Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών
και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής
και Υπολογιστών

Υλοποίηση μιας μεθοδολογίας αναλλοίωτων βασισμένη σε backpointers

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΕΛΕΥΘΕΡΙΟΣ ΚΡΗΤΙΚΟΣ

Επιβλέπων : Νικόλαος Σ. Παπασπύρου
Επίκ. Καθηγητής Ε.Μ.Π.

Αθήνα, Σεπτέμβριος 2012

Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών
και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής
και Υπολογιστών

Υλοποίηση μιας μεθοδολογίας αναλλοίωτων βασισμένη σε backpointers

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΕΛΕΥΘΕΡΙΟΣ ΚΡΗΤΙΚΟΣ

Επιβλέπων : Νικόλαος Σ. Παπασπύρου
Επίκ. Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 19η Σεπτεμβρίου 2012.

Νικόλαος Παπασπύρου	Κωστής Σαγώνας	Ευστάθιος Ζάχος
Επίκ. Καθηγητής Ε.Μ.Π.	Αν. Καθηγητής Ε.Μ.Π.	Καθηγητής Ε.Μ.Π.

Αθήνα, Σεπτέμβριος 2012

Ελευθέριος Κρητικός

Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Περίληψη

Ο κλάδος της τυπικής επαλήθευσης ασχολείται με την απόδειξη της ορθότητας προγραμμάτων και αλγορίθμων. Καθώς αυξάνεται -εκθετικά- η πολυπλοκότητα των υπολογιστικών συστημάτων αλλά και ενσωματώνονται σε καίρια σημεία, εντείνεται η ανάγκη να υπάρχουν αδιαμφισβήτητες εγγυήσεις ότι ένα πρόγραμμα θα έχει την επιθυμητή συμπεριφορά.

Ένα από τα πιο ενδιαφέροντα προβλήματα στην περιοχή της τυπικής επαλήθευσης είναι το πρόβλημα framing. Σε αυτή τη διπλωματική εργασία ασχολούμαστε με μια νέα μέθοδο που επεκτείνει τις μέχρι σήμερα μεθοδολογίες αντιμετώπισης του προβλήματος framing, τους backpointers.

Οι backpointers επεκτείνουν την εκφραστικότητα των προδιαγραφών παρέχοντας τη δυνατότητα αναφοράς σε αντικείμενα μη προσβάσιμα από τη τρέχουσα στοίβα. Δημιουργήσαμε μια υλοποίηση των backpointers για τη Chalice, μια αντικειμενοστραφή και concurrent γλώσσα προδιαγραφών που χρησιμοποιεί implicit dynamic frames και fractional permissions. Τέλος, χρησιμοποιήσαμε την επέκταση αυτή για την απόδειξη της ορθότητας μιας ταυτόχρονης δομής δεδομένων, τις copy-on-write λίστες και του priority inheritance protocol. Σύμφωνα με την έρευνα μας είναι η πρώτη προσπάθεια απόδειξης ενός προγράμματος αυτής της κατηγορίας με χρήση αυτόματης επαλήθευσης.

Λέξεις κλειδιά

backpointers, Chalice, τυπική επαλήθευση, προδιαγραφές, copy-on-write λίστες, τυπικές μέθοδοι, πρόβλημα framing, implicit dynamic frames, fictional disjoint δομές δεδομένων, priority inheritance protocol

Abstract

Solid guarantees that a computer program will behave the desired way is only possible by formal verification. As the complexity of designs grows exponentially and computer systems are used in critical applications, it becomes increasingly important to prove the correctness of algorithms.

One of the most challenging problems in formal specification and verification is the framing problem. In this thesis we propose an new formalism that extends the current methodologies targeting the framing problem: Backpointers.

Backpointers increase the expressiveness by allowing specifications to include objects not accessible from the current stack. We implemented this approach in Chalice, an object-oriented and concurrent specification language that uses implicit dynamic frames and fractional permissions. Finally, by using this extension of Chalice we worked on proving the validity of a concurrent implementation of copy-on-write lists and the priority inheritance proto-col. To the best of our knowledge this is the first attempt to prove a program from this family of problems using an automated verifier.

Key words

backpointers, Chalice, verification, specification, copy-on-write lists, formal methods, framing problem, implicit dynamic frames, fictional disjoint datastructures, priority inheritance prolocol

Ευχαριστίες

Θα ήθελα να ευχαριστήσω τον Γιάννη Κασσιό για την ευκαιρία που μου έδωσε να δουλέψω πάνω σε ένα πάρα πολύ ενδιαφέρον θέμα στο τομέα των τυπικών μεθόδων καθώς για την πολύ ευχάριστη και καρποφόρα συνεργασία που είχαμε.

Επίσης τον καθηγητή και υπεύθυνο της διπλωματικής κ. Νίκο Παπασπύρου και τον καθηγητή κ. Στάθη Ζάχο για αυτά που μου έμαθαν όσο ήμουν στο πολυτεχνείο.

Τέλος, θέλω να ευχαριστήσω την οικογένειά μου για την στήριξή και την υπομονή τους όλα αυτά τα χρόνια καθώς και τους φίλους μου που συμπαραστάθηκαν.

7c8e21f6bab324679072568a1b39585f2792cbf68072466e5a7b471ddb56f0a4f93efdb09a2ed7e-97abb0d968a56003c661000a86f671fed78feeb4e645d8485

Ελευθέριος Κρητικός,

Αθήνα, 19η Σεπτεμβρίου 2012

Η εργασία αυτή είναι επίσης διαθέσιμη ως Τεχνική Αναφορά CSD-SW-TR-1-12, Εθνικό Μετσόβιο Πολυτεχνείο, Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών, Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών, Εργαστήριο Τεχνολογίας Λογισμικού, Σεπτέμβριος 2012.

URL: `http://www.softlab.ntua.gr/techrep/`
FTP: `ftp://ftp.softlab.ntua.gr/pub/techrep/`

Contents

List of Figures

Listings

Chapter 1

Introduction

Formal verification of computer programs is the art of proving the correctness of programs using formal methods of mathematics; it has been a very active field of research in Computer Science for some time now [Hoar69]. There have been a lot of advances using various approaches; one of them is *automated program verification*.

Automated program verification is based on a tool, the verifier; the user provides the source code as well as the desired specification and the verifier proves the validity of the program with little or no help from the user. The specifications are written in a language based on a mathematical framework and in some cases they are integrated into the programming language of the source code. The language along with the underlying framework and optionally a set of rules, which determine which programs are valid, are sometimes called a *verification methodology*.

One of the most challenging problems in formal specification and verification is the *framing problem*. In an ordinary imperative language, a formalism is used in the semantics of a statement to define which part of the heap is affected by this statement. When seen from an object-oriented perspective, the specification of a class and its methods cannot and should not speak about the whole heap, but rather depend only on locally accessible objects; in this object-oriented setting, preserving abstraction boundaries in the specifications is of utmost importance. Since *Hoare Logic*, many new methodologies have been proposed that attempt to solve the framing problem, some of the most notable ones being *Separation Logic* and *Dynamic Frames*.

Formal verification is *hard*. Formal verification for concurrent programs is even *harder* as verification methodologies should be able to prove complex properties, such as absence of race conditions and deadlocks. One of the most useful approaches in a concurrent setting is the methodology of *implicit dynamic frames* that use *fractional and counting permissions* to reason about race conditions in a very concise way.

Chalice is an object-oriented language and its concurrency model supports threads and shared objects using monitors. Chalice, as a specification language, has been proved successful in verifying the correctness of concurrent algorithms and datastructures using the aforementioned methodology of implicit dynamic frames. Chalice also supports *monitor invariants*, a methodology that is very important in verifying object-oriented programs.

While specification methodologies that focus on solving the framing problem succeed in describing a plethora of problems, there are cases where these techniques fail to give a good solution, for example in programs whose specifications and invariants do not depend on objects that are visible from the current frame while they still have a degree of locality. More specifically there are cases where the invariants of an object a depend on other objects directly pointing to a through some field.

Backpointer methodology tries to attack exactly this type of problems. In this thesis we base our work on implicit dynamic frames, a methodology that builds upon dynamic frames, liberating the user from writing frame annotations by replacing them with *access permissions*, in a style that reminds us of the separation logic frame rule. A *backpointer* $(A.b)^{-1}$ on an object b is the set of all objects a of type A that point to b through the field $A.b$. We extended

Chalice to support this new formalism and made some extensions and changes to the existing implementation.

Using this extension of Chalice we worked on proving the validity of a challenging algorithm from the area of *fictional disjointed datastructures*, a concurrent implementation of *copy-on-write lists*. Similar problems have been investigated in the past [Mehn12] and are interesting verification challenges. To the best of our knowledge this is the first attempt to prove a program from this family of problems using an automated verifier.

In the following chapters we first introduce the basic methodology (section: 2.1) and the tools (Z3 and Boogie, sections: 2.2 and 2.3) that Chalice depends on. Then, we will introduce Chalice in section 2.4 and the technical details of its implementation in section 2.5. Afterward, in chapter 3 the methodology of backpointers is introduced with the respective extensions of Chalice. In chapter 4 we show how our formalism can be used to verify complex concurrent programs, using one example from the family of fictional disjointness datastructures and one concurrent datastructure from system programming. We explain the concurrent copy-on-write lists implementation (section 4.1) and give a specification, proving it using Chalice with Backpointers. We also provide a verification for the priority inheritance protocol (section 4.2) based on backpointers. Finally, in chapter 5 we show the results of our work and give directions on how this work can be extended in the future.

Chapter 2

Background

2.1 The evolution of verification methodologies

2.1.1 The framing problem

One of the most important challenges of program specification and verification is the framing problem. To describe the framing problem we will study an assignment statement in a framework similar to Hoare logic. Such as assignment will have the following form:

$$\{\ldots\}\ x := 42\ \{x = 42\}$$

While the changes in variable `x` are fully specified, the aforementioned code is not a complete specification because of the unknown modifications that may occur to the other variables of the program. This problem is especially intensified in case of an infinite heap space. Moreover, when writing a specification for an object oriented programming language it is of utmost importance that the specifications do not cross the abstraction boundaries; the specification must only refer to fields or functions accessible to the client in order to preserve the modularity of the code.

2.1.2 Relative work on the framing problem

There are many approaches that address the framing problem:

- Separation logic
- Dynamic Frames
- Implicit dynamic frames
- Ownership and Universes type systems[1]
- Regional logic[2]

Separation logic [Reyn02] extends Hoare logic by using partial functions to model heaps and assert the disjointness of two heaps as the disjointness of the two function domains. The notation $h \models l \mapsto e$ implies that the partial function h (the heap) is defined in exactly one position, at l, and has the value e. Using this notion of disjointness one can *separate* the heap (hence the name of the methodology) into the part that is affected by a statement and a part that is not affected.

The separating conjunction operator ($*$) is used to denote propositions that are valid in disjointed parts of the heap. With this formalism the frame rule can be expressed as:

$$\frac{\{P\}\ C\ \{Q\}}{\{P * R\}\ C\ \{Q * R\}}\ \mathsf{modifies}(C) \cap \mathsf{freevariables}(R) = \varnothing$$

[1] [Clar98, Mull02, Barn06, Diet07]

[2] [Bane08]

This way, it is possible to reason locally about the statement C and then generalize to a bigger heap with R as long as the two are disjoint and the variables that C modifies are not mentioned in R.

On the other hand, the methodology of dynamic frames [Kass06] uses a different approach: specification[3] variables or specification functions[4] are used to encode the set of locations that a statement can affect, which is referred as the *dynamic frame*. This however requires that the frame of a statement is explicitly annotated in its specification. By following the above methodology, when a statement changes a variable, we can deduce that any variable that belongs in a frame that is disjointed with the frame of the changed variable is not affected by the statement.

```
1  class Cell {
2    private int x;
3
4    Cell()
5      writes ∅;
6      ensures getX() = 0;
7      ensures fresh(footprint());
8    {}
9
10   pure int getX()
11     reads footprint();
12   { return x; }
13
14   void setX(int value)
15     writes footprint();
16     ensures getX () = value;
17     ensures fresh(footprint() \ old(footprint()));
18   { x := value; }
19
20   pure set footprint()
21     reads footprint();
22   { return { &x }; }
23 }
24
25 // Client code
26 Cell c1 := new Cell();
27 c1.setX(5);
28
29 Cell c2 := new Cell();
30 c2.setX(10);
31
32 assert c1.getX() = 5;
```

Listing 2.1: A `Cell` with dynamic frames

An example written in the form of [Sman08] with an object-oriented Java-like setting shown in listing 2.1 examines the specification of a boxed integer, a cell. Highlighted are the parts that have to do with the dynamic frame bookkeeping. Expressions inside **`old()`** are referring to the state just before the start of the method. We declare frames, i.e. sets of locations, with the type **`set`**. The specification function **`footprint()`** defines all the locations related to this class. Every method or function needs to declare its frame, i.e. the set of locations it depends on, using **`reads`** or **`writes`** clauses. The constructor initializes **`footprint`** with the location of **`x`** (**`&x`**). The **`set`** method restates that the fresh locations must be the ones

[3] Specification variables, sometimes called ghost variables, are variables that are only used in the specification and ordinary code cannot depend on them. Specification functions are functions that can depend on specification variables and can be used in other specifications

[4] If not specified otherwise the term *function* is used to denote pure functions without any side-effects

newly found in `footprint`, even though `footprint` is not changed, encoding the *swinging pivot property.*

The dynamic frames theory improves separation logic by making the specification more modular and enable hiding the internal implementation from the client. Additionally, it supports sharing of locations between objects, something that was not possible with separation logic. One the other hand, explicitly encoding the frame for each method is cumbersome, both for the developer and the tool used for the verification, something that the implicit dynamic frames approach tries to solve.

2.1.3 Implicit dynamic frames

Implicit dynamic frames [Sman09] extend the logic used in specification with *permissions*. A permission to a location is needed in order, for the code and the specification, to refer to that particular location. From the specification, one can then infer an upper bound of the set of locations writable or readable by the corresponding method. Reminiscent of separation logic's frame rule, programmers write access assertions in a linear logic style multiplicative conjunction, with the *access permission* to a location `acc(x)` being the equivalent of $x \mapsto \_$ in Separation Logic [Park11]. At the same time this formalism uses explicitly written annotations in the specifications in order to supply the frame of a method and access permissions can be encoded in specification predicates or functions the same way it is done in dynamic frames.

```
1  class Cell {
2    int x;
3
4    Cell()
5       ensures acc(this.x) ∧ getX() = 0;
6    { this.x := 0; }
7
8    void setX(int v)
9       requires acc(this.x);
10      ensures acc(this.x) ∧ getX() = v;
11   { this.x := v; }
12
13   pure int getX()
14      requires acc(this.x);
15   { return this.x; }
16
17   void dispose()
18      requires acc(this.x)
19      // no ensures
20   { /* delete x */ }
21 }
22
23 // Client code
24 Cell c1 := new Cell();
25 c1.setX(5);
26
27 Cell c2 := new Cell();
28 c2.setX(10);
29
30 assert c1.getX() = 5;
31 c1.dispose()
32 assert c2.getX() = 10;  // fails!
```

Listing 2.2: A `Cell` with implicit dynamic frames

The previous cell example written with implicit dynamic frames formalism [Sman09], listing 2.2, uses the access permission on `x` wherever the specification or the implementation

needs access to this location. We know that when a permission to a particular location is lost then we have no way of knowing what happens to that location, and thus can assert nothing about it.

We can improve on this example by hiding the permission on **this.x** in an *abstract predicate* [Park05]. This way the internal field **x** is hidden from the client. The resulting code is shown in listing 2.3. In this example an extra method **swap()** is provided. Keep in mind that permissions and abstract predicates as expressed in [Sman09] are resources, meaning that one cannot assume that:

```
1   acc(this.x)  ⟹  acc(this.x) ∗ acc(this.x)
```

This predicate **valid** is used by the client of the class as a token with which the client can use the interface of **Cell**. All methods and functions of **Cell** take it as a requirement and ensures its validity as a postcondition.

```
1  class Cell {
2    int x;
3
4    Cell()
5       ensures valid() ∧ getX() = 0;
6    { this.x := 0; }
7
8    void setX(int v)
9       requires valid();
10      ensures valid() ∧ getX() = v;
11   { this.x := v; }
12
13   predicate bool valid()
14   { return acc(this.x); }
15
16   pure int getX()
17      requires valid();
18   { return this.x; }
19
20   void swap(Cell c)
21      requires valid() ∗ c ≠ null ∧ c.valid();
22      ensures valid() ∗ c.valid();
23      ensures getX() = old(c.getX());
24      ensures c.getX() = old(getX());
25   { int i := x; x := c.getX(); c.setX(i); }
26 }
```

Listing 2.3: A **Cell** with permissions hidden with an abstract predicate

2.1.4 Fractional and Counting permissions

When considering a multi-threaded environment where many threads can operate on the same data/objects, one can argue that many threads must have permission on the same location at the same time. Of course we want to exclude the possibility of two threads reading and writing on the same location, without a predefined ordering of operations, creating a *race condition*. In order to avoid this we can impose a simple rule: either only one thread can write (and read) on a location and no one else can access it *or* multiple threads can read from the same location but no one can write to it. Given this scheme one can think of two types of permissions, given an implicit dynamic frames setting, a *write permission* and a *read permission*. Write permissions must be unique while read permissions on the same location can exist in many copies, and the existence of a read permission must exclude the write

permission. One way to think of it is to model the access permission on a location as an quantity that can be split; this is the methodology of *fractional permissions*, proposed by [Boyl03] and used by [Zhao07]. The permission has initially the value 100% and is preserved (there is no way to make more of this). Access permissions are fractions of this quantity, for example, one can have 50% access on some field `x`. When a thread has the full permission, i.e. 100%, then it has write access on the location while anything less than 100% is a read permission. By having a permission (100% or less), it is possible to split this permission into multiple (read) permissions, and thus a client can spawn threads that read of the same variable. Furthermore, when the threads terminate and join in the client, they return their permissions and the client can sum them up, regaining the original permission. Notice that by using this scheme only one thread can have write permission at any given time. There is also a alternative way to split access permissions, called *counting permission* [Born05] in which we use indivisible infinitesimal permission fractions called *epsilon permission*. Epsilon permissions are essentially read permissions.Note that in this methodology, we keep a counter of how many epsilon permissions were subtracted from a fractional permission; therefore, it can always be restored. This technique is useful when it is required to know how many objects have access to a specific location.

2.1.5 Verification Condition Generation (VCG)

There are many proposed ways to check or verify specifications: some methodologies make use of a proof assistants while others use run-time assertions checked by the run-time system, e.g. [Meye97, Cheo02]. Model checking [Clar82, Quei82], abstract interpretation [Cous77] and symbolic execution [King76] are also common techniques in formal verification. Finally, some methodologies make use of type-systems to express the desired behavior. In the context of automated verification, *verification condition generation* is a methodology for proving the validity of a program given its specifications. With this methodology, logic formulas are generated based on the source code and the specifications called *verification conditions*, or VC; the validity of these verification conditions implies the correctness of the original code against the specifications. VCs are usually first-order logic formulas and can be given to an independent proving tool to be verified. Popular proving tools used with this methodology are SMT (Satisfiability Modulo Theories) solvers like Z3. For a comparative study of VCG and symbolic execution see [Kass12b].

VCG is so popular that there are tools and libraries, such as Why/Why3 [5] [Fill03, Bobo11] and Boogie, that assist in the generation of VCs. Boogie from [Lein08] is a intermediate language that assists in the VCG process, acting as a proxy for VC generation. When building a verification tool for some other language one can translate the source language into Boogie code and then Boogie is responsible for generating the appropriate VCs and prove them (using some SMT solver). In fact, this is exactly the technique used in the verification tool Chalice, that is the implementation target of this thesis.

