Improving Permission-Based Verification of Concurrent Programs with Chalice

Bachelor Thesis Report

Chair of Programming Methodology
Department of Computer Science
ETH Zurich

http://www.pm.inf.ethz.ch/

By: Stefan Heule
stheule@student.ethz.ch

Supervised by: Dr. Alexander J. Summers
Prof. Dr. Peter Müller

Date: June 27, 2011

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Contents

1 Introduction

Software correctness is a major problem in computer science and many approaches exist. One of them is static verification, where the correctness of a certain piece of software is proven under any circumstances, that is regardless of the programs input or the behaviour of other software components that use it.

Proving correctness of sequential programs is already hard, and the verification of concurrent software is considerably more difficult. But with today's clear trend towards multi-core processors, the challenge to verify concurrent software becomes more and more important.

Chalice [LM09] is both a research language and static verifier for concurrent programs. It is a collaboration of Peter Müller, Rustan Leino from Microsoft Research and Jan Smans from Katholieke Universiteit Leuven, and this Bachelor thesis is concerned with improvements to its verification methodology.

Chapter 2 gives an introduction to Chalice and Boogie, such that people who are not familiar with these tools can still follow the rest of this report. Chapter 3 then presents the new permission model developed as the core of this Bachelor thesis, while Chapter 4 talks about an extension that further enhances Chalice's expressiveness. We evaluate the new permission model in Chapter 5, mention additional work of this Bachelor thesis in Chapter 6 and conclude in Chapter 7.

2 Background

Chalice and Boogie are two verification tools that this project is related to. Therefore we give a very brief overview to them in this section. We only give information sufficient to follow the rest of the report; full details can be found in [LM09, LMS09] and [ByECD+06, Lei08] for Chalice and Boogie, respectively.

2.1 Chalice

Chalice is an object-based language which allows the developer to *specify* the intended behaviour of the code. In particular, Chalice supports contracts such as pre- and postconditions or loop invariants. These annotations are then checked statically by the verifier to ensure that they can never be violated.

Chalice translates its input program to the intermediate verification language Boogie [ByECD+06] for which verification conditions in first-order logic can be generated. These verification conditions can then be solved by an SMT solver such as Z3 [dMB08]. The verification process is depicted in Figure 1.

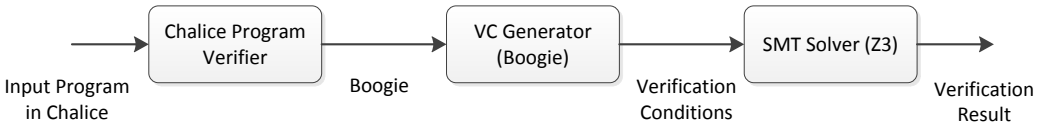

Figure 1: Verification process of Chalice.

This section describes Chalice as it existed before this project. In particular, none of the modifications that we applied as part of this project will be discussed here.

This description of some of the features of Chalice is loosely based on [LMS09, LM09].

2.1.1 Classes, Methods and Specifications

Let us look at a simple, sequential program to introduce the basic concepts of Chalice. A program in Chalice consists of one or several classes, which can contain fields and methods. For instance, we can write a class `Math` that provides a method `div` to compute the result of the division of two integers `a` and `b` and returns the result as `c` (which is not very interesting from a programming point of view, but will illustrate how Chalice works). As shown in Listing 1, we can use a precondition to indicate that the method may only be called for values of `b` other than 0. Furthermore, we can specify what method `div` computes.

We can also write a more specific method `div2` that divides an integer value by 2, and uses the more general method internally. Clearly, in this case we do not need a precondition any more, and our postcondition can be more specific. In the implementation, we can call the method `div`. At this point, we need to satisfy the precondition (which we do, as $2 \neq 0$), and we can assume the postcondition afterwards. This method is also shown in Listing 1.

2.1.2 Permissions

Permissions are the central component in the verification methodology of Chalice. They are used to determine whether a piece of code can read or write to a memory location, and these permissions are held by a particular invocation of a method, also called an activation record. We denote a full permission to the field `f` of an object `o` by **`acc(o.f)`**, which indicates exclusive access to this location.

```
class Math {
  method div(a: int, b: int) returns (c: int)
    requires b != 0
    ensures c == a / b
  {
    c := a / b
  }

  method div2(a: int) returns (c: int)
    ensures c == a / 2
  {
    call c := div(a, 2)
  }
}
```

Listing 1: A simple, sequential program in Chalice.

An activation record can also hold less than a full permission, which only allows read access. We can think of a full permission as representing 100%, in which case any percentage less than 100 indicates a read permission. Such a permission, e.g., one corresponding to 15%, is denoted by `acc(o.f,15)`.

