance in object-oriented software. In *PhD thesis. Technical Report CMU-ISR-09-108*, 2009. 4.1

[19] Kevin Bierhoff and Jonathan Aldrich. Modular typestate checking of aliased objects. In *OOPSLA*, pages 301–320, 2007. 2.1, 6

[20] Kevin Bierhoff and Jonathan Aldrich. Permissions to specify the composite design pattern. In *Proc of SAVCBS 2008*, 2008. 5.4, 5.4

[21] Alex Blewitt, Alan Bundy, and Ian Stark. Automatic verification of design patterns in java. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, ASE '05, pages 224–232, New York, NY, USA, 2005. ACM. ISBN 1-58113-993-4. doi: 10.1145/1101908.1101943. URL `http://doi.acm.org/10.1145/1101908.1101943`. 1.2.2

[22] Barry W. Boehm and Victor R. Basili. Software defect reduction top 10 list. In *Computer*, pages 135–137, 2001. 1

[23] Barry W. Boehm and Philip N. Papaccio. Understanding and controlling software costs. In *Understanding and Controlling Software Costs*, pages 1462–1477, 1988. 1

[24] Richard Bornat, Cristiano Calcagno, Peter O'Hearn, and Matthew Parkinson. Permission accounting in separation logic. In *POPL*, pages 259–270, 2005. 5.3, 6

[25] John Boyland. Checking interference with fractional permissions. In *Static Analysis Symposium*, pages 55–72, 2003. 1, 1.3, 1.4, 2.1, 2.4, 3.2, 6

[26] Sebastian Burckhardt, Rajeev Alur, and Milo M. K. Martin. Checkfence: Checking consistency of concurrent data types on relaxed memory models. *SIGPLAN Not.*, 42(6):12–21, June 2007. ISSN 0362-1340. 6

[27] Iliano Cervesato, Joshuas. Hodas, and Frank Pfenning. Efficient resource management for linear logic proof search. In *Proceedings of the 5th International Workshop on Extensions of Logic Programming*, pages 67–81. Springer-Verlag LNAI, 1996. 4.1

[28] Ernie Cohen, Michal Moskal, Wolfram Schulte, and Stephan Tobies. Local verification of global invariants in concurrent programs. In *CAV*, pages 480–494, 2010. 2.1, 4.2, 6

[29] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. ETAPS, pages 337–340, Berlin, Heidelberg, 2008. ISBN 3-540-78799-2, 978-3-540-78799-0. URL `http://dl.acm.org/citation.cfm?id=1792734.1792766`. 4.2

[30] Robert DeLine and Manuel Fähndrich. Typestates for objects. In *ECOOP*, pages 465–490, 2004. 2.1, 2.4, 2.5.1, 6

[31] Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, Matthew Parkinson, and Viktor Vafeiadis. Concurrent abstract predicates. In *ECOOP*, pages 504–528, 2010. 2.1, 6, 6.2.1

[32] Dino Distefano and Matthew J. Parkinson. jStar: Towards Practical Verification for Java. In *OOPSLA*, pages 213–226, 2008. 2.3, 2.3

[33] Manuel Fähndrich and Robert DeLine. Adoption and focus: practical linear types for imperative programming. In *PLDI*, pages 13–24, 2002. 1.4, 2.4

[34] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994. 5.4

[35] Jean-Yves Girard. Linear logic. *Theor. Comput. Sci.*, 50(1):1–102, 1987. 2.5.1

[36] S. Heule, I. T. Kassios, P. Müller, and A. J. Summers. Verification condition generation for permission logics with abstract predicates and abstraction functions. In *ECOOP*, 2013. 4.3.2

[37] C. A. R Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, 17:549–557, 1974. 2.1

[38] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: a minimal core calculus for Java and GJ. pages 132–146, 2001. 3

[39] Bart Jacobs and Frank Piessens. Expressive modular fine-grained concurrency specification. In *POPL*, pages 271–282, 2011. 6

[40] Bart Jacobs, Jan Smans, and Frank Piessens. Verifying the composite pattern using separation logic. In *Proc of SAVCBS 2008*, 2008. 2.2, 5.4, 6

[41] Ralph Johnson, John Vlissides, Richard Helm, and Erich Gamma. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995. 2.2, 6

[42] J.Smans, B.Jacobs, and F.Piessens. Vericool: An automatic verifier for a concurrent object-oriented language. In *FMOODS*, pages 220–239, 2008. 6

[43] Ioannis T. Kassios and Eleftherios Kritikos. A discipline for program verification based on backpointers and its use in observational disjointness. In *ESOP*, volume 7792 of Lecture Notes in Computer Science, pages 149–168, 2013. 6

[44] Neel R. Krishnaswami, Lars Birkedal, and Jonathan Aldrich. Verifying event-driven programs using ramified frame properties. In *TLDI '10*, pages 63–76, 2010. 2.2, 2.2.1, 2.3, 6