2.2 SMT solvers and Z3

Satisfiability Modulo Theories (SMT) is a research domain of mathematics and computer science that has produced many valuable results. SMT solvers are an extension to SAT solvers that can handle mathematical theories with equality; these solvers usually support the decision problem for some theory and also provide a model for that theory if it is proved satisfiable.

A common input and output languages for SMT solvers is the SMT-LIB version 2 language

[5] `http://why.lri.fr/`, `http://why3.lri.fr/`

([Barr10a] and [Barr10b] and a good tutorial: [Cok12]) developed by the SMT-LIB initiative[6]. In this language, that bears many similarities in its syntax with LISP, theories can be expressed and the validity of those can be verified by an SMT solver.

```
1  (set-option :print-success false)
2  (set-option :produce-models true)
3  (set-option :interactive-mode true)
4  (set-logic QF_NIA)  ; non-linear integer arithmetic
5  (declare-fun x () Int)
6  (declare-fun y () Int)
7  (declare-fun z () Int)
8  (assert (> x 0))
9  (assert (> y 0))
10 (assert (> z 0))
11 (assert (= (+ (* x x) (* y y)) (* z z)))
12 (check-sat)
13 (get-value (x y z))
14 (exit)
```

Listing 2.4: Example SMT theory

In listing 2.4 we assert the existence of three integer variables x, y, z and some constrains on them; these constrains are not-linear so in line 4 we use the integer theory without the linearity restriction. We ask the prover about the satisfiability of this theory and a corresponding model; in listing 2.5 we see the output of an SMT solver that proved the theory and founding a model for it.

```
1 sat
2 ((x 12)
3  (y 9)
4  (z 15))
```

Listing 2.5: SMT prover result

There are many industrial strength tools that can prove the satisfiability of logic formulas, such as Alt-Ergo[7], CVC3[8], STP[9], Yices[10] and Z3. Our work is based on Z3 from Microsoft Research[11] [De M08]; Z3 can handle theories with linear arithmetic, nonlinear arithmetic, bitvectors, arrays and datatypes, as well as empty theories.

Theories in SMT can include axioms with universal quantification; as a result the prover cannot be complete (although modern provers manage to cope well with quantification). Z3 uses several approaches to handle quantifiers [Z3Qu]; the most prolific approach is using pattern-based quantifier instantiation. The universal quantifier is annotated with a pattern: an expression that mentions all the quantified variables. Then, whenever in the search space ground-terms matching the pattern appear, the body of the quantifier is instantiated with the values found in the pattern. Any expression can be used as a pattern; it is not required to appear in the body of the quantifier. This method might, depending on the pattern, make Z3 incomplete and also put the decision procedure into a matching loop if newly created ground terms transitively match the pattern that created them. Z3 uses many heuristics to break matching loops which typically severely hinder the performance of the prover.

[6] http://www.smt-lib.org/

[7] http://ergo.lri.fr/

[8] http://www.cs.nyu.edu/acsys/cvc3/

[9] http://sites.google.com/site/stpfastprover/

[10] http://yices.csl.sri.com/

[11] http://z3.codeplex.com/

2.3 Boogie

Boogie from Microsoft Research[12] is an intermediate verification language which is used as a library to help with the verification condition generation procedure. It is used by a variety of tools including Spec# [Barn05], VCC [Cohe09], Dafny [Lein10a] and of course Chalice.

The Boogie language is a simple procedural language focused on specifications with a bounded space of global variables. Boogie features seven kinds of declarations: the mathematical constructs are types, constants, functions, and axioms while the imperative constructs are global variables, procedure declarations, and procedure implementations.

Type declarations introduce type constructors. For example,

```
1 type String;
```

declares a type (more precisely, a nullary type constructor) intended to represent strings. Types can be polymorphic by adding type variables to the type constructor, e.g.

```
1 type Field a;
```

is the type of fields of type `a`. Instantiations of `Field` can include `Field int` for the type of all integer fields and `Field String` for all string fields. Boogie supports some basic build-in types including `int` and `bool`. Boogie also supports possibly polymorphic *maps* also known as update-able maps, non-rigid functions or heterogeneous arrays. Syntactically, the domain types are listed within square brackets, followed by the range type. For example,

```
1 type Permissions = [Reference]bool;
```

makes `Permissions` a type alias of a map from `Reference`s to booleans that could encode the access permissions. Maps can have tuples[13] as their domain and can be polymorphic. For example

```
1 type Heap = <a>[Reference,Field a]a;
```

is a map from a `Reference` and a `Field` of type `a` to `a` and can model the heap. Symbolic constants are introduced by *constant declarations*, like

```
1 const emptyString : String;
```

which denotes that `emptyString` is a fixed yet unspecified value of type `String`. *Function declarations* introduce mathematical functions. For example,

```
1 function length(String) returns (int);
```

declares a function intended to return the length of `String`s. A function may have more than one return parameters, all written inside parentheses. Properties of constants and functions are postulated by *axiom declarations*. For example,

```
1 axiom length(emptyString) == 0;
```

[12] `http://research.microsoft.com/en-us/projects/boogie/`, `http://boogie.codeplex.com/`

[13] Cartesian product

says that `length` returns 0 for the `emptyString`.

The program state is created with mutable variables. Global *variable declarations* introduce the state on which all procedure operate. For example,

```
1 var heap: Heap;
```

introduces the variable **heap** for holding the current state of the program. A *procedure declaration* gives a name to a set of execution traces, which are specified by pre- and post-conditions. For example,

```
1 procedure NewFavorite(n: Wicket);
2     modifies favorite;
3     ensures favorite == n;
```

Finally, an *implementation declaration* defines a set of execution traces by giving a body of code. The implementation is correct if and only if its set of traces is a subset of the traces specified by the corresponding procedure. For example,

```
1 implementation NewFavorite(n: Wicket)
2 {
3     favorite := n;
4 }
```

gives a correct implementation of the procedure `NewFavorite`. Procedures are specified via pre- and post-conditions with a **modifies** clause used to encode which variables are affected by each procedure. Inside procedures ordinary control-flow structures are available.

Boogie expressions include many of the usual constructs like constants, variables, equality and arithmetic relations, boolean connectives, simple arithmetic operators and logical quantifiers. Figure 2.1 is a useful but not complete list of Boogie expressions.

expr ArithmOp expr	*ArithmOp* is an arithmetic operator one of `+`, `-`, `*`, `/` or `%`
expr CompOp expr	*CompOp* is a relational operator one of `==`, `!=`, `<`, `<=`, `>` or `>=`
`!`*expr*	Negation
expr `&&` *expr*	Conjunction
expr `||` *expr*	Disjunction
expr `==>` *expr*	Implication
expr `<==>` *expr*	Equivalence
id`(`*expr*`,` *expr*`, . . . )`	Function call
expr`[`*expr*`]`	Map selection
expr`[`*expr* `:=` *expr*`]`	Map update
`(forall` *var*`:` *type* `::` *expr*`)`	Universal quantification
`old(`*expr*`)`	Expression referring to the pre-state

Figure 2.1: Boogie expressions

Besides the usual expressions, Boogie supports selecting and updating syntax for maps. Another worth mentioning feature is the ability to reference *"old"* expressions. Expressions that are inside an **old()** expression refer to the pre-state, the state of the world just before the method began. Universal quantification over, possibly infinite, domains is allowed in Boogie expressions. Boogie also supports existential quantification, a feature not explored here.

When quantification is used, one can also supply triggers for the quantifier. Boogie triggers are directly translated to patterns at SMT level. A trigger can be supplied just before the body of the quantification, inside curly braces, as shown below.

```
    (forall a: A :: {trig1} {trig2} ...body...)
```

Each trigger is a list of comma separated expression patterns that mention the bounded variables of the quantification and maybe some of the visible variables or constants. For example:

```
function pow(a: int, n: int): int;
axiom (forall a: int, n: int :: { pow(a, n) }
    pow(a, n) == if n == 0 then 1 else a * pow(a, n-1)
);
axiom (forall a: int, b: int :: { (a * b) } (b * a) / b == a);
```

Writing triggers that are too specific limits completeness but preserves soundness. There are some restrictions on the use of triggers:

1. The list of terms in a trigger must include all bound variables of the quantifier.
2. A term listed in a trigger must not be a bound variable itself.
3. A trigger must not include logical operators or quantifiers.

Procedure implementations consist of ordinary statements found in most imperative languages but also support some constructs specific to program verification. Figure 2.2 is the, almost complete, syntax of Boogie statements. Note that procedures and functions have multiple return parameters and the assignment syntax can use multiple l-values at the same time.

The Boogie statements related to specification and verification are:

1. loop invariant syntax.
2. assert statement
3. assume statement
4. havoc statement

Loop invariants are conditions that must be true before and after one run of the loop body. Assert is used to force the prover to prove some property, in the form of a predicate, at a particular point in the code. For example:

Body ::= `{` *LocalVarDecl** *StmtList* `}`
LocalVarDecl ::= `var` *Attribute** *IdsTypeWhere*$^{+,}$ `;`
StmtList ::= *LStmt** *LEmpty*$^?$
LStmt ::= *Stmt* | *Id* `:` *LStmt*
LEmpty ::= *Id* `:` *LEmpty*$^?$
Stmt ::= `assert` *Attribute** *Expr* `;`
| `assume` *Attribute** *Expr* `;`
| `havoc` *Id*$^{+,}$ `;`
| *Lhs*$^{+,}$ `:=` *Expr*$^{+,}$ `;`
| `call` *CallLhs*$^?$ *Id* `(` *Expr**, `)` `;`
| *IfStmt*
| `while` `(` *Expr* `)` *LoopInv** *BlockStmt*
| `break` *Id*$^?$ `;`
| `return` `;`
| `goto` *Id*$^{+,}$ `;`
Lhs ::= *Id* *MapSelect**
MapSelect ::= `[` *Expr*$^{+,}$ `]`
CallLhs ::= *Id*$^{+,}$ `:=`
BlockStmt ::= `{` *StmtList* `}`
IfStmt ::= `if` `(` *Expr* `)` *BlockStmt* *Else*$^?$
Else ::= `else` *BlockStmt* | `else` *IfStmt*
LoopInv ::= `invariant` *Attribute** *Expr* `;`

Terminal symbols in the grammar are written in **monospace** font while non-terminal are *oblique*. The usual notation (*expr**, *expr*$^+$, *expr*$^?$) for repetition and optional inclusion in EBNF grammars is used while *expr**, or *expr*$^{+,}$ is used to denote repetition separated with a comma (`,`).

Figure 2.2: Boogie's statement grammar

```
1   var a: int; var b: int; var c: int; var n: int;
2   a, b, c, n := 1, 1, 1, 3;
3   while (true)
4       invariant a >= 1;
5   {
6       a := a + 1;
7       while (true)
8           invariant b >= 1;
9       {
10          b := b + 1;
11          while (true)
12              invariant c >= 1;
13          {
14              c := c + 1;
15              while (true)
16                  invariant n > 2;
17              {
18                  assert pow(a, n) + pow(b, n) != pow(c, n);
19                  n := n + 1;
20                  assert pow(a, n)/a + pow(b, n)/b != pow(c, n)/c;
21              }
22          }
23      }
24  }
```

In line 18 Boogie will try to prove the condition and, in case of failure, a warning message will be produced. For the rest of the program the proposition of the assert statement is assumed valid. Therefore, the next assert statement in line 20 will be easily proved.

Assume statements have the same effect as assert statements but the validity of the proposition is not proved but instead is just assumed for the rest of the program. Turning the first assert into an assume statement will not produce any errors.

```
17                  ...
18                  assume pow(a, n) + pow(b, n) != pow(c, n);
19                  n := n + 1;
20                  assert pow(a, n)/a + pow(b, n)/b != pow(c, n)/c;
21                  ...
```

Using an assume statement might introduce a contradiction without a warning so it should be used with caution. However, there are many cases where parts of the truth can only be axiomatically assumed so assume statements are required.

Another feature that is not found in many ordinary programing languages is the havoc statement. Havocing a variable has the effect of assigning to the variable an unspecified "random" value (the type of the variable is preserved). It is not the same as assigning null to the variable which is a specific value. A variable that has been havoced is a variable that we know nothing about it so any assertion regarding that variable will fail. Havoc is usually used in conjunction with a subsequent assume that limits the effects of havocing. For example:

```
1   havoc x;
2   assume x*x - 5*x +6 == 0;
```

will have the effect of setting `x` to one solution of the equation.

For a full reference of the Boogie language refer to [Lein08] and [Barn06].

2.4 Chalice

Chalice[14] is an object oriented, specification language for concurrent programs and a verification tool for that language. It supports the commonly encountered features of object oriented languages such as encapsulation and modularity that play a significant role in formal specification. Chalice programs are translated into Boogie intermediate language and the Boogie verification engine is used to prove the validity of program specifications.

Most of the imperative structures of Chalice are similar to the ones found in Boogie. From now on we will focus on the concurrent features of Chalice, exploring them through a series of examples.

Let's assume that we want to model a cell; a simple datastructure whose only purpose is to "box" a simple value. Chalice doesn't support polymorphic types or classes (known as generics in Java), so we will assume that this a integer cell.

```
1  class Cell
2  {
3      var x: int;
4  }
```

Let's assume that we want to implement a method in the class `Cell` that doubles the enclosed value. As mentioned before, Chalice uses implicit dynamic frames to solve the framing problem, so we need to have access permission in order to access the `x` field.

```
1  class Cell
2  {
3      var x: int;
4
5      method double()
6          requires acc(x)
7          ensures acc(x) && x == 2*old(x)
8      {
9          x := x*2;
10     }
11 }
```

In the contract of the method `double` we add the permission `acc(x)` both in the precondition and in the postcondition. The permission `acc(x)` (write permission or full permission) is needed because we modify the variable `x`. This way we do not lose any permissions, and the client of the method can subsequently access the field after using the call. In addition, we add a post-condition that describes the effect of this method i.e. doubling of the value `x`.

A simple client for this method would be:

```
1  class Client
2  {
3      method foo()
4      {
5          var c: Cell;
6          c := new Cell;
7          c.x := 21;
8          call c.double();
9          assert c.x == 42;
10     }
11 }
```

[14] `http://www.pm.inf.ethz.ch/research/chalice/`, `http://research.microsoft.com/en-us/projects/chalice/`

Given the previous definition of `Cell` the assert statement will succeed. In Chalice every call to a method has the effect of spawning a thread to run this method. So, in effect the previous call to `c.double()` is syntactic sugar for:

```
        ...
        fork tk := c.double();
        join tk;
        ...
```

The fork statement creates a thread that will execute `c.double()` and return a token. This token can then be used by a join statement that will wait for the previous thread to complete the execution.

When a thread executes a method with a fork statement, it consumes all the access permissions from the precondition of the method. That way, it is impossible for a second thread to execute `c.double()` concurrently.

```
    method bar()
    {
        var c: Cell;
        c := new Cell;
        c.x := 21;
        fork tk1 := c.double();
        fork tk2 := c.double();  // fails!
        join tk1;
        join tk2;
        assert c.x == 42;
    }
```

The second fork in this client would fail because at that point the calling thread does not have the permission on `x`. One way to solve this is to introduce a read permission; instead of `double()` we could have:

```
class Cell
{
    var x: int;

    method returnDouble() returns (res: int)
        requires rd(x)
        ensures rd(x) && res == 2*old(x)
    {
        res := x*2;
    }
}
```

`returnDouble()`, a method that does not change the cell value but rather return it doubled. We no longer have the full permission `acc(x)`, just a read permission on `x` denoted by `rd(x)`. However, consuming a read permission does not imply that no read permissions are left; a read permission can be split into arbitrary many copies. Using this implementation of `Cell` the client becomes:

```
class Client
{
    method baz()
    {
        var c: Cell;
        c := new Cell;
        c.x := 21;
```

```
8           fork tk1 := c.returnDouble();
9           fork tk2 := c.returnDouble();  // no problem
10          var c1: int; var c2: int;
11          join c1 := tk1;
12          join c2 := tk2;
13          assert c1 == 42 && c2 == 42;
14      }
15  }
```

The initial full permission, created when the Cell object was constructed, is split by the first fork of `c.returnDouble()` into two read permissions, one consumed by the fork and one left on the calling thread. This way, the calling thread has the necessary permission to fork one more time the same method before joining them. Tokens `tk1` and `tk2` are used in fork and join statements in order to retrieve the value returned by the methods. The assertion succeeds in determining that both calls return the double of the initial value.

Access permission can be passed from caller to callee through a fork statement and vice versa through a join statement. Access permissions can be lost but can only be created by creating a new object. In particular a write permission can be split in arbitrary many read permissions and read permissions can be subsequently split into more **`rd`** read permissions. It is also worth noting that we cannot combine some read permissions to make a write one. Using implicit dynamic frames and fractional access permissions ensures that no two distinct threads can access the same place (field of an object) and cause a race-condition. Only a single thread can have a write permission in a field while many threads can have read permissions at the same time and read from the same location. For a more detailed presentation of read permissions see [Heul11].

It is possible to solve the problem of concurrent access to the same resource using thread locks on resources hence introducing the concept of *monitors* [Hoar74]. A monitor is responsible for sequencing accesses to objects held by the monitor and pausing the thread until access to the objects is safe.

In Chalice terms, an object is initially *unshared*, i.e. it cannot be held by any monitor and is thread-local to the thread that created it. *Sharing* an object makes it available for locking and assigns a monitor to that object. The process of locking an object by a thread is called *acquiring* that object and the reverse is *releasing*. When an object `o` is acquired by an thread we say that the object is *held* by the thread or that the thread has *locked* the object, and the predicate **`holds(o)`** is true inside that thread. To make an object `o` available for locking we use the syntax **`share`** `o`. The **`acquire`** statement is used to acquire an object from the monitor while the **`release`** is used to release the object. An auxiliary statement **`share acquire`** is supplied that atomically shares an object and acquires it in the current thread. This statement was added during this thesis and is not part of the official release of Chalice. The object's life cycle is shown in figure 2.3.

In terms of access permissions, a monitor can hold access permissions just as a thread can. Taking the lock on an object from the monitor is translated in consuming the access permissions held by that monitor on the specific object. The access permissions held by a monitor when the object is shared and not held by any thread are those specified by the *monitor invariant* [Lein09b, sec. 4.0]. A monitor invariant is a set of predicates, along with access permissions, that must be held when the object is shared and not locked. That means that a thread can acquire an object, access it, change its state, but when it releases that object back to the monitor, the state of the object must comply with the specification encoded in the monitor invariant. Finally, before releasing, a thread must give back any permissions specified by the monitor invariant.

Revisiting the cell example, we have modeled the `Cell` class so that the cell is shared.