Consider the example shown in Listing 2: The execution starts with a single activation record for method `main`, which does not hold any permission. Then, a new object `c` of type `Cell` is created, which transfers full permission to the current activation record. This allows us to write to location `c.f` on the next line. We can also call method `increment`, as we are able to fulfil its precondition, which requires us to have `acc(c.f)`.

The postcondition of `increment` then transfers the permission to `c.f` back. Note that the postcondition also guarantees us some functional properties of the method. In particular, it ensures that the value of `f` is exactly one greater than before. This is expressed using the `old` construct, where `old(e)` denotes the value of expression `e` in the pre-state of the method.

The method `square` that is called next demonstrates how read permissions can be used. Since `square` only read the value `c.f`, it is enough to have a read permission to that location. In the example we use 5%, but this amount is chosen rather arbitrarily. In fact, any positive amount would be enough for the method to verify.

We can use Chalice to prove that there are no errors in our example, and indeed the verification successfully finishes with the following output:

```
Boogie program verifier finished with 7 verified, 0 errors
```

Chalice translated our example to seven Boogie methods, three corresponding to the methods of `Cell`, and four additional methods generated internally to ensure the well-formedness of our contracts.

More permissions. Besides the already mentioned permissions that correspond to a percentage, there is another type of permission available in Chalice. It is denoted by `rd(o.f)` and we can think of it as one ε of the locations permissions for some infinitesimal ε. Similar to the percentages, we can denote multiple such epsilon permissions by `rd(o.f,n)`, denoting $\mathtt{n} \cdot \varepsilon$. Such an epsilon permission cannot be split further; it denotes the smallest possible amount of permission in the system.

The interaction with the previously mentioned percentage permissions works as follows: As an epsilon permission is an arbitrarily small yet positive amount, it is clearly smaller than any percentage, in particular less than 1%. In fact, any (finite) number of epsilon permissions is smaller than 1%. This makes it possible to call a method that requires `rd(o.f,n)` (i.e., `n` epsilon permissions) if we hold at least one percent of the permission to that location, or `m` epsilon permissions with $\mathtt{m} \geq \mathtt{n}$. In summary, the following permissions are available in Chalice

```
class Cell {
  var f: int;

  method main () {
    var f2: int;
    var c: Cell := new Cell;

    c.f := 0;

    call c.increment();
    call f2 := c.square();
  }

  method increment()
    requires acc(this.f);
    ensures acc(this.f) && this.f == old(this.f)+1;
  { this.f := this.f + 1; }

  method square() returns (f2: int)
    requires acc(this.f,5);
    ensures acc(this.f,5) && f2 == this.f*this.f;
  { f2 := this.f*this.f; }
}
```

Listing 2: Illustrative example of Chalice's use of permissions.

(all referencing the field `f` of object `o`):

- *Percentage permissions*, denoted by **acc**`(o.f,p)` with parameter `p` between 1 and 100 inclusive. The often used *full permission* **acc**`(o.f,100)` can be abbreviated by **acc**`(o.f)`.
- *Epsilon permissions*, denoted by **rd**`(o.f,n)` with positive parameter `n`. Corresponds to a permission of $\mathtt{n} \cdot \varepsilon$. A permission denoting a single *epsilon* can also be written as **rd**`(o.f)` for short.

Interpretation of permissions. It is important to understand that permissions are used for the verification only and are part of the so-called ghost state. In the final compiled program, permissions do not play any role and in fact can be removed altogether.

We have seen that only a full permission **acc**`(o.f)` allows us to write to a location, while any other positive amount of permission allows us to read the location, but not modify it. As permissions cannot be duplicated or forged, the sum over all activation records of all permissions to a particular memory location can never exceed 100%. Also, whenever more than one activation record holds permissions to the same location (e.g., using threads, as seen next), both can only have a read permission. This prevents data races.

Permission encoding. Permissions are represented as a pair (p, n) of two integers p and n. The first part, p, corresponds to the percentage of a permission, and can have any value between 0 and 100. The second part then stores the number of epsilon permissions, and can have any integer value. Intuitively, such a pair corresponds to $p\% + n \cdot \varepsilon$, and this value must be greater or equal than 0 for the permission to be valid.For instance, the permission in Chalice are encoded as follows

- **acc**`(o.f)`: $(100, 0)$
- **acc**`(o.f,p)` for a percentage `p` between 1 and 100: $(\mathtt{p}, 0)$
- **rd**`(o.f)`: $(0, 1)$

- **rd**(o.f,n) for a positive number n: $(0, n)$

Of course during verification different amounts of permission can also occur. In particular, it is possible that the number of epsilon permissions is negative. For instance, if we start with a full permission, which is encoded as $(100, 0)$, and give away one epsilon, we are left with $(100, -1)$, which corresponds to $100\% - \varepsilon$. However, for the permission to be valid we require $(p > 0) \vee (p = 0 \wedge n \geq 0)$. Only permissions that fulfil this property represent a non-negative amount.