[45] Neelakantan R. Krishnaswami, Jonathan Aldrich, Lars Birkedal, Kasper Svendsen, and Alexandre Buisse. Design patterns in separation logic. In *Proceedings of the 4th International Workshop on Types in Language Design and Implementation*, TLDI '09, pages 105–116, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-420-1. doi: 10.1145/1481861.1481874. URL `http://doi.acm.org/10.1145/1481861.1481874`. 1.2.2

[46] Gary T. Leavens, K. Rustan M. Leino, and Peter Müller. Specification and verification challenges for sequential object-oriented programs. *Form. Asp. Comput.*, 19(2):159–189, June 2007. ISSN 0934-5043. 2.6, 5.4

[47] K. Rustan Leino and Peter Muller. A basis for verifying multi-threaded programs. In *ESOP*, pages 378–393, 2009. 4.2, 6

[48] K. Rustan M. Leino. Dafny: an automatic program verifier for functional correctness.

In *LPAR'10 Proceedings of the 16th international conference on Logic for programming, artificial intelligence, and reasoning*, pages 348–370, 2010. 4.2, 4.3.1, 6

[49] K. Rustan M. Leino and Peter Müller. Object invariants in dynamic contexts. In *In ECOOP*, 2004. 2.1

[50] K.R.M. Leino. This is boogie 2. Manuscript KRML 178. 2008. 4.5

[51] Feng Liu, Nayden Nedev, Nedyalko Prisadnikov, Martin Vechev, and Eran Yahav. Dynamic synthesis for relaxed memory models. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 429–440, 2012. ISBN 978-1-4503-1205-9. 6

[52] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, second edition, 1997. 2.1

[53] P. Müller, M. Schwerhoff, and A. J. Summers. Viper: A verification infrastructure for permission-based reasoning. In B. Jobstmann and K. R. M. Leino, editors, *VMCAI*, volume 9583 of *LNCS*, pages 41–62. Springer-Verlag, 2016. 6

[54] Peter Müller. Modular specification and verification of object-oriented programs. In *Lecture Notes in Computer Science*, volume 2262, 2002. 1.3

[55] Peter Müller, Arnd Poetzsch-Heffter, and Gary T. Leavens. Modular invariants for layered object structures. In *Science of Computer Programming*, volume 62(3), pages 253–286, 2006. 1.3

[56] 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. 6

[57] Ligia Nistor and Jonathan Aldrich. Using machine learning in the automatic translation of object propositions. In *AI4FM*, 2014. 5.6

[58] S. Owre, J. M. Rushby, , and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, jun 1992. Springer-Verlag. URL `http://www.csl.sri.com/papers/cade92-pvs/`. 4.3.1

[59] S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. Pvs language reference version 2.4. 2001. 4.3.1

[60] Matthew Parkinson. Class invariants: The end of the road? (position paper). In *IWACO*, 2007. 2.1

[61] Matthew Parkinson and Gavin Bierman. Separation logic and abstraction. In *POPL*, pages 247–258, 2005. 1.4, 2.1, 2.4, 6

[62] D.L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15:1053–1058, 1972. 1

[63] Frank Pfenning. `http://www.cs.cmu.edu/~fp/courses/15816-s12/`. 2012. 3.5.2, 4.5

[64] Jason Reed. A hybrid logical framework. In *PhD thesis. Technical Report CMU-CS-09-*

155, 2009. 4.3.2

[65] John Reynolds. Separation logic: A logic for shared mutable data structures. pages 55–74. IEEE Computer Society, 2002. 2.3

[66] Stan Rosenberg, Anindya Banerjee, and David A. Naumann. Local reasoning and dynamic framing for the composite pattern and its clients. In *Proceedings of the Third international conference on Verified software: theories, tools, experiments*, VSTTE'10, pages 183–198, Berlin, Heidelberg, 2010. Springer-Verlag. ISBN 3-642-15056-X, 978-3-642-15056-2. URL `http://dl.acm.org/citation.cfm?id=1884866.1884887`. 5.4

[67] Jan Smans, Bart Jacobs, and Frank Piessens. Implicit dynamic frames: Combining dynamic frames and separation logic. In *Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming*, Genoa, pages 148–172, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 978-3-642-03012-3. doi: http://dx.doi.org/10.1007/978-3-642-03013-0_8. URL `http://dx.doi.org/10.1007/978-3-642-03013-0_8`. 2.1

[68] 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. 1.3, 2.5.1, 5.1.1, 5.4, 6, 6.1.1

[69] Alexander J. Summers, Sophia Drossopoulou, and Peter Müller. The need for flexible object invariants. In *IWACO*, 2009. 2.1

[70] Roger Wolff, Ronald Garcia, Éric Tanter, and Jonathan Aldrich. Gradual typestate. In *ECOOP*, pages 459–483, 2011. 3.4