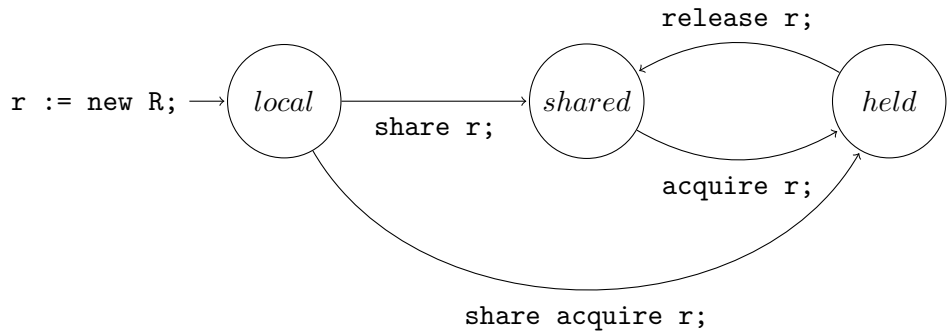


Figure 2.3: A object's life cycle

```
1  class Cell
2  {
3      var x: int;
4
5      invariant acc(x) && x % 2 == 0
6
7      method safeDouble()
8      {
9          acquire this;
10         x := x*2;
11         release this;
12     }
13 }
```

The monitor invariant for the cell has full access to field x, and thus each thread that locks a cell can change its value. We can also impose some other checks on the value of the cell, in this example that the value must be an even number. The "thread-safe" version of doubling, `safeDouble()`, first acquires the cell and then, when it is given the permission on x, can change the value before releasing. A client for this method could be:

```
1  class Client
2  {
3      method run(c: Cell)
4          requires c != null
5      {
6          while(true)
7          {
8              call c.safeDouble();
9          }
10         assert 42 == 17
11     }
12 }
```

Multiple clients can run in parallel without any interference between them, since the critical section of the value update is inside an acquire-release block.

```
1  class Main
2  {
3      method main()
4      {
5          var c: Cell;
6          var client1: Client;
7          var client2: Client;
8          c := new Cell;
```

```
9           c.x := 2;
10          client1 := new Client;
11          client2 := new Client;
12          share c;
13          fork tk1 := client1.run(c);
14          fork tk2 := client2.run(c);
15          join tk1;
16          join tk2;
17      }
18  }
```

Object invariants can been lengthy. It is common practice to break up the invariant into logical blocks concerning particular parts of the implementation. This tactic is also followed when some part of the invariant is "broken" i.e. does not hold at a particular point but the rest of the invariant holds true. This is a common case when we have acquired an object, changed its state, but have not finished processing it and some portion of its invariant is not valid. With a extension of Chalice, proposed and implemented in this thesis, we have a convenient way of expressing situations like this. Parts of the invariant can be named using an identifier and a colon (`:`) just after the keyword **`invariant`**:

```
1   class Cell
2   {
3       var x: int;
4       invariant Perm: acc(x)
5       invariant Even: x % 2 == 0;
6       ...
7   }
```

Then in the specifications we can mention the invariants with the syntax *object*`.`**`invariant(`** *invariantNames*`)`; for example something like:

```
41      ...
42      assert o.invariant(Perm);
43      ...
```

means that in this particular point in the code we assert that we have write permission for `o.x`. Replacing the name of the invariant with `*` implies that all the invariants hold at that point. Also the syntax **`broken`** *invariantNames* can be used to denote that all the invariants, except the ones named in it, hold. A common pattern emerges when a method is called to fix the invariant of an object already acquired. We can use the same syntax with the holds predicate:

```
41      ...
42      assert holds(this, Perm);
43      assert holds(this, *);
44      assert holds(this, broken Even);
45      ...
```

implying that the object **`this`** is acquired by the current thread and that the invariants mentioned hold: only the `Perm` invariant, *all* the invariants and all *but* the `Even` invariants respectively. For example the specification of a method of class `Cell` would be:

```
1   class Cell
2   {
3       ...
4       method FixInv()
```

```
        requires holds(this, broken Even);
        requires (x + 1) % 2 = 0;
        ensures holds(this, *)
    {
        x := x + 1;
    }
}
```

This syntax is inspired by *considerate reasoning* from [Summ10].

With this scheme we cannot know what is happening inside a cell and cannot assert anything about the state of its value since we have no permission on the field `x`. All the permissions are held by the monitor or by a thread that has locked the cell. The only thing we can be sure about is that when we acquire a cell the value must be an even number, i.e. only properties encoded in the monitor invariant.

Let's study a more flexible version of the cell, one that enables read access from different threads concurrently if the cell is shared and grants write access to the field `x` only if the cell is "owned" by only one thread. We need two distinct classes for this scheme. One class is the actual internal representation of the cell, called `CellImpl` and an interface, sentinel class, called `Cell`. An actual cell can be accessed through multiple sentinel objects, and we need to keep track of how many sentinels refer to that particular cell. This is done by introducing reference counting and the corresponding integer field `refCount`. As long as `refCount` $= 1$ we know that the cell is not shared and we can change the value of `x`.

To model access permissions we need to consider the following limitations:

1. Each sentinel object needs at least read access to the implementation cell in order to read its value.

2. If the cell is not shared we should be able the change the value.

By using read permissions (`rd(x)`) one cannot reconstruct a write permission and, in effect, the write access is lost forever, even if only one has read access. We need a way to know how many permissions are given away, or lost from the full permission, a methodology called *counting permissions* [Born05]. We coin the notion of an infinitesimal read permission that cannot be split called *epsilon permission*, symbolically ε. We also assert the existence of a read permission that is created if we consume a defined finite number (k) of epsilon permissions, symbolically written as: $100\%-k\varepsilon$. An $100\%-k\varepsilon$ permission can be further split into arbitrary more epsilon permissions, and can be joined with exactly k epsilons to form a full permission.

In our example we can assign one epsilon permission to each of the sentinel nodes and store the in the monitor invariant remaining permission, in order to ensure that it will not be lost. The remaining permissions are exactly $100\%-$`refCount`ε. That way each sentinel has access to the internal field `x` and if it is the only sentinel pointing to that cell (i.e. `refCount = 1`) it can reconstruct the full permission and write onto the cell, by acquiring the internal cell. When a sentinel finishes processing a cell it should return the epsilon permission on that cell, by properly updating the `refCount` field.

```
class CellImpl
{
    var x: int;
    var refCount: int;
    invariant acc(refCount) && acc(x, 100 - rd(refCount))
}

class Cell
{
    var c: CellImpl;

```

```
12      method set(x: int)
13          requires acc(c) && c != null && acc(c.x, rd(1))
14          ensures acc(c) && c != null && acc(c.x, rd(1))
15          ensures c.x == x
16      {
17          acquire c;
18          if (c.refCount == 1)
19          {
20              c.x := x;
21              release c;
22          } else {
23              ???
24          }
25      }
26  }
```

The Chalice syntax for one epsilon permission on `x` is `acc(x, rd(1))` while for the permission $100\% - k\varepsilon$ on `x` it is `acc(x, 100-rd(k))`. Note that in line 20 having acquired the internal cell `c` and knowing that `c.refCount` $= 1$ we can conclude that we have full access on `c.x` and so we can change it's value. With this syntax it is also possible to specify exact fractions [Boyl03] of the full permission and even mix them with counting permissions. For example `acc(x, 25)` means 25% of the full permission and `acc(x, 50+rd(42))` means 50% with 42 extra epsilon permissions. For example this expression:

```
1       acc(x, 25) && acc(x, 50+rd(42)) && acc(x, 25-rd(42))
```

is equivalent to a full permission.

The aforementioned methodology can be further illustrated by the following example: assume that we want to build a complete interface for the sentinel objects `Cell`, providing methods for creating new cells, copying existing cells, getting the value of a cell and setting a value to a cell. Internally, a copy will be represented by a new sentinel node that will point to the copied `CellImpl` after increasing the `refCount` value by one. When the cell is assigned a value, if the cell is not shared, the code above will be used. If the cell is shared (`refCount` > 1) then we will create a new internal cell, set its value and decrease the old cell's `refCount`. A possible code is shown in listing 2.6. The syntax of returning values from a method is illustrated in line 30.

```
1   class Cell
2   {
3       var c: CellImpl;
4
5       method init(x: int)
6           requires acc(c) && c == null
7           ensures acc(c) && c != null && acc(c.x, rd(1))
8           ensures get() == x
9       {
10          c := new CellImpl;
11          c.x := x;
12          c.refCount := 1;
13          share c;
14      }
15
16      method initCopy(other: Cell)
17          requires acc(c) && c == null
18          requires other != null && acc(other.c) && other.c != null &&
19                   acc(other.c.x, rd(1))
20          ensures acc(c) && c != null && acc(c.x, rd(1))
21          ensures acc(other.c) && other.c != null && acc(other.c.x, rd(1))
```

```
22          ensures other.c.x == c.x
23      {
24          acquire other.c;
25          c := other.c;
26          other.c.refCount := other.c.refCount + 1;
27          release other.c;
28      }
29
30      method get() returns(res: int)
31          requires acc(c) && c != null && acc(c.x, rd(1))
32          ensures acc(c) && c != null && acc(c.x, rd(1))
33          ensures res == c.x
34      {
35          res := c.x
36      }
37
38      method set(x: int)
39          requires acc(c) && c != null && acc(c.x, rd(1))
40          ensures acc(c) && c != null && acc(c.x, rd(1))
41          ensures c.x == x
42      {
43          acquire c;
44          if (c.refCount == 1)
45          {
46              c.x := x;
47              release c;
48          } else {
49              c.refCount := c.refCount - 1;
50              release c;
51              c := null;
52              call init(x);
53          }
54      }
55  }
```

Listing 2.6: Fictional disjoint `Cell`

Even though this is a accepted solution, it has one critical flaw. The client of the interface `Cell` needs to know about the internals of its implementation. In order to preserve information hiding and to abstract the implementation we introduce *abstract predicates* and *functions*.

Functions are pure expressions that are given a name and some necessary permissions. Functions are side-effect free and simply express the state of the object, as viewed from the domain specific point of the class. For example in our cell example the value of the cell could be modeled into a function and so replace the `get()` method:

```
1       function get(): int
2           requires acc(c) && c != null && acc(c.x, rd(1))
3       {
4           c.x
5       }
```

Because Chalice functions are *pure*, they can be used in expressions and most importantly in specification pre- and post-conditions. That way the specifications of the methods of a class can be expressed in terms of it's functions; i.e. the effect of the method that can be observed through it's public interface. This way we hide the internal representation of the class and the client does not need to know anything about how the class is implemented. For example the specifications of method `set()` becomes:

```
    method set(x: int)
        requires acc(c) && c != null && acc(c.x, rd(1))
        ensures acc(c) && c != null && acc(c.x, rd(1))
        ensures get() == x
```

Another example of the usage of functions in specifications could be the interface of a list:

```
class List
{
    ... // Implementation here

    function at(i: int): int        // value at index
        ensures i < length()
    { ... }

    function length(): int          // length of the list
    { ... }

    method deleteAll()              // empties the list
        ensures length() == 0
    { ... }

    method append(x: int)           // append an element to the end of the list
        ensures length() == old(length()) + 1
        ensures (forall i: int :: i < old(length()) ==> at(i) == old(at(i)))
        ensures at(length() - 1) == x
    { ... }

    method update(i: int, x: int)  // update elemet at index i with value x
        ensures length() == old(length())
        ensures (forall j: int :: j < length() ==>
                        ite(i == j, at(j) == x, at(j) == old(at(j))))
    { ... }
}
```

The expression `ite(`a`, `b`, `c`)` has the semantics of if-then-else and is short-hand for `(`a `==>` b `&& !`a `==>` c`)`. The code does not include the actual method implementation and the required permissions for simplicity.

As we can see from the code, the methods of class `Cell` need some, very specific, permissions and conditions in order to function properly. In particular, the predicate needed by most methods is:

```
acc(cell.c) && cell.c != null && acc(cell.c.x, rd(1))
```

on some `Cell` *cell*. We abstract this as an abstract predicate [Park05] called `valid`. `valid` is used as a token (not to be confused with the token returned by a join statement) by the client in order to use a cell. Acquisition of a `valid` predicate is possible only when the client has the required permissions to use the cell. All methods require a `valid` predicate and ensure its return back to the client after the call to the interface method. The only method not requiring this predicate, but only producing it, is the constructor methods `init()` and `initCopy()`. For example the specification of `set()` introducing `valid` would be:

```
    method set(x: int)
        requires valid
        ensures valid
        ensures get() == x
```

Abstract predicates are also useful when dealing with permissions or conditions that are arbitrary long and can be expressed using a recursive predicate, e.g. a type of the "token" required by a linked list class. In order to access all the items we would need read permissions for all the (finite, but arbitrary many) nodes of the list. A suitable predicate would be:

```
class Node {
    var next: Node;
    var value: int;

    predicate valid {
        acc(next) && acc(value) && (next != null ==> next.valid)
    }
}
```

Due to the recursive form of the predicate and its infinite expandability, the prover cannot decide how many times to expand it. This is a well known problem with recursive functions when dealing with automated provers. The solution given by Chalice is to manually direct the prover when to expand or retract an abstract predicate, using the *unfold* and *fold* directives respectively. When a predicate is unfolded the predicate is replaced with its body. The reverse happens when folding a predicate; if the necessary conditions and permissions of the body of the predicate are valid, they are replaced with the predicate itself. Note that abstract predicates, like permissions, use linear logic, e.g. a predicate can be consumed by a fork statement or can be returned by a join statement. A more important consequence is that a folded predicate cannot be spit it into two separate ones, consumed by different by different calls. In figure 2.7 we can see the whole implementation of `Cell` with its interface abstracted from its implementation. The highlighted code ensures that an object that is about to be acquired is not already acquired by the current thread.

```
class Cell
{
    var c: CellImpl;

    predicate valid { acc(c) && c != null && acc(c.x, rd(1)) }

    method init(x: int)
        requires acc(c) && c == null
        ensures valid
        ensures get() == x
    {
        c := new CellImpl;
        c.x := x;
        c.refCount := 1;
        share c;
        fold valid;
    }

    method initCopy(other: Cell)
        requires acc(c) && other != null && other.valid
        requires unfolding other.valid in !holds(other.c)
        ensures valid && other.valid
        ensures other.get() == get()
        ensures unfolding other.valid in !holds(other.c)
    {
        unfold other.valid;
        acquire other.c;
        c := other.c;
        other.c.refCount := other.c.refCount + 1;
        release other.c;
        fold other.valid
```

```
32          fold valid
33      }
34
35      function get(): int
36          requires valid
37      {
38          unfolding valid in c.x
39      }
40
41      method set(x: int)
42          requires valid
43          requires unfolding valid in !holds(c)
44          ensures valid
45          ensures get() == x
46          ensures unfolding valid in !holds(c)
47      {
48          unfold valid;
49          acquire c;
50          if (c.refCount == 1)
51          {
52              c.x := x;
53              release c;
54              fold valid;
55          } else {
56              c.refCount := c.refCount - 1;
57              release c;
58              c := null;
59              call init(x);
60          }
61      }
62  }
```

Listing 2.7: Fictional disjoint `Cell` with interface fixed

Besides race condition prevention another feature Chalice supports is *deadlock prevention* [Lein10b]. A deadlock occurs when some threads, even only one, try to transitively acquire the same resource. In order to avoid this situation all shared objects are ordered with a partial ordering relation. When trying to acquire an object, one must prove that the specific object is "higher" in the chain of objects for all other objects already acquired by this thread.

The ordering relation ($\sqsubset$), textually denoted by the `<<` operator, is defined to be dense on a set of values, **`Ord`**. Each object is equipped with a implicit special field called μ or in code `.mu`, of type **`Ord`**. The max of all the `mu` field of all acquired objects is called **`waitlevel`** and is a ghost variable of type **`Ord`** maintained by the implementation of Chalice, i.e. for **`waitlevel`** it is always true that:

$$\texttt{waitlevel} = \max_{o \in \texttt{holds}(o)} o.\mu$$

When sharing an object we must specify the bounds of its `mu` field. We do that with the share syntax:

```
1       share o above α;
```

meaning that:

$$\alpha \sqsubset \texttt{o}.\mu$$

or by:

```
1       share o below β;
```

meaning that:

$$\mathtt{o}.\mu \sqsubset \beta$$

finally

```
share o between α and β;
```

would mean that:

$$\alpha \sqsubset \mathtt{o}.\mu \sqsubset \beta$$

where α and β are valid expressions of type **Ord**. If no bounds are given it is implied that the object is shared above **waitlevel**.

For the mu field of the objects, even though it is implicitly defined in every class, we still need access permissions to access and modify it. Only the share statements modify the mu field but whenever we want to acquire an object we need read access to the mu field in order to prove that it is above **waitlevel**. For example the specifications of init(), initCopy() and set() could be re-written using deadlock avoidance:

```
    method init(x: int)
        requires acc(c) && c == null
        requires acc(c.mu) && c.mu == lockbottom
        ensures valid
        ensures get() == x
        ensures rd(c.mu) && waitlevel << mu
    {
        ...
        share c;  // above waitlevel
        ...
    }

    method initCopy(other: Cell)
        requires acc(c) && other != null && other.valid
        requires unfolding other.valid in
                         rd(other.c.mu) &&
                         waitlevel << other.c.mu
        ensures valid && other.valid
        ensures other.get() == get()
        ensures unfolding other.valid in rd(other.c.mu)
    {
        ...
        acquire other.c;  // succeededs due to the precondition
        ...
    }

    method set(x: int)
        requires valid
        requires unfolding valid in
                         rd(c.mu) &&
                         waitlevel << c.mu
        ensures valid
        ensures get() == x
        ensures unfolding valid in rd(c.mu)
    {
        ...
        acquire c;  // succeededs due to the precondition
        ...
    }
}
```

At every given time the following theorem is valid:

$$\forall o : \texttt{waitlevel} \sqsubset o.\mu \implies !\texttt{holds}(o)$$

It is possible to run Chalice without the checks done for deadlock avoidance using the flag `-noDeadlockChecks`. Using that, Chalice will ignore the check of the `mu` field during acquire and a manual proof that the current thread has not already acquired the object has to be supplied. This is useful in case of cyclic datastructures that cannot be fully ordered with an ordering relation, like in section 4.2.

When objects are not shared, the `mu` field of the object is set to the special value `lockbottom`. Thus, an alternative way to express that an object `o` is not shared is by:

```
assert o.mu == lockbottom;
```

Another useful feature of Chalice is the built-in type of *sequences*, also known as immutable lists. Figure 2.4 displays the basic syntax constructs of sequences in Chalice. The operations on sequences are functional: they don't modify the sequence, just return a new altered copy. Sequences are zero-indexed. The syntax of the sequence update, shown in figure 2.4, (*seq*`[`*idx*`:=`*value*`]`) is a new proposed extension to Chalice, developed during this thesis. The sequence type is handy in expressing properties of datastructures, especially lists. For example the interface of a linked list datastructure could be the one shown in listing 2.8. Access permissions are omitted for simplicity. With the addition of `toSeq()` the functions `at()` and `length()` are obsolete.

```
class Node
{
    var value: int;
    var next: Node;

    function toSeq(): seq<int>
    { [value] ++ ite(next != null, next.toSeq, []) }

    function at(i: int): int        // value at index
        ensures i < |toSeq()|
    { toSeq()[i] }

    function length(): int          // length of the list
    { |toSeq()| }

    method deleteAll()              // empties the list
        ensures toSeq() == []
    { ... }

    method append(x: int)           // append an element to the end of the list
        ensures toSeq() == old(toSeq()) ++ [x]
    { ... }

    method update(i: int, x: int)  // update elemet at index i with value x
        ensures toSeq() == old(toSeq())[i := x]
    { ... }
}
```

Listing 2.8: Interface of linked lists expressed in terms of sequences

In chapter 3 we introduce the new concept of backpointers added in Chalice during this thesis. For a more detailed presentation of Chalice see [Lein09a].