Theoretical background of permissions. The permissions used in Chalice are base on Boyland's fractional permissions [Boy03]. Boyland proposed linear capabilities (which he called permissions) for checking programs for interference, and in particular he introduced "fractional" permissions. Any fraction of a permission can be used to read a location, while only the full permission permits write access. Chalice uses this idea, but does not allow the full complexity of rational numbers. Chalice rather uses percentages, which are integer and can be between 0 and 100.

2.1.3 Threads

Chalice allows a program to work concurrently by using multiple threads. Creating threads is possible with the well-known fork-join mechanism. To start a possibly long-running computation in another thread, we can fork a method m on object o as follows:

```
fork o.m(arg1, ..., argn)
```

This statement starts a new thread that runs method m on object o, possibly using some arguments argi. The statement is non-blocking, that is the current thread does not wait for the newly started thread to complete. The statement also transfers all permission mentioned in the precondition of m to the new thread.

From a verification point of view this statement requires us to fulfil the precondition of m, and since m might not have finished before we execute the next statement, we cannot assume the postcondition afterwards. However, we can join the other thread again later during execution. To identify the new thread, **fork** optionally returns a token, which allows us to join this particular thread. At that point, method m has finally finished, and returns its result[1], and we can assume the postcondition. In particular, any permission mentioned in the postcondition is transferred back. The following lines illustrate this:

```
fork tk := o.m(arg1, ..., argn)
/* ... */
join res1, ..., resn := tk;
```

Note that the usual method call introduced earlier can be seen as a **fork** immediately followed by the corresponding **join** statement. These two statements have the same effect as a regular method call. The only difference possible is a different performance behaviour at runtime.

2.1.4 Monitors

Similar to Java or C#, every object in Chalice can be used as a monitor to control access to data structures in possibly sophisticated ways. To allow threads to give up their permission to certain locations in order that other threads can work on that location, a monitor in Chalice can hold permission. When the monitor is acquired, the permission is transferred from the monitor to the acquiring thread, and on releasing the lock the permission goes back to the monitor. This allows different threads to compete for resources and work on these resources

[1] Note that Chalice allows a method to have any number of output parameters.

in parallel. While the fork/join mechanism introduced so far allowed to transfer permission only when threads are forked or joined, monitors are much more flexible and allow to threads to exchange permissions while they are running. This enables programs to use shared data structures such as a shared buffer.

Object life-cycle. An object in Chalice is in one of three possible states: *not-a-monitor*, *available* or *held*. The object life-cycle starts in the state *not-a-monitor*, where the object cannot be used as a monitor yet. However, the statement **share** o transitions the object o from *not-a-monitor* to the *available* state. Now, the statement **acquire** o can be used to transition o from *available* to *held* and **release** o to go back to *held*. Finally, **unshare** o can be used to transition o from *held* to *not-a-monitor*.

Monitor invariants. The permissions held by a monitor are specified by its monitor invariant. Just like other specifications, monitor invariants can contain access assertions (i.e., permissions) and pure assertions (properties about variables). The monitor invariant has to hold whenever the object is in the state *available*, but not in the other two.

Therefore, when we share an object, we have to make sure that the monitor invariant holds and that we have the required permissions. These permissions are then stored in the monitor, and the object is ready to be used as a monitor. When a thread actually acquires the monitor, it may assume the monitor invariant. When the thread releases the monitor again, it has to provide the necessary permission, and make sure that the monitor invariant holds.

Example. Consider the example shown in Listing 3. We have a class `Cell` that holds an integer. Its monitor invariant contains the full permission to that integer, and specifies that the value of `f` must be larger than 1. In the client code (method `main`) we first create a cell `c` and set `c.f` to 4. This allows us to share the object, as we can provide the necessary permission, and the monitor invariant holds (since $\texttt{f} = 4 > 1$).

```
class Application {
  method main () {
    var c: Cell := new Cell { f := 4 };
    share c;
    acquire c;
    c.f := 0; // it is ok to break the monitor invariant here
    c.f := c.f+100; // monitor invariant re-established
    release c;

    c.f := 3; // ERROR: insufficient permission to write to c.f
  }
}
class Cell {
  var f: int;
  invariant acc(this.f) && this.f > 1;
}
```

Listing 3: Monitors in Chalice can contain permission.

Note that the permission to `c.f` has been transferred to `c`, and therefore we cannot read or write that location any more, as we are left with 0%. However, we can acquire the object, which transfers the permission back to us. This allows us to update the location, even in ways that break the monitor invariant (we set `c.f` to 0). However, when we release the object again, the invariant has to hold, and we lose the permission to the field `f` again.