`seq<`*type*`>`	The type of sequence with elements of type *type*
`[]`	The empty sequence
`[`*e1*`, `*e2*`, `*e3*`]`	A sequence with elements *e1*, *e2* and *e3*
seq1 `++` *seq2*	The concatenation of the two sequences *seq1* and *seq2*
seq`[`*idx*`]`	The *idx*'th element of *seq*
seq`[`*idx1*`..`*idx2*`]`	A subsequence of *seq* from index *idx1* to *idx2* (exclusive)
seq`[..`*idx*`]`	The first *idx* elements of subsequence *seq*
seq`[`*idx*`..]`	The sequence *seq* after dropping the first *idx* elements
seq`[`*idx*`:=`*value*`]`	A copy of *seq* with the element at index *idx* replaced with *value*
`|`*seq*`|`	The length of the sequence *seq*

Figure 2.4: Sequence syntax of Chalice

2.5 Translating Chalice into Boogie

In this section we sketch the methodology used to translate Chalice into Boogie code. Fields of Chalice are encoded as values of the polymorphic type `Field a`, where `a` is the type of the field. All the locations/objects of the heap are of type `ref`. The entire heap of the program is encoded in a polymorphic partial map in Boogie, which is indexed by locations and fields and thus encoding the state of all the objects of Chalice. The Boogie type of this map is called `HeapType` and the Boogie variable is called `Heap`

All permissions are encoded as a pair of integer (p, n): p is a percentage of the full 100% permission hence ranging from 0 to 100 and n is how many epsilon permissions we have, raging from $-\infty$ to $+\infty$. We can visually think of the permission as: $p + n\varepsilon\%$. The permissions of all fields at a given point are encoded as another map, the `Mask`. This map is also indexed by locations and fields, but returns the permission in the form of a map from `PermissionComponent` to int. `PermissionComponent` has two values, `perm$R` and `perm$N`. All the definitions described until now are in the *prelude* of every Boogie file generated by Chalice:

```
1  type ref;
2  const null: ref;
3  type Field a;
4
5  type HeapType = <a>[ref,Field a]a;
6  var Heap: HeapType;
7
8  type PermissionComponent;
9  const unique perm$R: PermissionComponent;
10 const unique perm$N: PermissionComponent;
11 type MaskType = <a>[ref,Field a][PermissionComponent]int;
12 var Mask: MaskType where IsGoodMask(Mask);
13 const ZeroMask: MaskType;
14 axiom (forall<T> o: ref, f: Field T ::
15   ZeroMask[o,f][perm$R] == 0 && ZeroMask[o,f][perm$N] == 0);
```

The various permission types have corresponding translations given the aforementioned encoding:

$$\begin{aligned}
\texttt{acc}(o.x) &\equiv \texttt{Mask}[o,\ O.x]\texttt{[perm\$R] == 100 \&\& Mask}[o,\ O.x]\texttt{[perm\$N] == 0}\\
\texttt{rd}(o.x) &\equiv \texttt{Mask}[o,\ O.x]\texttt{[perm\$R] > 0 || Mask}[o,\ O.x]\texttt{[perm\$N] > 0}\\
\texttt{acc}(o.x,\ p) &\equiv \texttt{Mask}[o,\ O.x]\texttt{[perm\$R] == } p \texttt{ \&\& Mask}[o,\ O.x]\texttt{[perm\$N] == 0}\\
\texttt{acc}(o.x,\ \texttt{rd}(n)) &\equiv \texttt{Mask}[o,\ O.x]\texttt{[perm\$R] == 0 \&\& Mask}[o,\ O.x]\texttt{[perm\$N] == } n\\
\texttt{acc}(o.x,\ p\ \texttt{-}\ \texttt{rd}(n)) &\equiv \texttt{Mask}[o,\ O.x]\texttt{[perm\$R] == } p \texttt{ \&\& Mask}[o,\ O.x]\texttt{[perm\$N] == -}n
\end{aligned}$$

Here we consider o been an instance of class O with a field x.

From now on we use the following shorthands:

$$\begin{aligned}
\textit{CanRead(o.x)} &\equiv o \texttt{ != null \&\&}\\
&\quad \texttt{Mask}[o,\ O.x]\texttt{[perm\$R] == 0 \&\& Mask}[o,\ O.x]\texttt{[perm\$N] == 0}\\
\textit{CanWrite(o.x)} &\equiv o \texttt{ != null \&\&}\\
&\quad \texttt{Mask}[o,\ O.x]\texttt{[perm\$R] > 0 || Mask}[o,\ O.x]\texttt{[perm\$N] > 0}
\end{aligned}$$

Permissions can be consumed and returned by statements. The transfer of permissions is done with two operations of the translation: *exhale* and *inhale*. Both operations are defined by structural induction on the shape of expressions.

Roughly speaking, *Exhale(E)* checks that expression E holds, more specifically, that the current thread holds the permissions required by E, and then takes away these permissions. *Inhale(E)* assumes E and transfers the permissions required by E to the current thread. If the current thread obtains some permission for a location `o.f` for which it previously had no permission, *Inhale* assigns an arbitrary value to `o.f`, which models the fact that another thread might have modified the location since the current thread last accessed it (*framing rule*). The definitions for both operations are shown in figure 2.5. With *Tr(E)* we denote the translation of simple Chalice expressions into Boogie which is almost identical. *Tr(E)* should never be called on an expression E containing a permission. These cases are forbidden (for example using a permission in the left-hand-side of an implies operator).

The translation of Chalice includes the translation of the special fields *held* and μ for each object. These fields are directly translated to Boogie as ordinary fields. `held` is a boolean field and `mu` has the special type `Mu`. As mentioned before, a partial order relation $\sqsubset$ on values of `Mu` is defined and translated as the Boogie function `MuBelow`. The `mu` field of non-shared objects has the special value `LockBottom`. `waitlevel` is a special local variable of type `mu` translated into the max of all the `mu` fields of the object the current thread holds. For example the expression `waitlevel` $\sqsubset u$ is translated to:

```
(forall p: ref :: Heap[p, held] ==> IsBelow(Heap[p, mu], u))
```

The fragment of the prelude regarding the aforementioned translation follows:

```
type Mu;
const unique mu: Field Mu;
function MuBelow(Mu, Mu) returns (bool);  // strict partial order
axiom (forall m: Mu, n: Mu ::
  { MuBelow(m,n), MuBelow(n,m) }
    !(MuBelow(m,n) && MuBelow(n,m)));
axiom (forall m: Mu, n: Mu, o: Mu ::
  { MuBelow(m,n), MuBelow(n,o) }
    MuBelow(m,n) && MuBelow(n,o) ==> MuBelow(m,o));
const $LockBottom: Mu;
axiom (forall m, n: Mu :: MuBelow(m, n) ==> n != $LockBottom);

const unique held: Field bool;
```

Exhale(`acc(`*E.f*`,` *p* `+ rd(`*n*`))`*) ≡*

```
  assert Mask[Tr(E), f][perm$R] >= Tr(p) &&
         (Mask[Tr(E), f][perm$R] == Tr(p) ==> Mask[Tr(E), f][perm$N] >= Tr(n));
  Mask[Tr(E), f][perm$R] := Mask[Tr(E), f][perm$R] - Tr(p);
  Mask[Tr(E), f][perm$N] := Mask[Tr(E), f][perm$N] - Tr(n);
```

Exhale(`rd(`*E.f*`)`*) ≡*

```
  assert Mask[Tr(E), f][perm$R] > 0;
  var p: int; havoc p;
  assume p > 0 && p < Mask[Tr(E), f][perm$R];
  Mask[Tr(E), f][perm$R] := Mask[Tr(E), f][perm$R] - p;
```

Inhale(`acc(`*E.f*`,` *p* `+ rd(`*n*`))`*) ≡*

```
  if(Mask[Tr(E), f][perm$R] == 0 && Mask[Tr(E), f][perm$N] == 0)
    { havoc Heap[Tr(E), f]; }    //Modeled as IsGoodInhaleState
  Mask[Tr(E), f][perm$R] := Mask[Tr(E), f][perm$R] + Tr(p);
  Mask[Tr(E), f][perm$N] := Mask[Tr(E), f][perm$N] + Tr(n);
```

Inhale(`rd(`*E.f*`)`*) ≡*

```
  if(Mask[Tr(E), f][perm$R] == 0 && Mask[Tr(E), f][perm$N] == 0)
    { havoc Heap[Tr(E), f]; }    //Modeled as IsGoodInhaleState
  var p: int; havoc p;
  assume p > 0 && Mask[Tr(E), f][perm$R] + p < 100;
  Mask[Tr(E), f][perm$R] := Mask[Tr(E), f][perm$R] + p;
```

Exhale(P `&&` *Q) ≡*
Exhale(Q);
Exhale(P);

Exhale(P `==>` *Q) ≡*
`if(`*Tr(P)*`)` { *Exhale(Q)*; }

Otherwise:
Exhale(E) ≡
`assert` *Tr(E)*;

Inhale(P `&&` *Q) ≡*
Inhale(Q);
Inhale(P);

Inhale(P `==>` *Q) ≡*
`if(`*Tr(P)*`)` { *Inhale(Q)*; }

Otherwise:
Inhale(E) ≡
`assume` *Tr(E)*;

Figure 2.5: Exhale and Inhale operations

```
x := new C; ≡
  havoc x;
  assume x != null;
  assume Heap[x, held] == false;
  assume Heap[x, mu] == lockbottom;
  //Foreach field f of class C:
  assume Mask[x, C.f][perm$R] == 100;
  assume Mask[x, C.f][perm$N] == 0;
  //Foreach reference field f
  //of class C:
   assume Heap[x, C.f] == null;[15]
```

```
x := o.f; ≡
  assert CanRead(o.f);
  x := Heap[o, f];
```

```
fork o.foo(); ≡
  Exhale(o.Precondition<foo>);
```

```
call o.foo(); ≡
  Exhale(o.Precondition<foo>);
  Inhale(o.Postcondition<foo>);
```

```
acquire o; ≡
  assert CanRead(o.mu);
  (forall p: ref :: Heap[p, held] ==>
   MuBelow(Heap[p, mu], Heap[o, mu]));
  Heap[o, held] := true;
  Inhale(o.invariant(*));
```

```
while (cond)
  invariant inv;
{ body } ≡
  Exhale(inv);
  havoc variables<body>;
  if(cond) {
    Mask := ZeroMask;
    Inhale(inv);
    Tr(body);
  } else {   // assume !cond
    Inhale(inv);
    // The rest of the method body
  }
```

```
o.f := x; ≡
  assert CanWrite(o.f);
  Heap[o, f] := x;
```

```
join o.foo(); ≡
  Inhale(o.Postcondition<foo>);
```

```
share o between p and s; ≡
  assert CanWrite(o, mu);
  assert Heap[n, mu] == lockbottom;
  assert MuBelow(p, s);
  var w: Mu; havoc w;
  assume MuBelow(p, w) && MuBelow(w, s);
  Heap[o, mu] := w;
  Exhale(o.invariant(*));
```

```
release o; ≡
  assert o != null;
  assert Heap[o, held];
  Exhale(o.invariant(*));
  Heap[o, held] := false;
```

Figure 2.6: Translation of Chalice statements into Boogie code

The translation of most Chalice statements is shown in figure 2.6. In [Lein09b] one can find more information on the implementation of Chalice and its translation in Boogie code. Also, for more recent developments in Chalice see [Heul12, Heul13].

[15] This initialization was implemented during this thesis.

Chapter 3

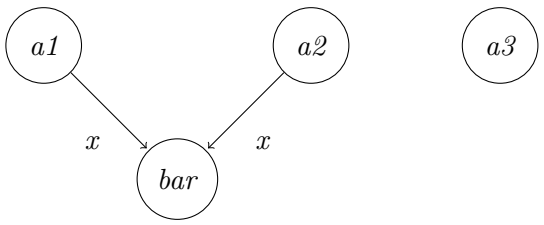
Backpointers in Chalice

3.1 Backpointers

A backpointer[1] defines the reverse relation of a class field. It is the set of all objects pointing to a specific object.

```
1  class A {
2    var x: B;
3  }
4
5  class B { ... }
6
7  ...
8  ...
9  var bar: B;
10 bar := new B;
11 ...
```

In the listing above, class `A` has a field `x` pointing to an object of type `B`. For an instance of class `B` e.g. `bar`, the backpointer $(A.x)^{-1}$ is the set of all the objects of type `A` pointing to `bar` through the field `x`.



In the situation displayed in the figure above, the backpointer set `bar`.$(A.x)^{-1}$ consists of objects *a1* and *a2*, and *not a3*. In Chalice, fields of classes that can potentially have backpointers are declared by using the `tracked` modifier.

Backpointers are considered a ghost field, maintained by Chalice. A backpointer on an object `bar` is denoted by `bar.~(A.x)`. As any other field, backpointers have access permissions; in order to assign a value to a tracked field it is required to gain not only the access permission for that field but also access permission on all the backpointer fields that are modified, as demonstrated in the code snippet 3.1 where a value is assigned in line 11.

3.2 Backpointer encoding

Backpointers are encoded as regular fields in Boogie code. For every tracked field `A.x` a backpointer field named `A.x$BP` is defined. This encoding of backpointers as Boogie state

[1] Also introduced in [Kass12a] in terms of separation logic

```
class A {
    tracked var x: B;

    method test(newX: B)
        requires acc(x) && acc(x.~(A.x)) &&
                                acc(newX.~(A.x))
        ensures acc(x) && acc(x.~(A.x)) &&
                                acc(newX.~(A.x))
        ensures x == newX
    {
        this.x := newX
        assert this in newX.~(A.x)
    }
}
```

Listing 3.1: Simple example with backpointers

makes our methodology easier to implement, since assignments in Boogie code are easily managed. The Boogie code in listing 3.3 describes the tracked field `A.x` in listing 3.2.

```
class A {
  tracked var x: B;
}
```

Listing 3.2: Chalice code

```
const unique A.x: Field (ref);
const unique A.x$BP: Field (BackpointerSet);
axiom isBackpointerField(A.x$BP);
axiom isBackpointerFieldOf(A.x$BP, A.x);
```

Listing 3.3: Boogie code produced from Chalice code 3.2

The set of the objects belonging to the backpointer are encoded as a map from references to booleans as displayed in listing 3.4. Two helper functions, `isBackpointerField` and `isBackpointerFieldOf`, define the relationship of ordinary fields with backpointer fields. Equality on backpointer sets is also defined.

```
type BackpointerSet = <a>[BackpointerSetComponent a]a;
type BackpointerSetComponent a;
const unique BackpointerSetComponent#Set: BackpointerSetComponent ([ref]bool);
function isBackpointerField<a>(bpf: Field(a)) returns (bool);
function isBackpointerFieldOf(bpf: Field(BackpointerSet),
                              f: Field(ref)) returns (bool);
function BackpointerSet#Equal(a: BackpointerSet, b: BackpointerSet) returns (bool)
{
    a[BackpointerSetComponent#Set] == b[BackpointerSetComponent#Set]
}
```

Listing 3.4: Boogie backpointer prelude

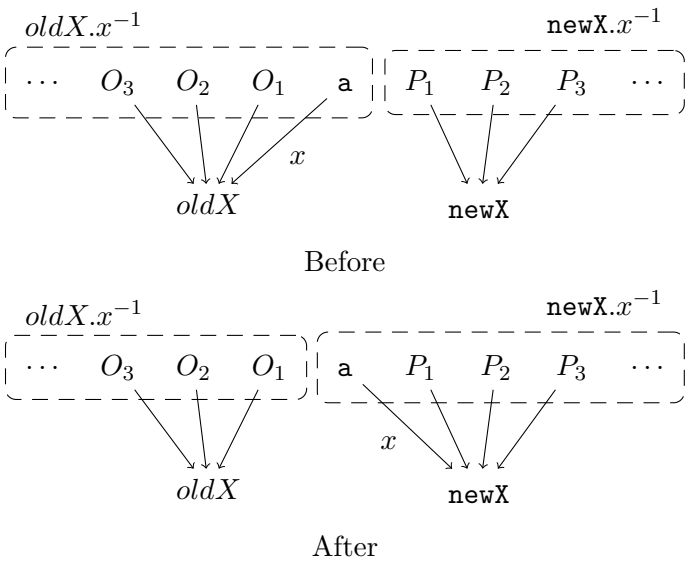


Figure 3.1: Assignment on a tracked field

An assignment on a tracked field:

```
1   a.x := newX;
```

is performed in three sequential steps:

1. Remove `a` from `a.x`'s backpointer set
2. Add `a` to `newX`'s backpointer set
3. The actual assignment

The situation is depicted graphically in figure 3.1. The produced Boogie code for the above operation is shown in listing 3.5.

```
1   // Remove from old object's backpointer set
2   if (Heap[this, A.x] != null) {
3     assert {:msg "  8.9: LHS.~(A.x): location  might not be writable"}
4       CanWrite(Mask, Heap[a, A.x], A.x$BP);
5     Heap[Heap[a, A.x], A.x$BP][BackpointerSetComponent#Set][a] := false;
6   }
7
8   // Add to new object's backpointer set
9   if (newX != null) {
10    assert {:msg "  8.9: RHS.~(A.x): location might not be writable"}
11      CanWrite(Mask, newX, A.x$BP);
12    Heap[newX, A.x$BP][BackpointerSetComponent#Set][a] := true;
13  }
14
15  // update field next
16  assert {:msg "  8.9: Location might not be writable"}
17    CanWrite(Mask, a, A.x);
18  Heap[a, A.x] := newX;
```

Listing 3.5: Boogie code for assignment on tracked field

As shown in the Boogie code in listing 3.5 to perform a single assignment, three write permissions should be acquired: one for each backpointer set that is affected and one for the tracked field where the assignment is performed.

3.3 Aggregates

By using the aforementioned additions to Chalice it is now possible to use aggregates on backpointer sets such as:

1. Cardinality
2. Sum
3. Min
4. Max
5. Product
6. Average

Currently only the first four aggregates are fully implemented and documented in Chalice; sum, min, max, product and average are performed over integer fields of objects in the backpointer set.

```
class Node {
  method test1()
    requires acc(next) && next == null {
    var x: Node;
    x := new Node;
    assert |x.~(Node.next)| == 0
    this.next := x;
    assert |x.~(Node.next)| == 1
  }
}
```

Listing 3.6: Using the cardinality of a backpointer set

The syntax of cardinality over backpointer sets is shown in listing 3.6. The syntax is borrowed by the sequence feature of Chalice. The syntax used for the rest aggregates is:

(*aggregate_name running_variable*:*backpointer_set* :: *running_variable*.*integer_field*)

At the moment, only expressions with the above format are allowed in the body of an aggregate. An example of usage is shown in listing 3.7.

```
class Node {
    tracked var next: Node;
    var x: int;
    method test2()
      requires acc(next) && next == null
               && acc(x) && x == 42 {
      var newNode: Node;
      newNode := new Node;
      this.next := newNode;
      assert (sum o in newNode.~(Node.next) :: o.x) == 42
      this.x := 24;
      assert (sum o in newNode.~(Node.next) :: o.x) == 24
    }
}
```

Listing 3.7: Sum aggregate usage

In order to frame a sum such as `(sum o in this.~(Node.next) :: o.x)` , besides the permission in the backpointer set `this.~(Node.next)` we should have permission in all fields `x` of objects `o` that belong in the backpointer set. Therefore, the framing permission for sum is `rd(this.~(Node.next)) &&` ($\forall$ $o \in$ `this.~(Node.next)` . `rd(`o`.x)`). The special syntax introduced in Chalice for the bulk permission is: `acc(this.~(Node.next).x)` (see section 3.5).

To implement the aggregates we extended the implementation for backpointers that was shown above in the following way:

```
const unique BackpointerSetComponent#Cardinality: BackpointerSetComponent (int);
const unique BackpointerSetComponent#Sums: BackpointerSetComponent ([Field int]int);
const unique BackpointerSetComponent#Mins: BackpointerSetComponent ([Field int]int);
const unique BackpointerSetComponent#Maxs: BackpointerSetComponent ([Field int]int);
function BackpointerSet#Equal(a: BackpointerSet, b: BackpointerSet) returns (bool)
{
  a[BackpointerSetComponent#Cardinality] == b[BackpointerSetComponent#Cardinality]
  &&      a[BackpointerSetComponent#Set] == b[BackpointerSetComponent#Set]
}
```

Listing 3.8: Boogie backpointer prelude (aggregates)

One backpointer set component for each aggregate is used. Aggregates, like sum and max, that are defined in terms of a integer field are modeled as maps indexed by that field. We extend the definition of backpointer equality to include only cardinality (see section 3.5).

Chalice maintains aggregates of backpointer sets. Aggregates change when an object is added or removed from the set. In an assignment, one object is deleted and one is added (provided that there is no `null` reference involved). Assume a backpointer set *bps*, and an object *a* that is added or removed from *bps*. In pseudo Boogie code these two operations are shown in listing 3.9 (`BackpointerSetComponent` abbreviated as `BSC`) which the revised version of listing 3.5 with aggregates taken into account.

Sum, min and max of sets are also affected by assignments on the respective integer fields of objects that belong to the set. Chalice's implementation maintains these fields when such an assignment happens. For example the assignment:

```
    a.n := newN;
```

where `a` of type `A` with an integer field `n`, is translated into the code in listing 3.10. This code is produced for each tracked field of class `A`. The treatment of min and max in our implementation is incomplete but it is sufficient for most cases.

```
addRefToBackpointerSet(bps, a) {
  bps[BPC$Set][a] := true;
  bps[BPC$Cardinality] := bps[BPC$Cardinality] + 1;
  // for each int field n of a
  bps[BPC$Sums][A.n] := bps[BPC$Sums][n] + Heap[a, A.n];
  if(bps[BSC$Mins][A.n] > Heap[a, A.n])
    bps[BSC$Mins][A.n] := Heap[a, A.n];
  if(bps[BSC$Maxs][A.n] < Heap[a, A.n])
    bps[BSC$Maxs][A.n] := Heap[a, A.n];
  // end for each
}

removeRefFromBackpointerSet(bps, a) {
  bps[BPC$Set][a] := false;
  bps[BPC$Cardinality] := bps[BPC$Cardinality] - 1;
  // for each int field n of a
  bps[BPC$Sums][A.n] := bps[BPC$Sums][n] - Heap[a, A.n];
  if(bps[BSC$Mins][A.n] == Heap[a, A.n])
    havoc bps[BSC$Mins][A.n];
  assume bps[BSC$Mins][A.n] <= Heap[a, A.n];
  if(bps[BSC$Maxs][A.n] == Heap[a, A.n])
    havoc bps[BSC$Maxs][A.n];
  assume bps[BSC$Maxs][A.n] >= Heap[a, A.n];
  // end for each
}


  // Remove from old object's backpointer set
  if (Heap[this, A.x] != null) {
    assert {:msg ''  8.9: LHS.~(A.x): location  might not be writable''}
      CanWrite(Mask, Heap[a, A.x], A.x$BP);
    removeRefFromBackpointerSet(Heap[Heap[a, A.x], A.x$BP], a);
  }

  // Add to new object's backpointer set
  if (newX != null) {
    assert {:msg ''  8.9: RHS.~(A.x): location might not be writable''}
      CanWrite(Mask, newX, A.x$BP);
    addRefToBackpointerSet(Heap[newX, A.x$BP], a);
  }

  // update field next
  assert {:msg ''  8.9: Location might not be writable''}
    CanWrite(Mask, a, A.x);
  Heap[a, A.x] := newX;
```

Listing 3.9: Boogie code for assignment on tracked field with aggregates

```
1  // for each tracked field x of a
2  // a belongs the backpointer set bps
3  // bps ≡ Heap[Heap[a, A.x], A.x$BP]
4  // Sums
5  bps[BPC$Sums][A.n] := bps[BPC$Sums][A.n] - Heap[a, A.n] + newN;
6  // Mins
7  var oldMin: int;
8  oldMin := bps[BPC$Mins][A.n];
9  havoc bps[BPC$Mins][A.n];
10 assume (oldMin >= newN) ==> (bps[BPC$Mins][A.n] == newN);
11 assume (oldMin < newN && oldMin != Heap[a, A.n])
12        ==> (bps[BPC$Mins][A.n] == oldMin);
13 assume (oldMin < newN && oldMin == Heap[a, A.n])
14        ==> (newN >= bps[BPC$Mins][A.n] && bps[BPC$Mins][A.n] >= oldMin);
15 // Maxs
16 var oldMax: int;
17 oldMax := bps[BPC$Maxs][A.n];
18 havoc bps[BPC$Maxs][A.n];
19 assume (oldMax <= newN) ==> (bps[BPC$Maxs][A.n] == newN);
20 assume (oldMax > newN && oldMax != Heap[a, A.n])
21        ==> (bps[BPC$Maxs][A.n] == oldMax);
22 assume (oldMax > newN && oldMax == Heap[a, A.n])
23        ==> (newN <= bps[BPC$Maxs][A.n] && bps[BPC$Maxs][A.n] <= oldMax);
24 // end for each
25 Heap[a, A.n] := newN;
```

Listing 3.10: Boogie code: changing an int field of an object that belongs to a backpointer

3.4 Axioms

The backpointer theory is founded on top of certain axioms. Those axioms are often a crucial part of the verification process since they define the behavior of backpointers as well as the aggregates on the backpointer sets. The following axioms make the treatment of backpointers as ghost state more complete and provide a way to prove specifications involving backpointers through axiomatization.

- The first axiom defines backpointers:

  Axiom 1. Definitional axiom
  for a tracked field A.f:

  $$\forall \{a, b\} \in A \times B \ . \ a.f = b \quad \textit{iff} \quad a \in b.(A.f)^{-1}$$

  In other words, a points to b if and only if a belongs to the corresponding backpointer set of b. It is implemented in Boogie with the following code

```
axiom (forall h: HeapType, m: MaskType, sm: MaskType, f: Field(ref),
       a: ref, b: ref, bpf: Field(BackpointerSet) ::
    (wf(h, m, sm) && isBackpointerFieldOf(bpf, f) &&
     a != null && b != null) ==>
    (h[a, f] == b <==> h[b, bpf][BackpointerSetComponent#Set][a]));
```

In all the axioms we take special care for the case of the null pointer.

- All cardinalities are positive integers or zero. This is expressed in our theory with the second axiom:

  Axiom 2. Cardinality is not negative
  for a tracked field A.f:

  $$\forall b \in B \ . \ \left|b.(A.f)^{-1}\right| \geq 0$$

  and in Boogie code:

```
axiom (forall h: HeapType, m: MaskType, sm: MaskType, bps: BackpointerSet ::
    bps[BackpointerSetComponent#Cardinality] >= 0);
```

In the following axioms we describe an empty backpointer set.

- If a set has cardinality zero (i.e. is an empty set) no object can belong to this backpointer set. This is expressed by the following axiom:

  Axiom 3. Belongs to empty set

  $$\forall \ bps \in BP, a \in A \ . \ |bps| = 0 \implies a \notin bps,$$

  BP is the class of all backpointer sets and A a class A.

  and in Boogie code:

```
axiom (forall h: HeapType, m: MaskType, sm: MaskType,
       bps: BackpointerSet, a: ref ::
    (wf(h, m, sm) && a != null) ==>
     (bps[BackpointerSetComponent#Cardinality] == 0 ==>
      !bps[BackpointerSetComponent#Set][a]));
```

Most axioms that include an implication can be read in an contrapositive way. For example this axiom can be read as "if there is some object that belongs to the set then the cardinality cannot be zero". This is a useful way of thinking in the process of verification.

- If a set has cardinality zero then every sum on that set is zero:

  Axiom 4. Sum of empty set

  $$\forall\ bps \in BP\ .\ |bps| = 0 \implies \sum_{\alpha \in bps} \alpha.n = 0,$$

  BP is the class of all backpointer sets and $A.n$ a integer field of class A.

  and in Boogie code:

```
axiom (forall h: HeapType, m: MaskType, sm: MaskType,
       bps: BackpointerSet, n: Field(int) ::
    wf(h, m, sm) ==>
     (bps[BackpointerSetComponent#Cardinality] == 0 ==>
      bps[BackpointerSetComponent#Sums][n] == 0));
```

The following axioms are used in case of an unit set (i.e. a set with only one element).

- This axiom asserts that no two distinct objects can belong to a unit set.

  Axiom 5. Two members of unit set

  $$\forall\ bps \in BP, a1 \in A, a2 \in A\ .\ |bps| = 1 \wedge a1 \in bps \wedge a2 \in bps \implies a1 = a2,$$

  BP is the class of all backpointer sets and A a class A.

  Due to the special treatment of universal quantification in Z3, and SMT solvers in general, this axiom is encoded in two different forms in Boogie:

```
axiom (forall h: HeapType, m: MaskType, sm: MaskType,
       bps: BackpointerSet, a: ref ::
    (wf(h, m, sm) && a != null) ==>
    ((bps[BackpointerSetComponent#Set][a] &&
      bps[BackpointerSetComponent#Cardinality] == 1) ==>
       (forall b: ref ::
        (b != null && bps[BackpointerSetComponent#Set][b]) ==>
          a == b)));
```

```
axiom (forall h: HeapType, m: MaskType, sm: MaskType,
       bps: BackpointerSet, a: ref, b: ref ::
    (wf(h, m, sm) && a != null && b != null) ==>
    ((bps[BackpointerSetComponent#Set][a] &&
      bps[BackpointerSetComponent#Set][b] &&
      bps[BackpointerSetComponent#Cardinality] == 1) ==>
       a == b));
```

Each version is specialized in verifying different assertions.

- When we know that a set has only one element and if we have an object belonging to that set then we know the value of all aggregates on that set. Hence the following axioms:

  Axiom 6. Sum of unit set

  $$\forall\ bps \in BP, a \in A\ .\ |bps| = 1 \land a \in bps \implies \sum_{\alpha \in bps} \alpha.n = a.n,$$

  BP is the class of all backpointer sets and $A.n$ an integer field of class A.

  Axiom 7. Max of unit set

  $$\forall\ bps \in BP, a \in A\ .\ |bps| = 1 \land a \in bps \implies \max_{\alpha \in bps} \alpha.n = a.n,$$

  BP is the class of all backpointer sets and $A.n$ an integer field of class A.

  Axiom 8. Min of unit set

  $$\forall\ bps \in BP, a \in A\ .\ |bps| = 1 \land a \in bps \implies \min_{\alpha \in bps} \alpha.n = a.n$$

  ,
  BP is the class of all backpointer sets and $A.n$ an integer field of class A.

  The corresponding code in Boogie is:

```
axiom (forall h: HeapType, m: MaskType, sm: MaskType,
       bps: BackpointerSet, n: Field(int), a: ref ::
    (wf(h, m, sm) && a != null) ==>
    ((bps[BackpointerSetComponent#Cardinality] == 1 &&
      bps[BackpointerSetComponent#Set][a]) ==>
       (bps[BackpointerSetComponent#Sums][n] == h[a, n])));
```

```
axiom (forall h: HeapType, m: MaskType, sm: MaskType,
       bps: BackpointerSet, n: Field(int), a: ref ::
    (wf(h, m, sm) && a != null) ==>
    ((bps[BackpointerSetComponent#Cardinality] == 1 &&
      bps[BackpointerSetComponent#Set][a]) ==>
       (bps[BackpointerSetComponent#Mins][n] == h[a, n])));
```

```
axiom (forall h: HeapType, m: MaskType, sm: MaskType,
       bps: BackpointerSet, n: Field(int), a: ref ::
    (wf(h, m, sm) && a != null) ==>
    ((bps[BackpointerSetComponent#Cardinality] == 1 &&
      bps[BackpointerSetComponent#Set][a]) ==>
       (bps[BackpointerSetComponent#Maxs][n] == h[a, n])));
```

3.5 Framing syntax

When using a aggregate on a backpointer set, the frame of this expression is not only the access permission to the backpointer field but also the permission on the field that is aggregated over all the objects that belong to the backpointer set. This special bulk permission is written as:

```
1   acc(o.∼(A.x).n)   or   rd(o.∼(A.x).n)   or   acc(o.∼(A.x).n, 50)
```

for all the objects of the backpointer set `o.∼(A.x)` on the integer field `n`. For the first expression, it is just like writing (with some variations in the implementation; it uses triggers internally to improve the prover's speed):

```
1   (forall i:A :: i in a.∼(A.x) ==> acc(i.n))
```

This syntax is used in any kind of permission.

It takes a lot of time to check bulk permissions by the prover, especially for Chalice's framing check (see section: 2.5), where we check for the necessary permissions for each field in order to infer if there has been a change by a statement. So, by default, Chalice assumes that all the aggregates are lost during statements like this, unless the programmer explicitly specifies that Chalice should check for this bulk permission. The equality relation used in the inhale implementation is the one defined in section 3.3. The syntax introduced is a predicate following the statement written with a triple &. For example, if we want the call to a method `foo()` to preserve the aggregates of the backpointer set `o.∼(A.x)` on a field `n` we should write the following:

```
1   call x.foo() &&& acc(o.∼(A.x).n)
```

forcing Chalice to make an extra check if this permission is indeed preserved by the call and if so, to assume that all the aggregates of the set `o.∼(A.x)` on the field `n` are the same as before the call. The statements that support this *framing syntax* are:

- while
- share
- release
- lock (not mentioned here)
- send (not mentioned here)
- method call
- method fork

3.6 Triggering - "use axiom" syntax

The set of axioms presented in section 3.4 are not a complete definition of the aggregates specified but makes a good case for most examples that we explored. For a more detailed presentation of aggregates and a discussion of triggering issues see [Lein09c].

Even so, these axioms make heavy use of universal quantification and thus lower the performance of the prover. In our implementation, the developer is responsible for introducing

most of the axioms, enabling him to fine-tune the prover. Whenever an axiom is needed by the prover, the user has to make a fake call to a function on a backpointer set that will trigger the usage of the axiom. The parameters of the function call are the values of the universally quantified variables in the body of the axiom. Some axioms take as a parameter a field and for those axioms we extended Chalice to include references to class fields, prefixing the qualified name of the field with the "at" sign (@). Some examples of triggering axioms are shown in listing 3.11.

```
class Lala {
    ghost tracked var x: Mama;
    var n: int;
}

method BPReversibility1(z: Lala)
    requires z != null && acc(z.x) && z.x == this && acc(~(Lala.x))
    requires holds(this; A)
    ensures acc(~(Lala.x)) && z in ~(Lala.x)
{
    assert z.useDefinitionalAxiom(@Lala.x, this);
}

method BPReversibility2(z: Lala)
    requires z != null && acc(z.x) && acc(~(Lala.x)) && z in ~(Lala.x)
    ensures acc(z.x) && z.x == this
{
    assert z.useDefinitionalAxiom(@Lala.x, this);
}

method EmptySetCardinality3(r: Lala)
    requires acc(~(Lala.x)) && r != null && r in ~(Lala.x)
    ensures acc(~(Lala.x)) && |~(Lala.x)| > 0
{
    assert ~(Lala.x).useAxiomBelongsToEmptySet(r);
    assert ~(Lala.x).useAxiomCardinalityNotNegative();
}

method UnitSetSum2(z: Lala)
    requires z != null && acc(~(Lala.x)) && z in ~(Lala.x) && rd(z.n)
                       && rd(~(Lala.x).n) && (sum y in ~(Lala.x) :: y.n) != z.n
    ensures z != null && acc(~(Lala.x)) && |~(Lala.x)| > 1
{
    assert ~(Lala.x).useAxiomSumOfUnitSet(z, @Lala.n);
    assert ~(Lala.x).useAxiomBelongsToEmptySet(z);
    assert ~(Lala.x).useAxiomCardinalityNotNegative();
}
```

Listing 3.11: Example usage of "use-axiom" syntax

The definitional axiom for an object `x`, that points through the field `f` to an object `y`, is used on the object `x` and the two parameters of the function are first the reference to the field `f` and the other object `y`. All the other "axiom trigger functions" are called on the backpointer set they refer to. The list of functions that are supported, along with the corresponding axioms, are shown in figure 3.2.

Axiom 1	<*object*>.useDefinitionalAxiom(<*field*>, <*object*>)
Axiom 2	<*BPSet*>.useAxiomCardinalityNotNegative()
Axiom 3	<*BPSet*>.useAxiomBelongsToEmptySet(<*object*>)
Axiom 4	<*BPSet*>.useAxiomSumOfEmptySet(<*int field*>)
Axiom 5 first form	<*BPSet*>.useAxiomTwoMembersOfUnitSet1(<*object*>)
Axiom 5 second form	<*BPSet*>.useAxiomTwoMembersOfUnitSet2(<*object*>, <*object*>)
Axiom 6	<*BPSet*>.useAxiomSumOfUnitSet(<*object*>, <*int field*>)
Axiom 7	<*BPSet*>.useAxiomMaxOfUnitSet(<*object*>, <*int field*>)
Axiom 8	<*BPSet*>.useAxiomMinOfUnitSet(<*object*>, <*int field*>)

Figure 3.2: Triggering functions per axiom

Chapter 4

Applications of Backpointers

Backpointers may be of use in various areas of verification problems. Some problem examples include:

1. Copy-on-Write lists
2. Double linked lists
3. Composite
4. Union-Find
5. Priority inheritance protocol (PIP)

Fictional disjointness is a datastructure feature where the interface of the datastructures promises to the user that each object is distinct while the objects share data in the implementation level. This category of problems is of great interest in the verification area and many attempts have been made to prove programs as it provides a verification challenge not only for automated verification but also for verification with the use of proof assistants.

A notable example from this family is the snapshotable trees that were examined in [Mehn12]. By using proof assistants (Coq in particular) it was proved that the implementation presented there is a fictional disjointed datastructure consisting of a mutable tree structure were snapshots of the tree could be taken for later use. The snapshots of the tree are immutable subsequent changes of the tree. Furthermore, four variations of the trees were presented from which only one's correctness was proved while the rest are left as a verification challenge.

We mainly studied a similar problem from the area of fictional disjointed problems, the Copy-on-Write list datastructure.

4.1 Copy-on-Write Lists

This datastructure implements a mutable list interface. At any moment the user may create a copy of a list which is guaranteed that it will be distinct and separate from the original list; in other words, disjoint. This implies that a change in the original list will not affect the copied list and vice versa. However, the implementation optimizes the way the lists are stored in order to minimize the consumed memory by sharing parts of lists. In particular, the copied lists share with the original the longest common tail.

As displayed in figure 4.1 in order to make a copy of list $L1$, $L2$, we can simply make a shallow copy of $L1$ into $L2$. Here the sentinel objects (or interface nodes) are shown with squares and the actual nodes of the implementation are shown with circles. As it is observable, the user has the illusion that a second, independent list has been created with `L2` being its sentinel node while the pointer `L1` is still accessible. However, as it is shown, the actual implementation saves memory space by having both of `L1` and `L2` point on the same list.

Adding a head node to any list is easy because the new node created is not shared with any other list. Changing a value at a specific node of a list is more complicated. For example

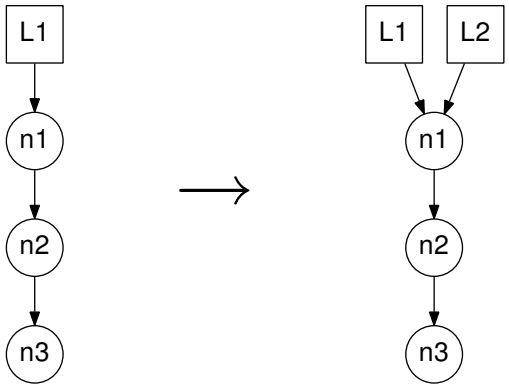


Figure 4.1: Making a copy of a copy-on-write list

consider the example in figure 4.2. We want to change the first element of $L2$, $n2$. The fact that nodes $n3$, $n4$ and $n5$ are shared with list $L1$ is irrelevant since the node we want to change, $n2$, is not shared and can be done in-place. However, it is not possible to change $n4$ (the third element of list $L2$, figure 4.3) in-place because that would also change the third element of $L1$. As a result, the shared part of the list's implementation, up to the point we want to change, nodes $n3$ and $n4$, must be copied to new nodes $n3*$ and $n4*$, as shown in figure 4.3.