When we try to write to `c.f` after releasing the object, this fails as we do not hold enough permission. In fact, we do not hold any permission at all to `c.f`, as the full permission is stored in the monitor.

2.1.5 Data Abstraction

Chalice provides two kinds of abstraction in specifications; predicates can be used to abstract over pure and access assertions, while functions provide abstraction over pure assertions only.

Let us look at functions first. They allow us to specify behaviour without exposing a class' internal representation. That is, we can abstract over the values of memory locations. In Listing 4 we use a function `getCredit` to abstract over the value of the field `credit`. Functions can have preconditions, in particular to get the necessary permissions. However, functions cannot change the program state (but only read it). This is also why permissions required by a function need not be explicitly returned to the caller via a postcondition, but rather they are transferred back automatically.

```
class BankAccount {
  var credit: int;

  function getCredit()
    requires rd(this.credit);
  { this.credit }

  method withdraw(amount: int)
    requires acc(this.credit) && 0 < amount && amount < getCredit()
    ensures acc(this.credit) && getCredit() == old(getCredit()) - amount
  {
    this.credit := this.credit - amount
  }
}
```

Listing 4: Using functions to abstract over values of memory locations.

Note that the internal representation of a bank account in Listing 4 is not yet completely hidden, as the precondition still mentions the field `credit`. This can be changed by using predicates.

Predicates abstract both over permissions and the values of memory locations. For instance, if we look at the bank account example again, but also using predicates this time, we can introduce a predicate `valid`. This predicate is the abbreviation of the permission to `credit` and the fact that `credit` contains a non-negative value, as shown in Listing 5.

There are two possible views on a predicate, the abstract (or folded) and the concrete (or unfolded) view. While the abstract view is independent of the predicates definition, this definition is expanded in the concrete view. We can switch between those two views by using the statements **`fold`** and **`unfold`**. For instance, in method `withdraw` we expand the predicate `valid` to its concrete view and thus get access to `credit`. This allows us to update the location, and then we fold the predicate again. Folding the predicate requires us to have access to `credit`, and we need to show that `credit` holds a non-negative value. That is, we can think of **`fold`** `valid` being a method call with precondition **`acc(this.credit) && credit > 0`** and postcondition `valid`.

Note that also in the precondition of the function `getCredit` we now can replace the access permission by the predicate. To temporarily unfold the predicate and peek at the value of `credit`, we can use the **`unfolding`** statement.

2.1.6 Loops

Loops in Chalice are denoted by the **`while`** keyword and it is possible to specify a loop invariant. The loop invariant needs to hold immediately before the loop, and the loop body must preserve

```
class BankAccount {
  var credit: int;

  function getCredit()
    requires valid;
  { unfolding valid in this.credit }

  predicate valid { acc(this.credit) && this.credit > 0 }

  method withdraw(amount: int)
    requires valid && 0 <= amount && amount <= getCredit();
    ensures valid && getCredit() == old(getCredit()) - amount;
  {
    unfold valid
    this.credit := this.credit - amount
    fold valid
  }
}
```

Listing 5: Predicates bring even more abstraction than functions.

this invariant. An example of a program using a loop is shown in Listing 6.

```
class Application {
  method main () {
    var c: Cell := new Cell { f := 4 };
    var i: int := 2;

    while (i < c.f)
      invariant acc(c.f);
      invariant i > 0;
    {
       c.f := c.f*i;
       i := i+1;
    }
  }
}
class Cell {
  var f: int;
}
```

Listing 6: Loops and loop invariants.

2.1.7 Modelling Permission Transfer

Permissions are transferred between activation records, thread and monitors at various places in Chalice. For instance, method calls, acquire and release statements, and forks transfer permission. We model all of these with two operations called Inhale and Exhale.

Intuitively, to inhale an expression E means that we get the permission mentioned in E, and we can assume the pure assertions in E. Furthermore, we lose all information about a location, if we gained permission to a location we didn't have permission to before. This models the fact that another thread might have changed the location, and is modelled with the **havoc** command of Boogie (for a description of **havoc**, see Section 2.2.1). Exhaling an expression E, on the other hand, means to assert that we have at least the amount of permission mentioned

in E, and then removing these permissions. Furthermore, we assert the pure assertions in E.

Next, we introduce two shorthand notations to check whether we have enough permission. `CanWrite(o,f)` can be used to indicate full access to field `f` of object `o`, and `CanRead(o,f)` similarly for read access. Knowing how permissions are encoded, we can easily give the definitions for `CanRead` and `CanWrite` as follows:

`CanWrite(o,f)` $\equiv$ *let* (p, n) be the permission to `o.f` *in* $p = 100 \wedge n = 0$
`CanRead(o,f)` $\equiv$ *let* (p, n) be the permission to `o.f` *in* $p > 0 \vee n > 0$

With these definitions, we can now rather easily express the verification conditions in Chalice. When we verify a method `m`, we can assume its precondition *pre*(`m`), verify its body and then check that the postcondition *post*(`m`) holds. Using the new terminology from above we first inhale the precondition (and thus also get all the permissions mentioned there), then verify the body and exhale the postcondition in the end. This exhaling corresponds to checking that the postcondition holds, and that we have the necessary permissions (as specified by the postcondition).

In this manner, we can also give the precise verification semantics of all statements. For instance, a method call **`call`** `m()` corresponds to the following sequence

Exhale [*pre*(`m`)];
Inhale [*post*(`m`)]

Intuitively, we need to check that the precondition holds (including that the necessary permissions are available) before the call (which is done by exhaling *pre*(`m`)) and then can assume the postcondition (and gain any permissions mentioned in *post*(`m`)) afterwards.

For all other statements, similar verification conditions are necessary. The full semantics can be found in [LM09].

2.1.8 Other Features

Chalice comes with many more features than the ones presented here. For instance, it supports deadlock prevention, dynamic lock reordering and inference of certain annotations. However, these are not relevant for the rest of the report and are therefore omitted. Again, we refer the interested reader to [LM09] or [LMS09].

2.2 Boogie

Boogie is an intermediate verification language and is used by various static verifiers. It also abstracts over the interfaces of several theorem provers. It is an imperative language that allows the easy encoding of verification conditions.

In this section we describe a very small subset of the features of Boogie. The full documentation can be found in [ByECD+06], but the description here will suffice to follow the rest of this report.

2.2.1 Assumptions and Assertions: Encoding Verification Conditions

Assertions are used to prescribe properties that need to be proven. For instance, a method call `b.m()` might result in an assertion that the receiver is not **`null`**, i.e., **`assert`** `b` $\neq$ **`null;`**.

On the other hand, an **`assume`** statement is used to introduce an assumption, that is information that the prover can use. It has the effect of rendering traces where the assumption would not hold infeasible. For instance, assumptions can be used to constrain the range of values of a variable at a given program point. Boogie allows one to assign an arbitrary value to a variable using the **`havoc`** statement. An assumption can then be used to limit the range of possible

values that this variable can take. For example, we can assign an arbitrary value between 0 and 200 to the variable `i` using the statements **`havoc i; assume`** $0 \leq i \wedge i \leq 200;$.

2.2.2 Maps

Boogie supports polymorphic map types to denote updateable maps. Such a map type consists of a number of domain types, and a single range type. Furthermore it is possible to have one or more polymorphic type arguments. The domain types are written comma-separated inside square brackets followed by the range type. For instance **`[int]bool`** denotes a map from integers to boolean values. For instance, a typical use of such maps is to encode the heap as a map from object references and field names to values.

Chalice uses maps for several things, including to encode the permission mask, which is used to keep track of the amount of permission we have at any particular point in time. For instance, if we want to use the mask to keep track of the percentage of permission (let us ignore epsilon permissions here for a moment) that is currently available, we can use a map as shown in Listing 7. We first define a type **`ref`** for references and a polymorphic type `Field a` for fields

```
type ref;
type Field a;
type MaskType = <a>[ref,Field a]int;
var Mask: MaskType;
```

Listing 7: A permission mask to keep track of percentages.

of type `a`. Then, our mask is a map from references and fields to an integer. These maps can be read and modified, as is shown in Listing 8.

```
const unique field.f: Field (int);
var obj: ref;
/* .. */

// read permission to obj.f
var i: int;
i := Mask[obj, field.f];

// set permission to obj.f to 0
Mask[obj, field.f] := 0;
```

Listing 8: The permission mask can be updated.

3 The New Permission Model

In this section we present the new permission model. We first motivate why the current model needs to be changed at all, and then present our new permission model. An evaluation of this new model follows, along with a description of several steps towards the final version of the model, including the problems we encountered.

3.1 Motivation

We identified two main problems with the old permission model of Chalice. Both of these issues are addressed by our new permission model.

3.1.1 Permission Splitting

The old encoding does not allow permissions to be split arbitrarily many times (a single epsilon permission was not splittable at all). The fact that any percentage permission can be split in an arbitrary (finite) number of epsilon permissions is useful and allows the verification of many examples. However, this is not always enough, as the following example illustrates.