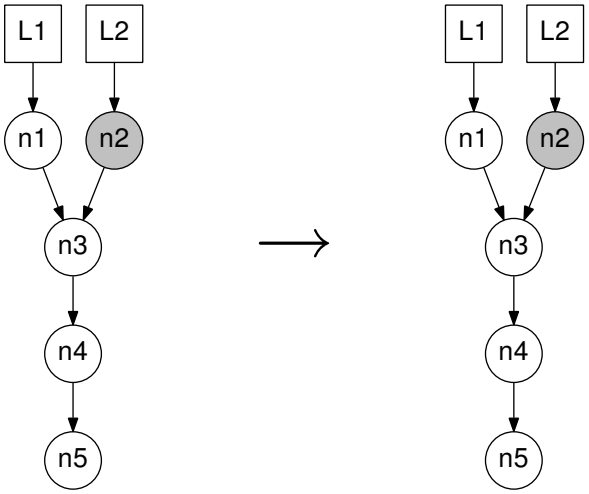


Figure 4.2: Changing a non-shared node ($n2$); no copying

4.1.1 Programmer's code

The problem we encounter is determining whether a node is shared with another list. To solve this, the programmer uses reference counting, more specifically a form of local reference counting using an integer field, `refCount`, that keeps track of how many objects point to a node. These objects can either be node objects or list objects, the sentinel class. In the example above, the usage of `refCount` is shown in figure 4.4, with the field `refCount` abbreviated as *rc*. However it is not possible to deduce if a node is shared or not just from that. To utilize this information, the algorithm traverses the nodes of the list up to the node we wish to change. At any point, if a node with `refCount` > 0 is found, we infer that all nodes from that point on are shared and the implementation starts copying them. Note that the field `refCount` is a real field existing in the program without specifications. Also, while we copy the nodes, the

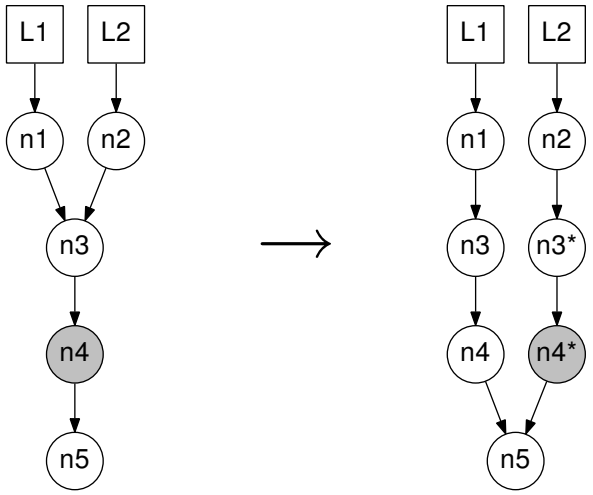


Figure 4.3: Changing a shared node ($n4$) implies copying

original nodes have their `refCount` field decreased by one since list $L2$ no longer "owns" them. Listing 4.1 is the implementation of the aforementioned datastructure in Chalice syntax.

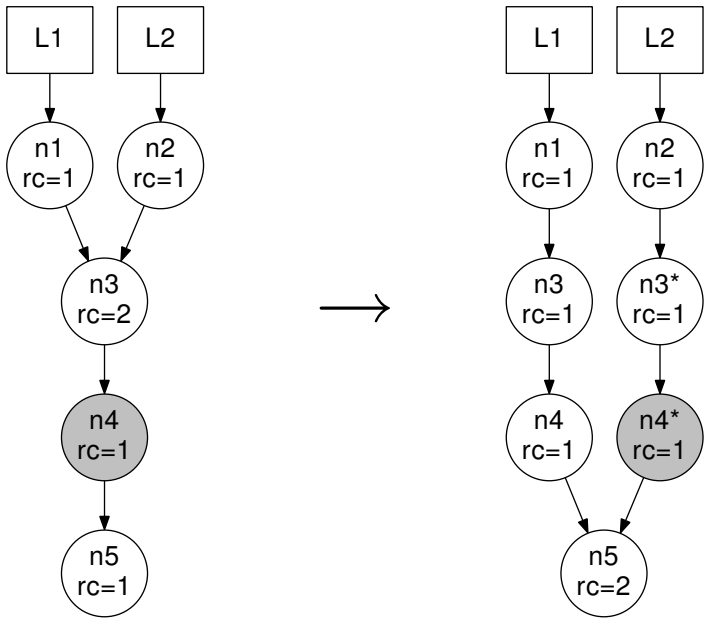


Figure 4.4: Changing a shared node with `refCount`

```
1  class List
2  {
3    var head:Node
4
5    function toSeq(): seq<int>
6    {
7      head == null ? [] : head.toSeq()
8    }
9
10   method initEmpty()
11   {
12     share this;
13   }
14
```

```
15    method initCopy(other: List)
16    {
17      share acquire this;
18      acquire other;
19      if(other.head != null)
20      {
21        acquire other.head;
22        head := other.head;
23        head.refCount := head.refCount + 1;
24        release head;
25      }
26      release other;
27      release this;
28    }
29
30    method insert(value: int)
31    {
32      var n: Node;
33      n := new Node;
34      n.value := value;
35      if(head != null) { acquire head; }
36      n.next := head;
37      head := n;
38      head.refCount := 1;
39      if(head.next != null) { release head.next; }
40      share head;
41    }
42
43    method set(index: int, value: int)
44    {
45      var h: Node;
46      if(head != null)
47      {
48        acquire head;
49        if(head.refCount == 1)
50        {
51          call head.set(index, value);
52        }
53        else
54        {
55          head.refCount := head.refCount - 1;
56          h := head;
57          head := new Node;
58          call h.copySet(index, value, head);
59        }
60      }
61    }
62  }
63
64
65  class Node
66  {
67    var value: int;
68    var next: Node;
69    var refCount: int;
70
71    function toSeq(): seq<int>
72    {
73      [value] ++ (next == null ? [] : next.toSeq())
74    }
75
76    method set(index: int, newValue: int)
77    {
```

```
78      var h: Node;
79
80      if (index == 0)
81      {
82        value := newValue;
83        release this;
84      }
85      else
86      {
87        if(next != null)
88        {
89          acquire next;
90          if (next.refCount == 1)
91          {
92            release this;
93            call next.set(index - 1, newValue);
94          }
95          else
96          {
97            next.refCount := next.refCount - 1;
98            h := next;
99            next := new Node;
100           call h.copySet(index - 1, newValue, next);
101           release this;
102         }
103       }
104       else /* next == null (index out of bounds) */
105       {
106         release this;
107       }
108     }
109   }
110
111   method copySet(index: int, newValue: int, newNode: Node)
112   {
113     var h: Node
114
115     newNode.refCount := 1;
116     if(next != null)
117     {
118       acquire next
119     }
120     if(index == 0)
121     {
122       newNode.value := newValue
123       newNode.next := next
124       if(next != null)
125       {
126         next.refCount := next.refCount + 1
127         release next;
128       }
129     }
130     else
131     {
132       newNode.value := value;
133       if(next != null)
134       {
135         newNode.next := new Node;
136         call next.copySet(index - 1, newValue, newNode.next);
137       }
138       else
139       {
140         newNode.next:=null;
```

```
141         }
142       }
143       share newNode;
144       release this;
145     }
146 }
```

Listing 4.1: Copy-on-Write lists: Program listing

There are a few things worth mentioning about the implementation.

- Whenever an index is given that is out of bounds the implementation ignores the call.
- The interface of the `List` class contains four methods and a function.
- There are two *constructor* methods `List#initEmpty` and `List#initCopy`. The former constructs an empty list, encoded with `head == null`, while the latter makes a copy of an existing list by simply using the same `head` as the other list.
- The other two methods mutate a list. `List#insert` prepends an element in a list while `List#set` changes an element at `index` with the new value being the parameter `value`.
- The function `List#toSeq` simply translates `List`s to Chalice's native sequences and is a pure function.
- The implementation is concurrent. The objects that are going to be examined or changed are first locked by an acquire statement and latter released.
- The `Node`'s implementation is split into two methods. Method `Node#set` traverses the nodes of the list until a node with `refCount` > 0 is found. Method `Node#copySet` is called which makes a copy (line 135) of each node it visits generating a copy of the original sequence in `newNode` parameter. Both methods stop when they find the node that needs to be changed. If that happens in `Node#set` then the change is done in place since the node is not shared. Otherwise, in method `Node#copySet` the copy of the node is altered and it is set to point to the rest of the old list structure.
- The algorithm locks all the nodes as it traverses them in order to ensure thread safety and then releases them in the reverse order.
- The algorithm is expressed in a recursive form and not in iterative form.
- The `List` objects are *not* thread-safe themselves, but the internal implementations is, meaning that using two different lists for different threads is safe even thought the two implementation may share some nodes.

4.1.2 Interface Specification

We will first examine the specifications of the sentinel class and then specify the interface class' behavior which is the target of our verification process. The specifications of the List class are shown in listing 4.2 in which the highlighted pre- and post-conditions ensure the absence of deadlocks and can be ignored.We assume the existence of a predicate `inv` that is a token used by client code to abstractly store the necessary permissions. The two constructor methods absorb the permissions of the newly created list object and make up the `inv` token. The mutation methods require an `inv` token and return it in the end. The specification is given in terms of the function `toSeq`. `initCopy` ensures that the list that is copied does not change and that the new list is the same as the `other`. Methods `insert` and `set` use the sequence notation to encode the desired specification. Notably, the specification of `set` encodes the behavior in case `index` is out of bounds (i.e. `index >= |toSeq()|`) .

```
class List
{
  predicate inv { ... }

  function toSeq(): seq<int>
    requires inv
  {
    ...
  }

  method initEmpty()
    requires acc(head) && acc(sequence) && head == null
    requires acc(mu) && mu == lockbottom
    ensures inv
    ensures rd*(mu) && waitlevel << mu
    ensures toSeq() == []
  {
    ...
  }

  method initCopy(other: List)
    requires acc(head) && acc(sequence) && head == null
    requires acc(mu) && mu == lockbottom
    requires other != null && this != other && other.inv
    requires rd*(other.mu) && waitlevel << other.mu
    ensures inv && other.inv
    ensures rd*(mu) && waitlevel << mu
    ensures rd*(other.mu) && waitlevel << other.mu
    ensures toSeq() == other.toSeq()
    ensures other.toSeq() == old(other.toSeq())
  {
    ...
  }

  method insert(value: int)
    requires inv
    requires rd*(mu) && waitlevel << mu
    ensures inv
    ensures rd*(mu)
    ensures toSeq() == [value] ++ old(toSeq())
  {
    ...
  }

  method set(index: int, value: int)
    requires inv && 0 <= index
    requires rd*(mu) && waitlevel << mu;
    ensures inv
    ensures rd*(mu)
    ensures index < old(|toSeq()|)
          ? toSeq() == old(toSeq()[index := value])
          : toSeq() == old(toSeq())
  {
    ...
  }
}
```

Listing 4.2: Specification of the `List` interface

With this specification we can now use the `List` class with a client and reason about the behavior of copy-on-write lists. A sample client is show in listing 4.3.

```
class Main
{
  method main()
  {
    var l1:List
    var l2:List

    l1 := new List
    call l1.initEmpty()
    assert l1.toSeq() == []
    call l1.insert(42)
    assert l1.toSeq()[0] == 42
    call l1.insert(43)
    assert l1.toSeq()[0] == 43
    assert l1.toSeq()[1] == 42
    l2 := new List
    call l2.initCopy(l1)
    assert l2.toSeq() != []
    assert l2.toSeq()[0] == 43
    assert l2.toSeq()[1] == 42
    call l1.set(0, -42)
    assert l1.toSeq()[0] == -42
    assert l2.toSeq()[0] == 43
    assert |l1.toSeq()| == 2
    assert |l2.toSeq()| == 2
    call l2.insert(1337)
    assert l2.toSeq()[0] == 1337
    assert l2.toSeq()[1] == 43
    assert l2.toSeq()[2] == 42
    assert l1.toSeq()[0] == -42
    assert l1.toSeq()[1] == 42
    assert |l1.toSeq()| == 2
    assert |l2.toSeq()| == 3
    call l1.set(1, -99)
    assert l1.toSeq()[1] == -99
    assert l2.toSeq()[2] == 42
  }
}
```

Listing 4.3: Simple client for the `List` interface in listing 4.2

4.1.3 Verification

In order for the `List` class to be able to access the nodes of the list, the sentinel node must have read permissions to all the nodes that belong to the list. In particular we can give one epsilon permission to each sentinel on the fields `value` and `next` of each `Node` transitively accessible by the sentinel. We do that by a recursive predicate:

```
predicate inv {
    acc(value, rd(1)) && acc(next, rd(1)) &&
    (next != null ==>  next.inv)
}
```

We use counting permission in order to know exactly how many epsilons we have given away to sentinels. To achieve this we introduced a new field, `transRefCount`, that denotes how many `List` objects have access to a node. Using this field we can write the remaining permission to a `Node` as:

```
acc(value, 100-rd(transRefCount)) && acc(next, 100-rd(transRefCount))
```

Using this counting scheme and knowing that the field `transRefCount` = 1 we can deduce that, when combined with the epsilon permission from the sentinel, we have full (write) access to the node; in other words we can prove that the node is not shared. Note here that `transRefCount` is a field introduced for verification and no "real" code can depend on it (i.e. a ghost field). The example in figure 4.4 with `transRefCount` introduced, abbreviated as *trc*, in shown in figure 4.5

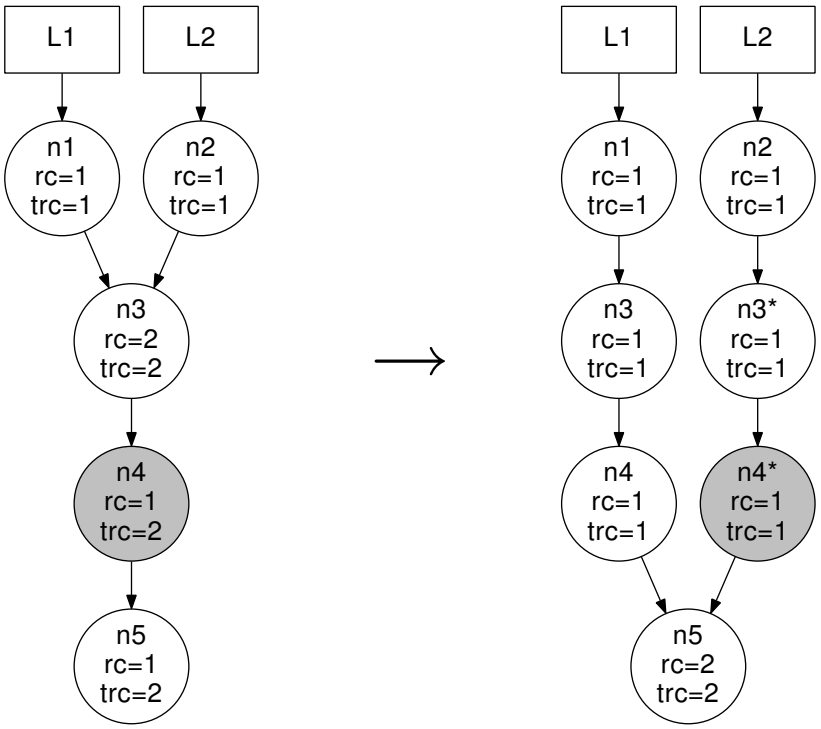


Figure 4.5: Changing a shared node with `refCount` and `transRefCount`

The theorem that we want to prove about the programmer's code is that as the algorithm iterates the nodes of the list, as long as we visit nodes with `refCount` = 1 these nodes are not shared. For our verification model (using `transRefCount` to model sharing nodes) this is translated to deducing that, assuming that `transRefCount` = 1 for the current node and that `refCount` = 1 for the next node, `transRefCount` = 1 for then next node. In compact form:

$$this.transRefCount = 1 \wedge next.refCount = 1 \implies next.transRefCount = 1$$

This is not a trivial theorem! Our proposed solution is the use of *backpointers*. Using the extensions described in chapter 3 we can write two monitor invariants that can lead to the desired conclusion.

$$this.refCount = \left|this.(Node.next^{-1})\right| + \left|this.(List.head^{-1})\right|$$

$$this.transRefCount = \sum_{n \in this.(Node.next^{-1})} n.transRefCount + \left|this.(List.head^{-1})\right|$$

These invariants are always valid and describe the semantics of `refCount` and `transRefCount`. They are especially useful in case `this.transRefCount` = 1 and `this.next.refCount` = 1.

Bellow we provide an outline of the proof. Our premise is that $this.transRefCount = 1$ and $next.refCount = 1$. We know that $this.next \neq null$ therefore, by using the definitional axiom (1), we infer that $this \in next.(Node.next)^{-1}$. Using both the contrapositive of axiom 3 (on $next.(Node.next)^{-1}$ and $this$) and axiom 2 we get that:

$$\left|next.(Node.next^{-1})\right| = 1 \wedge \left|next.(List.head^{-1})\right| = 0$$

from the first invariant on $next$. from the second invariant on $next$ we can deduce that:

$$next.transRefCount = \sum_{n \in next.(Node.next^{-1})} n.transRefCount$$

Using axiom 6 on $next.(Node.next^{-1})$ since we know that $\left|next.(Node.next^{-1})\right| = 1$ and $this \in next.(Node.next)^{-1}$ we get:

$$next.transRefCount = \sum_{n \in next.(Node.next^{-1})} n.transRefCount = this.transRefCount$$

and thus $next.transRefCount = 1$.

So, in the end, the invariant for the `Node` becomes:

```
1     invariant A: acc(refCount,50) && acc(transRC,50) &&
2                  acc(next, 100-rd(transRC)) && acc(value, 100-rd(transRC));
3     invariant B: refCount == |~(Node.next)| + |~(List.head)|;
4     invariant C: transRC == (sum x in ~(Node.next) :: x.transRC) + |~(List.head)|;
5     invariant D: (next==null ==> acc(refCount,50) && acc(transRC,50));
6     invariant E: (forall o:~(Node.next) :: acc(o.refCount,50) && acc(o.transRC,50));
7     invariant F: acc(~Node.next) && acc(~List.head);
```

The A part of the invariant gives the necessary permissions to `refCount` and `transRC` for the current node as well as (read) access to `value` and `next` with counting permissions as described before. We need access to the fields `refCount` and `transRC` of our "parents" in relation to `Node.next` and `List.head` and because of that we have to split the full permission on those fields and give 50% in the invariant of the current node and 50% to the invariant of the object `this.next`. That is the E part of the invariant, expressed with backpointers. If the `this.next` field is `null` then we should not loose any permissions, so we keep the 50% of the full permission to the invariant of current node and that is part D. Parts B and C are the invariants used for the proof of the theorem done above. Finally, in part F, we give permissions to the backpointer fields of this `Node.next` and `List.head`.

When attaching a new `List` object to some `Node`, during the `initCopy()` method we have to make up a new `Node.inv` for that list. The way this is done is by increasing the `transRefCount` by one, recursively, through the rest of the list. This is done with the `addOneToTransRefCount()` method. This is not an actual method of the implementation, it is a ghost method operating on the ghost field `transRefCount`. The method takes the invariant broken at C and fixes it by returning an extra `inv` predicate. This way the new `List` gains interest in the nodes of it's implementation. The method is shown is listing 4.4.

```
1  method addOneToTransRefCount()
2      requires inv
3      requires holds(this; broken C) && unfolding inv in waitlevel==mu
4      requires transRefCount ==
5               (sum n in ~(Node.next) :: n.transRefCount) + |~(List.head)| - 1
6      ensures inv && inv
7      ensures holds(this; *) && unfolding inv in waitlevel==mu
8      ensures toSeq() == old(toSeq())
```

```
    ensures ~(Node.next)==old(~(Node.next)) && ~(List.head)==old(~(List.head))
{
    unfold inv
    if(next!=null)
    {
        acquire this.next
        assert useDefinitionalAxiom(@Node.next, next)
    }
    transRefCount := transRefCount + 1;
    if(next!=null)
    {
        call next.addOneToTransRefCount()  &&& rd(~(Node.next).transRefCount)
        release this.next   &&& rd(~(Node.next).transRefCount)
    }
    fold inv
    fold inv
}
```

Listing 4.4: The `addOneToTransRefCount` method

Losing interest to nodes of the list happens when we make a new copy of the nodes during `copySet()`. This is done inside the method while traversing the nodes of the list in contrast with `addOneToTransRefCount()`. Again the precondition of the method takes the invariant broken at C and fixes it by removing one from `transRefCount` and releasing one epsilon permission into the monitor invariant of the node.