Consider a program that works on a large binary tree of unknown size. At every node, the program creates two additional threads that will process the child nodes. Furthermore, we assume that there is some global data that all of those threads need access to. The thread on the root has a full permission to this global data, and needs to split this permission in some way to give its child threads the possibility to perform their work. However, if we don't know the size of the tree, we have to be prepared that the child threads require a permission that can be further split arbitrarily often. However, this is exactly what Chalice cannot provide: if we give some number of epsilons to one of our children, we cannot make sure that the child thread does not (recursively) spawn more threads than it has epsilon permissions. On the other hand, giving a percentage permission does not work either, as we can split such a permission in at most 100 pieces before we are left with only 1%. At that point, we would have to split into epsilons, which again leads to the same problem.

Such an example with a binary tree is shown in Listing 9. We pass an object `data` to the method `work`, where `data` is the global data that every thread needs read access to. The problem is now to write the precondition of this method `work`, where we want to require some kind of read permission to `data.f`. But no matter what permission we put there, this does not allow us to (always) recursively call `work` on our children.

3.1.2 Permissions and Information Hiding

If we specify a method, the programmer has to decide whether the method requires read or write access to the relevant memory locations. This should already be enough, as it does not matter how much the method exactly gets in case of the read permission. Any positive amount of permission will do and allow us to access the memory location.

In contrast, the old permission model of Chalice required us to specify the precise amount for every permission, for instance an epsilon permission, or 2%. The specification of a method contains more information than would actually be needed.

To illustrate the problem, consider a method `m` that performs two subtasks, both requiring read access to a certain location `o.f`. If we execute the two subtasks sequentially (shown as `m1` in Listing 10), one epsilon of the permission to `o.f` is enough for method `m`. However, if we later decide that the two task should be executed in parallel using two threads (as shown in Listing 10, method `m2`), we cannot change the implementation without changing the contract of `m` as well, even though we still only require read access.

```
class Node {
  var l: Node; var r: Node;
  method work(data: Data)
    requires valid;
    requires /* read access to data.f */;
  {
    unfold valid;
    if (this.l != null) { fork this.l.work(data) }
    if (this.r != null) { fork this.r.work(data) }
    /* perform work on this node, using data.f */
  }
  predicate valid {
    rd(this.l) && rd(this.r) && (this.l != null ==> this.l.valid)
        && (this.r != null ==> this.r.valid)
  }
}
class Main {
  method main(tree: Node)
    requires tree != null && tree.valid;
  {
    var data: Data := new Data { data.f := 1 }
    fork tree.work(data);
  }
}
class Data { var f: int; }
```

Listing 9: Working on a binary tree with arbitrary permission splitting.

On the other hand, it is also not a good idea to just require large amounts of permission in the first place, such that a later change in the implementation (hopefully) is possible without changing the specification. If we did this, clients that only have a small amount of permission cannot call such a method any more.

When specifying a method in the old permission model of Chalice, we have to give some information on how we use the permission away through the specification. This violates information hiding and makes the process of specifying methods non-modular: In general we cannot decide how much permission we require for a method `m` without knowing the amount of permission the other methods in our system require. This additional work of choosing the precise amounts of read permissions is only imposed by the methodology.

3.2 The New Permission Model

To overcome the shortcomings of the old permission model identified in Section 3.1.2, we recall Boyland's fractional permissions [Boy03]. There, a permission corresponds to a rational number between 0 and 1, where 1 indicates exclusive access, and any other positive value corresponds to a read permission. Such fractions have the beautiful mathematical property that they can be split arbitrarily often into two positive amounts.

A major problem with fractional permissions is that the user has to pick appropriate values in the specification. This can be very tedious and for the programmer it would suffice to think more abstractly only in terms of read or write permissions. Intuitively it is not important what precise fractions are used, only the difference between full permission, read permission and no permission matters to the programmer. This commitment to specific fractional in the specification also leads to the information hiding problems mentioned before.

To enable a more abstract view, we use two permissions in our methodology: By **acc(o.f)**

```
class Worker {
  var f: int;

  method m1()
    require rd(this.f,1);
  {
    call subtask_1();
    call subtask_2();
  }

  method m2()
    require rd(this.f,1);
  {
    fork tk1 := subtask_1();
    fork tk2 := subtask_2(); // ERROR: we have no permission left to start
        subtask 2
    join tk1; join tk2;
  }

  method subtask_1/2()
    requires rd(this.f,1);
    ensures rd(this.f,1);
  { /* perform some computation involving f ... */ }
}
```

Listing 10: Information hiding problem in the old permission model.

we denote the full permission to field `f` of object `o`, and similarly we use **rd**`(o.f)` as a read permission to that location, which we call *abstract read permission*. Clearly, the full permission corresponds to the fraction 1 and we interpret **rd**`(o.f)` for some location `o.f` as an *unknown*, positive fraction. Even though in the previous model **rd**`(o.f)` was also used to denote read permission, there is no further relation. In particular, the abstract read permission is not an epsilon permission.

Note that every read permission **rd**`(o.f)` can in principle correspond to a different fraction, even if the referred memory location is the same. An important question when designing this new permission model was, in which situations these fractions should not be completely arbitrary, but rather correspond to the same fraction in order to give useful properties.

One common example is that a method requires read access to some location `o.f` and then returns this permission again via the postcondition. That is, both the pre- and postcondition mention **rd**`(o.f)`. In this case, it is often convenient that those two permissions correspond to the same fraction. For instance, if we start with full permission to `o.f` and call such a method, we would like to still be able to write to `o.f` after the call. That is, two instances of a read permission to the same location in the contracts of a method should correspond to the same amount.

However, a method might return permission to a certain location not via the same reference it got the permission. For instance, if the precondition of the method mentions **rd**`(a.f)` and the postcondition **rd**`(b.f)`, we usually cannot decide whether these two abstract read permissions should be interpreted as representing the same amount. In general, we do not know full aliasing information statically. For this reason, we adopt the following rule: *If a method specification mentions abstract read permission in the pre- or postcondition for any location, then the amounts of all these abstract read permission are identical.*

We will later see, why this restriction of having the same fraction for all abstract read permissions does not limit our model in practice.

3.2.1 Verification of Methods

To verify a method, we have to give some meaning to the abstract read permission that might be mentioned in its specification. To this end, we introduce a variable π_{method} that corresponds to the amount of any abstract read permission. By design, we do not know this value. However, we can give the following underspecification that will suffice to verify the method:

$$0 < \pi_{\text{method}} \leq 1$$

Because we do not choose a specific value for π_{method}, a successful verification proves that the method is correct for any value of π_{method} in $(0, 1]$.

This allows us to then use the same technique as in the old version of Chalice; we inhale the precondition, verify the method's body and then exhale its postcondition. If the postcondition mentions a read permission `rd(o.f)`, then we need to ensure that we have at least fraction π_{method} of the permission to `o.f`.

Let us look at an example. Listing 11 shows a simple Chalice program which we use to illustrate how the verification works with the new model. When we verify method `m`, we first fix the fraction `pi_method` using the **havoc** statement and a simple assumption. Then we inhale the precondition, verify the method body (which we don't consider yet) and then exhale the postcondition. A high-level Boogie-like encoding of those steps can be seen in Listing 12. For brevity we only show the parts relevant for the new permission model and we will do so in all the following examples as well.

```
class A {
  var f: int;
  var g: int;

  method m()
    requires rd(this.f) && acc(this.g);
    ensures rd(this.f);
  {
    this.g := this.f;
    call n();
  }

  method n()
    requires rd(this.f);
    ensures rd(this.f);
  { /* ... */}
}
```

Listing 11: Verifying methods in the new permission model.

Looking more closely at Listing 12, we can see that we use the map `Mask` as our permission mask. As permissions correspond to fractions, this mask has rational numbers as its range type. When we inhale the precondition of `m`, the permission is transferred to the current activation record and thus we increment our mask. We increment the permission to **`this.f`** by `pi_method`, and the permission to **`this.g`** by `1`, as `1` corresponds to the full permission. At then end of the method, we exhale the postcondition and therefore have to provide the necessary permission. For the location **`this.f`** this means to check that we have at least fraction `pi_method` in our mask. Note that we use the same fraction `pi_method` for both inhaling the precondition as well as exhaling the postcondition.

```
// fix fraction pi_method
  var pi_method: int;
  havoc pi_method;
  assume 0 < pi_method ≤ 1;
// inhale precondition of m
  Mask[this,f] := Mask[this,f] + pi_method;
  Mask[this,g] := Mask[this,g] + 1;
// verification of the methods body
  /* ... */
// exhale postcondition of m
  assert Mask[this,f] ≥ pi_method; // check permission to fulfil postcondition
  Mask[this,f] := Mask[this,f] - pi_method;
```

Listing 12: Boogie proof obligations related to permissions for the example shown in Listing 11.

3.2.2 Verification of Synchronous Method Calls

If the method body calls another method `n`, then the specification of `n` might also mention abstract read permissions. Since we verify method `n` for any fraction in $(0, 1]$, the caller is free to interpret the abstract read permissions of `n` with any positive value.

Therefore, if method `n` requires an abstract read permission to some location `o.f`, we only need to check that we currently hold a positive amount of permission to `o.f`. If we do, we can always find a fraction that is smaller and still positive, and we can give this fraction to the method.