Full listing of the actual code used in verification can be found in appendix A.

4.2 Priority Inheritance Protocol (PIP)

The priority inheritance protocol [Sha90] is an algorithm and datastructure from the domain of system programming about synchronizing tasks executing in parallel multiprocessors. From the point of view of formal methods [Summ09] PIP is a very interesting example of a practical datastructure that has non-local invariants. PIP is, abstractly, a graph structure represented by `Node` objects, in which a `Node` has at most one parent and where cycles in the parent relation are possible (see figure 4.6). In listing 4.5 we can see the programmer's code for the PIP datastructure.

```
class Node
{
    var parent: Node;
    var initVal: int;
    var value: int;

    method init(val: int)
    {
        parent := null;
        value := val;
        initVal := val;
    }

    method acquire(nd: Node)
    {
        nd.parent := this;
        call this.update(nd.value);
    }

    method update(newValue: int)
    {
        if(newValue > value)
```

```
23          {
24              if(parent != null)
25              { parent.update(value); }
26          }
27      }
28  }
```

Listing 4.5: The priority inheritance protocol user code

The transitive invariant that all `Node`s must obey is:

$$node.value = max(\{node.initVal\} \cup \{n.value \mid node \in n(.parent)^*\})$$

stating that the current node's `value` must be the maximum of its `initVal` value, and the `value` of all its (transitive) descendants. This property for the nodes is obviously true given the implementation in listing 4.5. It is interesting to note that it is a non-local invariant; the invariant on `node` depends on objects not accessible from `node`.

In [Summ09] four solutions to PIP are given one of which is the usage of a ghost variable to track the "backpointer" of the parent relation. In the paper it is dismissed in favor of *considerate reasoning* [Summ10] because there are no guarantees that the ghost set of objects is correctly maintained. The formulation of backpointers in this thesis does exactly that.

We have implemented the PIP using the formalism of backpointers in Chalice. The above invariant, written in our formalism, is:

```
1      value == (|~(Node.parent)|==0 ?
2                initVal :
3                max(initVal, (max c in ~(Node.parent) :: c.value)))
```

We use the aggregate *max* directly on the backpointer field for the descendants relation. The specification of `update()`, in the spirit of considerate reasoning, takes the invariant broken and in the postcondition returns it fixed.

A problem we encountered was the fact that PIP is potentially a cyclic datastructure. The current implementation of Chalice prohibits cyclic datastructures because all objects that can be shared must be in a partial ordering relation, due to the restriction imposed

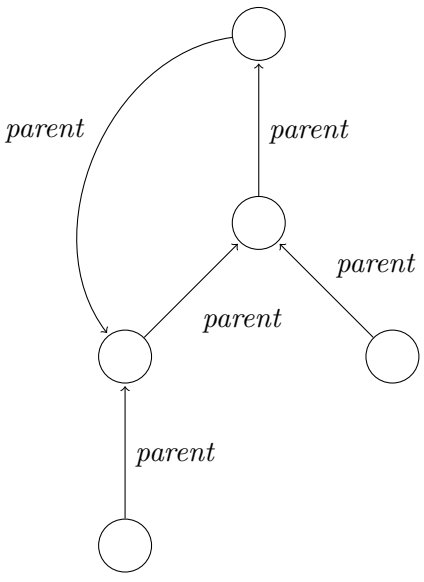


Figure 4.6: An instance of PIP

by the deadlock avoidance mechanism. We had to circumvent this obstacle and verify PIP without the guarantee of deadlock freedom.

The full code listing of the implementation of PIP can be found in appendix B.

Chapter 5

Conclusion

In this chapter we discuss the contribution of this work in the field of formal specification and verification, and in particular in the field of invariant disciplines and the research on observational disjointness. We compare our work with other methodologies targeting similar problems. Finally, we also examine the ways in which this work can be extended in the future.

5.1 Results

We developed a verification methodology based on backpointers. We implemented it in the verification language Chalice and solved some issues that were caused by implementing this discipline in the framework of implicit dynamic frames with fractional and counting permissions. We revisited our initial approach on backpointers and simplified it by relying on the already existing infrastructure of field access permissions and implicit dynamic frames of Chalice instead of using complex rules to deduce when an assignment is permitted.

After successfully implementing our extension we tried the new formulation on some challenging problems, namely copy-one-write lists and the priority inheritance protocol. We found that our methodology succeeded in encoding the specifications of those datastructures, which have the common characteristic that their object invariants rely on objects that are non-accessible from the local frame. To the best of our knowledge this is the first time such an attempt based on automated proving is made for a problem of the general category of fictional disjoint datastructures.

An importand issue we resolved was optimizing the verification process to shorten the verification times which were, in the beginning, too large to be practical, even for small programs. We modified the encoding and the implementation of backpointers many times in order to guide the prover. Many triggering issues had to be solved and some times an incomplete axiomatization was needed, in favor of better verification times.

Despite our efforts, one branch of the copy-one-write lists implementation remains unproved, probably due to a matching loop in the automated prover. We were not successful in finding and eliminating the conditions that led the prover into a loop.

5.2 Related work

The backpointer methodology is in effect an *invariant discipline*. It is a set of rules ensuring that the definitional axiom of a backpointer's ghost field is always valid, i.e. that $r \in r.f.f^{-1}$ in any state, for any non-null record r and tracked field f such that $r.f \neq \texttt{null}$. We call conditions like these *system invariants*. Some other conditions are given by the programmer, for example *object* or *monitor invariants*. There are several ways of treating program-specific invariants, mostly focusing on the special case of object invariants [Dros08]. Various forms of ownership [Lein04, Barn05, Mull02] are popular invariant disciplines.

There was an academic debate on the necessity and usefulness of invariant disciplines. [Park07] stated the case in favor of using abstract predicates. In [Summ09] the authors argued

that even with the usage of abstract predicates, object invariant are still useful in expressing properties of objects in an object oriented way that does better job in separating concerns in specification. In the same paper the term *global invariant* is used to describe object invariants that are closely related to invariants concerning backpointers in Chalice. Their verification of PIP [Sha90] uses a "backpointer" ghost field to localize the non-local property of this datastructure. It seems, the authors argue, that the priority inheritance protocol example is not easy to handle with abstract predicates alone.

Our verification methodology used in verifying the copy-on-write lists case is heavily inspired by *considerate reasoning* [Summ10]. Their specification and verification technique based on object invariants provides a way for method specifications to "notify" all interested parties about an object invariant that might break. Our specification and verification of `addOneToTransRefCount` is an adaptation of their `addToTotal` method from the implementation of the composite pattern.

Observational disjointness is a field of study that posses many challenges on formal verification. Separation logics have been very successful in describing the part of the heap that a data structure resides in, but have many problems when it comes to datastructures that are observationally disjoint but share some internal representation. Our approach based on implicit dynamic frames and fractional/counting permission enables a more flexible description of the heap. In particular, bookkeeping of reference counting through backpointers and counting permissions is essential to the copy-on-write lists case study.

Concurrent abstract predicates [Dins10] is a theory where special predicates called *capabilities* allow exclusive access to a shared portion of the heap, thus supporting hidden sharing of state. This approach has been successful in specifying and verifying indexing datastructures [da R11] however, our work does not emulate the theory of CAPs in dealing with these problems. After all, it is not clear how CAPs would handle copy-on-write lists. Backpointers and CAPs are orthogonal methodologies, and one could, in the future, combine both of them in solving even more difficult problems.

Fictional Separation Logic [Jens12] is an ambitious mathematical framework that allows the implementer to choose their own separation algebra as part of the implementation. This idea completely decouples heap disjointness from separating conjunction. The use of fractional permissions as well as other examples of observational disjointness are shown to be special cases of this very general methodology. The generality comes at the price of complexity at the part of the implementer, so it remains an open question if this idea scales up to reasonably-sized programs. Furthermore, it seems that fictional separation logic has no provision for object and monitor invariants, nor does it provide the means of mentioning unreachable parts of the heap, like we do.

In [Mehn12] another problem from the family of fictional disjoint data structures, the *snapshotable trees*, is proposed as a challenge. In this example clients see the same mutable tree and can take immutable snapshots of the tree at any time. All snapshots and the tree appear to be heap-disjoint, but, in fact, the implementation uses lazy copying and shares as much as possible. There are four different versions of the structure, one of which is verified by the authors, using whole-heap predicates (and therefore restricting it to sequential programs). Compared to copy-on-write lists the implementation of snapshotable trees is easier since each copy of the tree (snapshot) is immutable and thus no resorting to reference counting is needed. In the terminology of [Dris86], snapshotable trees are *partially persistent*, while copy-on-write lists are *fully persistent*. Snapshotable trees could be verified using the framework of Chalice without the usage of backpointers, which is not true for the case of copy-on-write lists.

5.3 Future work

One future extension to this work could be the complete verification of the copy-on-write lists example, possibly by identifying and fixing any remaining triggering issues in the current implementation of backpointers in Chalice. An even further inspection of the implementation for matching loops could lead to an even better performance from the prover.

One could extend the specification language of backpointers in many ways. One approach could be the exploration of other commonly used patterns of ghost fields that, along with some discipline, could make non-local properties accessible to the specifications. Another possibility could be to extend the list of aggregate functions currently supported by the implementation or even to lift the restriction on the kinds of expressions that can appear inside the body of the aggregate. Note that enabling referencing fields of fields in the body of aggregates introduces the risk of *abstract aliasing* [Lein02]. Last, a more comprehensive set of axioms, inspired from [Lein09c], could be implemented.

In conjunction with some techniques that could reduce or infer the, at the moment, necessary permission annotations in the specification, one could dramatically reduce the specifications needed by even the simplest of examples that uses backpointers as is the formalism at the moment in Chalice.

The current version of our methodology, binded with the overall framework of Chalice does not deal with inheritance and subtyping, features commonly found in object oriented languages. This is a topic that could be explored in the future.

Finally, the current interface specification of copy-on-write lists exposes a subtle but very important implementation detail to the client: the usage of locks. Furthermore, they impose a non-trivial requirement about the locking behavior of the client. This is due to technicalities of the Chalice deadlock-avoidance features. A future work could be to find a way to hide this information from the client, possibly by refining the deadlock avoidance methodology in Chalice.

Appendix A

Copy-on-Write lists program listing

Here is the complete listing of Copy-on-Write lists that we tried to verify.

```
class List
{
  tracked var head:Node
  var sequence:seq<int>

  predicate inv
  {
       acc(head) && acc(sequence) && rd*(mu)
    && (head != null ? head.inv && sequence==head.toSeq() && rd*(head.mu) && mu <<
        head.mu
                     : sequence==[]
        )
  }

  function toSeq():seq<int>
    requires inv
  { unfolding inv in sequence }

  method initEmpty()
    requires acc(head) && acc(sequence) && acc(mu) && mu==lockbottom && head==null
    ensures inv && rd*(mu) && waitlevel<<mu
    ensures toSeq() == []
  {
    share this above waitlevel
    sequence:=[]
    fold inv
  }

  method initCopy(other:List)
    requires acc(head) && acc(sequence) && acc(mu) && mu==lockbottom && head==null
    requires other!=null && this!=other && other.inv && rd*(other.mu) && waitlevel
        <<other.mu
    ensures inv && other.inv
    ensures rd*(mu) && waitlevel<<mu
    ensures rd*(other.mu) && waitlevel<<other.mu
    ensures toSeq() == other.toSeq()
    ensures other.toSeq() == old(other.toSeq())
  {
    unfold other.inv
    sequence:=other.sequence
    share this between waitlevel and other.mu
    acquire this
    acquire other
    if(other.head!=null)
    {
      acquire other.head
      head := other.head
      head.refCount := head.refCount + 1
```

```
47        call head.addOneToTransRefCount()
48        release head
49      }
50      release other
51      release this
52      fold other.inv
53      fold inv
54    }
55
56    method insert(value:int)
57      requires inv && rd*(mu) && waitlevel<<mu
58      ensures inv && rd*(mu) && toSeq() == [value]++old(toSeq())
59    {
60      var n:Node;
61      n:=new Node;
62      n.value:=value;
63      unfold inv;
64      if(head!=null) { acquire head; }
65      n.next:=head;
66      head:=n;
67      head.refCount:=1;
68      head.transRefCount:=1;
69      head.sequence:=[value]++sequence;
70      sequence:=[value]++sequence;
71      if(head.next!=null) { release head.next &&& rd(head.~(Node.next).transRefCount)
            }
72      share head between this and head.next
73      fold head.inv
74      fold inv
75    }
76
77    method set(index:int, value:int)
78      requires inv && 0<=index
79      requires rd(mu) && waitlevel<<mu;
80      ensures inv && (index<old(|toSeq()|) ? toSeq()==old(toSeq()[index:=value]) :
            toSeq()==old(toSeq()))
81    {
82      var h:Node
83      unfold inv
84      if(head!=null)
85      {
86        acquire head
87        if(head.refCount==1)
88        {
89          assert useDefinitionalAxiom(@List.head, head)
90          assert head.~(List.head).useAxiomBelongsToEmptySet(this)
91          assert head.~(List.head).useAxiomCardinalityNotNegative()
92          assert head.~(Node.next).useAxiomCardinalityNotNegative()
93          assert head.~(Node.next).useAxiomSumOfEmptySet(@Node.transRefCount)
94          assert head.~(List.head).useAxiomSumOfUnitSet(this, @Node.transRefCount)
95          call head.set(index, value)
96        }
97        else
98        {
99          head.refCount:=head.refCount-1
100         h:=head
101         head:=new Node
102         call h.copySet(index, value, head)
103       }
104       sequence:=head.toSeq()
105     }
106     fold inv
107   }
```

```
}



class Node
{
  var value:int
  var sequence:seq<int>
  tracked var next:Node
  var refCount:int
  var transRefCount:int

  invariant A: acc(refCount) && acc(transRefCount, 50) && acc(value, 100-rd(transRefCount)) && acc(next, 100-rd(transRefCount)) && acc(sequence, 100-rd(transRefCount))
  invariant D: next==null ==> acc(transRefCount, 50)
  invariant F: acc(~(Node.next)) && acc(~(List.head))
  invariant B: refCount == |~(Node.next)| + |~(List.head)|
  invariant E: acc(~(Node.next).transRefCount, 50)
  invariant C: transRefCount == (sum n in ~(Node.next) :: n.transRefCount) + |~(List.head)|

  predicate inv
  {
    acc(value, rd(1)) && acc(next, rd(1)) && acc(sequence, rd(1)) && rd*(mu) && lockbottom != this.mu &&
    (next != null ?  next.inv && rd*(next.mu) && this.mu << next.mu && sequence==[value]++next.toSeq()  :  sequence==[value])
  }

  function toSeq():seq<int>
    requires inv
  { unfolding inv in sequence }

  method set(index: int, newValue:int)
    requires inv && rd*(mu)
    requires holds(this; *) && waitlevel==mu
    requires transRefCount == 1 && refCount == 1
    requires 0<=index
    lockchange this
    ensures inv
    ensures index<old(|toSeq()|) ? toSeq() == old(toSeq()[index:=newValue]) : toSeq() == old(toSeq())
    ensures !holds(this)
  {
    var h:Node
    var O:seq<int>

    unfold inv
    assert sequence==old(toSeq())
    if (index == 0)
    {
      value := newValue
      sequence:=sequence[0:=newValue]
      release this
    }
    else
    {
      if(next!=null)
      {
        acquire next
        if(index<|sequence|) { sequence:=sequence[index:=newValue] }
        if (next.refCount == 1)
```

```
165         {
166           release this  &&& rd(next.~(Node.next).transRefCount)
167 
168           assert useDefinitionalAxiom(@Node.next, next)
169           assert next.~(Node.next).useAxiomBelongsToEmptySet(this)
170           assert next.~(Node.next).useAxiomCardinalityNotNegative()
171                   assert next.~(List.head).useAxiomCardinalityNotNegative()
172           assert next.~(List.head).useAxiomSumOfEmptySet(@Node.transRefCount)
173           assert next.~(Node.next).useAxiomSumOfUnitSet(this, @Node.transRefCount)
174           call next.set(index-1, newValue)
175         }
176         else
177         {
178           next.refCount:=next.refCount-1
179           h:=next
180           assert useDefinitionalAxiom(@Node.next, next)
181           next:=new Node
182                   assert acc(transRefCount)
183                   assert useDefinitionalAxiom(@Node.next, next)
184                   assert next.~(Node.next).useAxiomTwoMembersOfUnitSet1(this)
185           call h.copySet(index-1, newValue, next) &&& rd(~(Node.next).transRefCount
                  )
186           assert refCount == |~(Node.next)| + |~(List.head)|
187           assert transRefCount == (sum n in ~(Node.next) :: n.transRefCount) + |~(
                  List.head)|
188           release this
189         }
190       }
191       else { release this }
192     }
193     O:=sequence
194     fold inv
195     assert toSeq()==O
196   }
197 
198   method copySet(index:int, newValue:int, newNode:Node)
199     requires inv && rd*(mu)
200     requires newNode!=null && acc(newNode.value) && acc(newNode.next) && newNode.
          next==null && acc(newNode.sequence) && acc(newNode.refCount) && acc(newNode
          .transRefCount) && acc(newNode.mu) && newNode.mu==lockbottom
201     requires acc(newNode.~(Node.next)) && acc(newNode.~(List.head)) && |newNode.~(
          Node.next)|+|newNode.~(List.head)|==1
202         requires acc(newNode.~(Node.next).transRefCount,50) && (sum m in newNode.~(
              Node.next) :: m.transRefCount) + |newNode.~(List.head)| == 1
203     requires waitlevel==mu && holds(this; broken C)
204     requires transRefCount == (sum m in ~(Node.next) :: m.transRefCount) + |~(List.
          head)| + 1
205     requires 0<=index
206     lockchange this
207     ensures newNode.inv
208     ensures index<old(|toSeq()|) ==> newNode.toSeq() == old(toSeq()[index:=newValue
          ])
209     ensures index>=old(|toSeq()|) ==> newNode.toSeq() == old(toSeq())
210     ensures rd*(newNode.mu) && rd*(mu) && mu<<newNode.mu
211     ensures !holds(this) && !holds(newNode)
212   {
213     var h:Node
214     var O:seq<int>
215 
216     unfold inv
217     assert sequence==old(toSeq())
218     newNode.refCount:=1
219     newNode.transRefCount:=1
```

```
220     if(next!=null)
221         {
222           acquire next
223           assert useDefinitionalAxiom(@Node.next, next)
224         }
225         assert useDefinitionalAxiom(@Node.next, next)
226     transRefCount:=transRefCount-1
227     if(index==0)
228     {
229       newNode.value:=newValue
230           assert useDefinitionalAxiom(@Node.next, next)
231       newNode.next:=next
232       if(next!=null)
233           {
234         next.refCount:=next.refCount+1
235         release next &&& rd(newNode.~(Node.next).transRefCount) && rd(~(Node.next).
              transRefCount)
236       }
237     }
238     else
239     {
240       assert newNode.transRefCount==1
241       newNode.value:=value
242       if(next!=null)
243       {
244         assert newNode.useDefinitionalAxiom(@Node.next, newNode.next)
245         newNode.next:=new Node
246         call next.copySet(index-1, newValue, newNode.next) &&& rd(~(Node.next).
              transRefCount) && rd(newNode.~(Node.next).transRefCount)
247       }
248       else { newNode.next:=null }
249     }
250     newNode.sequence:= [newNode.value] ++ (newNode.next!=null ? newNode.next.toSeq
          () : [])
251     assert newNode.transRefCount==1
252     share newNode between this and newNode.next &&& rd(~(Node.next).transRefCount)
253     O:=newNode.sequence
254     fold newNode.inv
255     assert newNode.toSeq()==O
256     release this
257     assert index<old(|toSeq()|) ==> newNode.sequence == old(toSeq()[index:=newValue
          ])
258     assert index>=old(|toSeq()|) ==> newNode.toSeq() == old(toSeq())
259   }
260
261   method addOneToTransRefCount()
262     requires inv
263     requires holds(this; broken C) && unfolding inv in waitlevel==mu
264     requires transRefCount == (sum n in ~(Node.next) :: n.transRefCount) + |~(List.
          head)| - 1
265     ensures inv && inv
266     ensures holds(this; *) && unfolding inv in waitlevel==mu
267     ensures toSeq() == old(toSeq())
268     ensures ~(Node.next)==old(~(Node.next)) && ~(List.head)==old(~(List.head))
269   {
270     unfold inv
271     if(next!=null)
272     {
273       acquire this.next
274           assert useDefinitionalAxiom(@Node.next, next)
275     }
276     transRefCount := transRefCount + 1;
277     if(next!=null)
```

```
    {
      call next.addOneToTransRefCount()  &&& rd(~(Node.next).transRefCount)
      release this.next   &&& rd(~(Node.next).transRefCount)
    }
    fold inv
    fold inv
  }
}


class Main
{
  method main()
  {
    var l1:List
    var l2:List

    l1 := new List
    call l1.initEmpty()
    assert l1.toSeq() == []
    call l1.insert(42)
    assert l1.toSeq()[0] == 42
    call l1.insert(43)
    assert l1.toSeq()[0] == 43
    assert l1.toSeq()[1] == 42
    l2 := new List
    call l2.initCopy(l1)
    assert l2.toSeq() != []
    assert l2.toSeq()[0] == 43
    assert l2.toSeq()[1] == 42
    call l1.set(0, -42)
    assert l1.toSeq()[0] == -42
    assert l2.toSeq()[0] == 43
    assert |l1.toSeq()| == 2
    assert |l2.toSeq()| == 2
    call l2.insert(1337)
    assert l2.toSeq()[0] == 1337
    assert l2.toSeq()[1] == 43
    assert l2.toSeq()[2] == 42
    assert l1.toSeq()[0] == -42
    assert l1.toSeq()[1] == 42
    assert |l1.toSeq()| == 2
    assert |l2.toSeq()| == 3
    call l1.set(1, -99)
    assert l1.toSeq()[1] == -99
    assert l2.toSeq()[2] == 42
  }
}
```

Listing A.1: Copy-on-Write lists: complete program listing

Appendix B

Priority inheritance protocol program listing

Here is the complete listing of priority inheritance protocol implemented in Chalice with backpointers.

```
1  class Node
2  {
3    function mx(x:int, y:int):int { x>y ? x : y }
4
5    tracked var parent: Node;
6    var value: int;
7    var initVal: int;
8
9    invariant /* A */ acc(~(Node.parent)) && acc(~(Node.parent).value, 50) && acc(
         value, 50) && rd*(initVal) && acc(parent, 50);
10   invariant /* B */ value == (|~(Node.parent)|==0 ? initVal : mx(initVal, (max c in
          ~(Node.parent) :: c.value)));
11   invariant /* C */ (parent==null ==> acc(value, 50));
12
13   method Init(v:int)
14     requires acc(parent) && parent==null && acc(value) && acc(initVal) && acc(~(
           Node.parent)) && |~(Node.parent)|==0 && this.mu == lockbottom;
15         ensures acc(parent, 50) && lockbottom << this.mu;
16   {
17     initVal:=v;
18         value:=v;
19         share this;
20   }
21
22   method acq(n:Node)
23     requires acc(mu) && acc(n.mu) && lockbottom << this.mu && !holds(this) && acc(
           parent, 50) && parent==null && n!=null && n!=this;
24         ensures rd*(parent) && parent==n;
25   {
26     acquire n;
27         parent:=n;
28         call n.update(value);
29         release n;
30   }
31
32   method update(v:int)
33     requires holds(this);
34     requires /* A */ acc(~(Node.parent)) && acc(~(Node.parent).value, 50) && acc(
           value, 50) && rd*(initVal) && acc(parent, 50);
35     requires v<=value ==> /* B */ value == (|~(Node.parent)|==0 ? initVal : mx(
           initVal, (max c in ~(Node.parent) :: c.value)));
36     requires /* C */ (parent==null ==> acc(value, 50));
37         requires v>value ==> v == mx(initVal, (max c in ~(Node.parent) :: c.value))
               && !holds(parent);
38         ensures holds(this);
39     ensures /* A */ acc(~(Node.parent)) && acc(~(Node.parent).value, 50) && acc(
           value, 50) && rd*(initVal) && acc(parent, 50);
```

```
40      ensures /* B */ value == (|∼(Node.parent)|==0 ? initVal : mx(initVal, (max c in
            ∼(Node.parent) :: c.value)));
41      ensures /* C */ (parent==null ==> acc(value, 50));
42    {
43            if(v>value)
44            {
45              if(parent!=null) { acquire parent; call parent.update(value); }
46                  value:=v;
47                  if(parent!=null) { release parent; }
48            }
49    }
50  }
```

Listing B.1: Priority inheritance protocol: complete program listing

Appendix C

Source code and tools

The source code of the extensions in Chalice, developed during this thesis, in the form of patches as well as information and examples about the usage of backpointers in Chalice can be found under:
`http://www.softlab.ntua.gr/~lkritik/backpointers`

All patches are based on Chalice from Boogie version 9c96c519e063 found here:
`http://boogie.codeplex.com/SourceControl/changeset/9c96c519e063`

Binary builds of Boogie can be found here:
`http://boogie.codeplex.com/`
and binary releases of Z3 here:
`http://z3.codeplex.com/` or
`http://research.microsoft.com/en-us/um/redmond/projects/z3/download.html`

To build Chalice from source use "`./sbt compile`". To run Chalice use one of "`./chalice.sh` file.chalice" or "`chalice.bat` file.chalice" depending on your operating system.

The patches are released under the same license Chalice and Boogie are released, the Microsoft Public License (Ms-PL).

Chalice, Boogie and Z3 are not intellectual property of the author or in any way related to the author or the National Technical University of Athens.

Bibliography

[Bane08] Anindya Banerjee, David A. Naumann and Stan Rosenberg, "Regional Logic for Local Reasoning about Global Invariants", in *Proceedings of the 22nd European conference on Object-Oriented Programming*, ECOOP '08, pp. 387–411, Berlin, Heidelberg, 2008, Springer-Verlag.

[Barn05] Mike Barnett, K. Rustan M. Leino and Wolfram Schulte, "The spec# programming system: an overview", in *Proceedings of the 2004 international conference on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, CASSIS'04, pp. 49–69, Berlin, Heidelberg, 2005, Springer-Verlag.

[Barn06] 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 *Proceedings of the 4th international conference on Formal Methods for Components and Objects*, FMCO'05, pp. 364–387, Berlin, Heidelberg, 2006, Springer-Verlag.

[Barr10a] Clark Barrett, Aaron Stump and Cesare Tinelli, "The SMT-LIB Standard: Version 2.0", in A. Gupta and D. Kroening, editors, *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, UK)*, 2010.

[Barr10b] Clark Barrett, Aaron Stump and Cesare Tinelli, "The SMT-LIB Standard: Version 2.0", Technical report, Department of Computer Science, The University of Iowa, 2010. Available at `www.SMT-LIB.org`.

[Bobo11] François Bobot, Jean-Christophe Filliâtre, Claude Marché and Andrei Paskevich, "Why3: Shepherd Your Herd of Provers", in *Boogie 2011: First International Workshop on Intermediate Verification Languages*, pp. 53–64, Wrocław, Poland, August 2011.

[Born05] Richard Bornat, Cristiano Calcagno, Peter O'Hearn and Matthew Parkinson, "Permission accounting in separation logic", in *Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '05, pp. 259–270, New York, NY, USA, 2005, ACM.

[Boyl03] John Boyland, "Checking interference with fractional permissions", in *Proceedings of the 10th international conference on Static analysis*, SAS'03, pp. 55–72, Berlin, Heidelberg, 2003, Springer-Verlag.

[Cheo02] Yoonsik Cheon and Gary T. Leavens, "A runtime assertion checker for the Java Modeling Language (JML)", in *PROCEEDINGS OF THE INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING RESEARCH AND PRACTICE (SERP '02), LAS VEGAS*, pp. 322–328, CSREA Press, 2002.

[Clar82] Edmund M. Clarke and E. Allen Emerson, "Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic", in *Logic of Programs, Workshop*, pp. 52–71, London, UK, UK, 1982, Springer-Verlag.

[Clar98] David G. Clarke, John M. Potter and James Noble, "Ownership types for flexible alias protection", in *Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '98, pp. 48–64, New York, NY, USA, 1998, ACM.

[Cohe09] Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, Michał Moskal, Thomas Santen, Wolfram Schulte and Stephan Tobies, "VCC: A Practical System for Verifying Concurrent C", in *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics*, TPHOLs '09, pp. 23–42, Berlin, Heidelberg, 2009, Springer-Verlag.

[Cok12] David R. Cok, *The SMT-LIBv2 Language and Tools: A Tutorial*, GrammaTech, Inc., August 2012. Available at `http://www.grammatech.com/resources/smt/SMTLIBTutorial.pdf`.

[Cous77] Patrick Cousot and Radhia Cousot, "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints", in *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '77, pp. 238–252, New York, NY, USA, 1977, ACM.

[da R11] Pedro da Rocha Pinto, Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner and Mark Wheelhouse, "A simple abstraction for complex concurrent indexes", in *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '11, pp. 845–864, New York, NY, USA, 2011, ACM.

[De M08] Leonardo De Moura and Nikolaj Bjørner, "Z3: an efficient SMT solver", in *Proceedings of the Theory and practice of software, 14th international conference on Tools and algorithms for the construction and analysis of systems*, TACAS'08/ETAPS'08, pp. 337–340, Berlin, Heidelberg, 2008, Springer-Verlag.

[Diet07] Werner Dietl, Sophia Drossopoulou and Peter Müller, "Generic universe types", in *Proceedings of the 21st European conference on Object-Oriented Programming*, ECOOP'07, pp. 28–53, Berlin, Heidelberg, 2007, Springer-Verlag.

[Dins10] Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, Matthew J. Parkinson and Viktor Vafeiadis, "Concurrent abstract predicates", in *Proceedings of the 24th European conference on Object-oriented programming*, ECOOP'10, pp. 504–528, Berlin, Heidelberg, 2010, Springer-Verlag.

[Dris86] J R Driscoll, N Sarnak, D D Sleator and R E Tarjan, "Making data structures persistent", in *Proceedings of the eighteenth annual ACM symposium on Theory of computing*, STOC '86, pp. 109–121, New York, NY, USA, 1986, ACM.

[Dros08] S. Drossopoulou, A. Francalanza, P. Müller and A. J. Summers, "A Unified Framework for Verification Techniques for Object Invariants", in *Proceedings of the 22nd European conference on Object-Oriented Programming*, ECOOP '08, pp. 412–437, Berlin, Heidelberg, 2008, Springer-Verlag.

[Fill03] Jean-Christophe Filliâtre, "Why: a multi-language multi-prover verification tool", Research Report 1366, LRI, Université Paris Sud, March 2003. `http://www.lri.fr/~filliatr/ftp/publis/why-tool.ps.gz`.

[Heul11] Stefan Heule, K. Rustan M. Leino, Peter Müller and Alexander J. Summers, "Fractional permissions without the fractions", in *Proceedings of the 13th Workshop on Formal Techniues for Java-Like Programs*, FTfJP '11, pp. 1:1–1:6, New York, NY, USA, 2011, ACM.

[Heul12] S. Heule, I. T. Kassios, P. Müller and A. J. Summers, "Verification Condition Generation for Permission Logics with Abstract Predicates and Abstraction Functions", Technical Report 776, ETH Zurich, Department of Computer Science, 11 2012.

[Heul13] S. Heule, K. R. M. Leino, P. Müller and A. J. Summers, "Abstract Read Permissions: Fractional Permissions without the Fractions", in *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, Lecture Notes in Computer Science, Springer-Verlag, 2013. To appear.

[Hoar69] C. A. R. Hoare, "An axiomatic basis for computer programming", *Commun. ACM*, vol. 12, no. 10, pp. 576–580, October 1969.

[Hoar74] C. A. R. Hoare, "Monitors: an operating system structuring concept", *Commun. ACM*, vol. 17, no. 10, pp. 549–557, October 1974.

[Jens12] Jonas Braband Jensen and Lars Birkedal, "Fictional separation logic", in *Proceedings of the 21st European conference on Programming Languages and Systems*, ESOP'12, pp. 377–396, Berlin, Heidelberg, 2012, Springer-Verlag.

[Kass06] I. Kassios, "Dynamic frames: Support for framing, dependencies and sharing without restrictions", *FM 2006: Formal Methods*, vol. 4085, pp. 268–283, 2006.

[Kass12a] I. T. Kassios and E. Kritikos, "A Discipline for Program Verification based on Backpointers and its Use in Observational Disjointness", Technical Report 772, ETH Zurich, Department of Computer Science, 2012.

[Kass12b] Ioannis T. Kassios, Peter Müller and Malte Schwerhoff, "Comparing verification condition generation with symbolic execution: an experience report", in *Proceedings of the 4th international conference on Verified Software: theories, tools, experiments*, VSTTE'12, pp. 196–208, Berlin, Heidelberg, 2012, Springer-Verlag.

[King76] James C. King, "Symbolic execution and program testing", *Commun. ACM*, vol. 19, no. 7, pp. 385–394, July 1976.

[Lein02] K. Rustan M. Leino and Greg Nelson, "Data abstraction and information hiding", *ACM Trans. Program. Lang. Syst.*, vol. 24, no. 5, pp. 491–553, September 2002.

[Lein04] K. R. M. Leino and P. Müller, "Object Invariants in Dynamic Contexts", in M. Odersky, editor, *European Conference on Object-Oriented Programming (ECOOP)*, vol. 3086 of *Lecture Notes in Computer Science*, pp. 491–516, Springer-Verlag, 2004.

[Lein08] K. Rustan M. Leino, "This is Boogie 2", Technical report, Microsoft Research, Microsoft Research, Redmond, WA, USA, 2008. Available at `http://research.microsoft.com/~leino/papers.html`.

[Lein09a] K. Leino and Peter Müller, "A Basis for Verifying Multi-threaded Programs", in Giuseppe Castagna, editor, *Programming Languages and Systems*, vol. 5502 of *Lecture Notes in Computer Science*, pp. 378–393, Springer Berlin / Heidelberg, 2009. 10.1007/978-3-642-00590-9_27.

[Lein09b] K. Rustan Leino, Peter Müller and Jan Smans, "Foundations of Security Analysis and Design V", in Alessandro Aldini, Gilles Barthe and Roberto Gorrieri, editors, *Foundations of Security Analysis and Design V*, chapter Verification of Concurrent Programs with Chalice, pp. 195–222, Springer-Verlag, Berlin, Heidelberg, 2009.

[Lein09c] K. Rustan M. Leino and Rosemary Monahan, "Reasoning about comprehensions with first-order SMT solvers", in *Proceedings of the 2009 ACM symposium on Applied Computing*, SAC '09, pp. 615–622, New York, NY, USA, 2009, ACM.

[Lein10a] K. Rustan M. Leino, "Dafny: an automatic program verifier for functional correctness", in *Proceedings of the 16th international conference on Logic for programming, artificial intelligence, and reasoning*, LPAR'10, pp. 348–370, Berlin, Heidelberg, 2010, Springer-Verlag.

[Lein10b] K. Rustan M. Leino, Peter Müller and Jan Smans, "Deadlock-Free channels and locks", in *Proceedings of the 19th European conference on Programming Languages and Systems*, ESOP'10, pp. 407–426, Berlin, Heidelberg, 2010, Springer-Verlag.

[Mehn12] Hannes Mehnert, Filip Sieczkowski, Lars Birkedal and Peter Sestoft, "Formalized Verification of Snapshotable Trees: Separation and Sharing", in Rajeev Joshi, Peter Müller 0002 and Andreas Podelski, editors, *VSTTE*, vol. 7152 of *Lecture Notes in Computer Science*, pp. 179–195, Springer, 2012.

[Meye97] Bertrand Meyer, *Object-oriented software construction (2nd ed.)*, Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1997.

[Mull02] Peter Müller, *Modular specification and verification of object-oriented programs*, vol. 2262 of *Lecture Notes In Computer Science*, Springer-Verlag, Berlin, Heidelberg, 2002.

[Park05] Matthew Parkinson and Gavin Bierman, "Separation logic and abstraction", in *Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '05, pp. 247–258, New York, NY, USA, 2005, ACM.

[Park07] M. Parkinson, "Class invariants: The end of the road", *International Workshop on Aliasing, Confinement and Ownership*, 2007.

[Park11] Matthew J. Parkinson and Alexander J. Summers, "The relationship between separation logic and implicit dynamic frames", in *Proceedings of the 20th European conference on Programming languages and systems: part of the joint European conferences on theory and practice of software*, ESOP'11/ETAPS'11, pp. 439–458, Berlin, Heidelberg, 2011, Springer-Verlag.

[Quei82] Jean-Pierre Queille and Joseph Sifakis, "Specification and verification of concurrent systems in CESAR", in *Proceedings of the 5th Colloquium on International Symposium on Programming*, pp. 337–351, London, UK, UK, 1982, Springer-Verlag.

[Reyn02] John C. Reynolds, "Separation Logic: A Logic for Shared Mutable Data Structures", in *LICS*, pp. 55–74, 2002.

[Sha90] L. Sha, R. Rajkumar and J. P. Lehoczky, "Priority Inheritance Protocols: An Approach to Real-Time Synchronization", *IEEE Trans. Comput.*, vol. 39, no. 9, pp. 1175–1185, September 1990.

[Sman08] Jan Smans, Bart Jacobs, Frank Piessens and Wolfram Schulte, "An automatic verifier for Java-like programs based on dynamic frames", in *Proceedings of the Theory and practice of software, 11th international conference on Fundamental approaches to software engineering*, FASE'08/ETAPS'08, pp. 261–275, Berlin, Heidelberg, 2008, Springer-Verlag.

[Sman09] 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, pp. 148–172, Berlin, Heidelberg, 2009, Springer-Verlag.

[Summ09] Alexander J. Summers, Sophia Drossopoulou and Peter Müller, "The need for flexible object invariants", in *International Workshop on Aliasing, Confinement and Ownership in Object-Oriented Programming*, IWACO '09, pp. 6:1–6:9, New York, NY, USA, 2009, ACM.

[Summ10] Alexander J. Summers and Sophia Drossopoulou, "Considerate Reasoning and the Composite Design Pattern", in Gilles Barthe and Manuel V. Hermenegildo, editors, *VMCAI*, vol. 5944 of *Lecture Notes in Computer Science*, pp. 328–344, Springer, 2010.

[Z3Qu] "Z3 SMT 2.0 Guide". `http://rise4fun.com/z3/tutorial/guide` section Quantifiers.

[Zhao07] Yang Zhao, *Concurrency analysis based on fractional permissions*, Ph.D. thesis, University of Wisconsin at Milwaukee, Milwaukee, WI, USA, 2007. AAI3279111.