We achieve this by again using an underspecified variable π_{call} that stands for the amount of permission corresponding to abstract read permissions in the specification of the called method. When we exhale the precondition (and thus go through all permissions), we first check that we have a positive amount of permission to the necessary location. Then, if we do, we just assume π_{call} to be smaller than the amount we currently hold, and subtract π_{call} from our permission mask.

As we have seen in Section 3.2.1, when we exhale the postcondition at the end of the verification of a method, we check that we precisely have some fixed amount of permission, namely π_{method}. The necessary checks for exhaling the precondition at a method call are rather different: here, it suffices to only assert that some positive amount of permission is available. To emphasize the difference, we call this version InexactExhale. Note however, that we still do the usual checks for a full permission: In this case, clearly just checking for some permission is not enough, but we rather check that our mask contains at least fraction 1.

When we inhale the postcondition after the call, we just increment the mask again for any abstract read permission by π_{call}. Again all read permissions in the specification of the method we call are interpreted by the same fraction. However, different calls possibly use different fractions, even if the same method is called multiple times.

To illustrate the semantics of InexactExhale, we look again at Listing 11 and consider the verification of the method body of `m`. The first line is an assignment, and we thus have to check that `g` is writable, and `f` is readable. The next statement is the method call to `n`, a method which requires and ensures read access to `f`. Thus, we first check that we have some permission left to `f`, and if so, constrain the fraction `pi_call` to be smaller than what we currently have. Note that we pick the fraction to be *strictly* smaller, which means that the caller gives some permission away, but also keeps some of the permission for himself. This allows the caller to preserve all the knowledge about a location `o.f` across a method call, even if the method requires an abstract read permission to that location.

If the method precondition mentions other read permissions as well, then we would just further constrain the range of `pi_call`, even if the permissions all refer to possibly different locations.

Now we can also see why the fact that we interpret all **rd**-expressions of a specification with the same fraction does not cause any problems: we just constrain that fraction to be smaller than the amounts we currently hold to all locations mentioned in abstract read permissions. Listing 13 shows the pseudo code of the verification of the method body.

```
// fix fraction pi_method and inhale precondition of m
  /* ... */
// statement: g := f;
  assert Mask[this,g] ≥ 1; // g must be writable
  assert Mask[this,f] > 0; // f must be readable
// statement: call n();
  var pi_call: int;
  havoc pi_call;
  assume 0 < pi_call;
  // inexactly exhale precondition of n
  assert Mask[this,f] > 0; // we need some permission to f
  assume pi_call < Mask[this,f];
  Mask[this,f] := Mask[this,f] - pi_call;
  // inhale postcondition of n
  Mask[this,f] := Mask[this,f] + pi_call;
// exhale postcondition of m
  /* ... */
```

Listing 13: Boogie proof obligations related to permissions for the example shown in Listing 11.

3.2.3 Verification of Asynchronous Method Calls

Asynchronous method calls are handled by the fork-join mechanism, and their verification is similar to that of synchronous method calls. However, when we fork a method, we only exhale its precondition, and only at the corresponding join we inhale the postcondition.

More precisely, when we encounter a **fork** statement, we use a variable π_{fork} to refer to the amount of any abstract read permission in the specification of the forked method. Then, we inexactly exhale the precondition of the method. These are exactly the steps we also took for the first part of the **call** statement. The second part, namely inhaling the methods postcondition, is only done at the **join** statement. Again, we would like to do the analogous steps as for regular method calls. In particular, we would like to use the same fraction π_{fork} to interpret any abstract read permission that might occur in the postcondition. However, in general this is not possible, as the **fork** and **join** statement can be arbitrarily far apart, e.g., they can appear in different methods. If they appear not in a scoped manner, but rather in different methods, we cannot use π_{fork} at the **join** statement as well.

Therefore, we use another underspecified variable π_{join} to refer to the amount of abstract read permissions in the postcondition. At this point, we can only provide the following bounds for this variable:

$$0 < \pi_{\text{join}} \leq 1$$

In cases where the **fork** and **join** statements are in fact scoped, and don't appear in different methods for instance, we would like to get the knowledge that $\pi_{\text{join}} = \pi_{\text{fork}}$. This is useful for the same reason we decided to interpret all read permissions in the specification of a method in the same way; when we start with a full permission, fork and later join a method that requires read access to this location, we would like to regain all the permission we gave away.

This missing information can be provided rather easily. When we constrain the fraction π_{fork}, we additionally store this fraction in a ghost field `fraction` of the corresponding token. Then, when we verify the **join** statement, we add the assumption that π_{join} is equal to this ghost

field. If the prover is able to relate the **fork** and **join** statements of a token, then we actually get the knowledge that $\pi_{\text{join}} = \pi_{\text{fork}}$. Otherwise (e.g., if joining happens in a different method), we just don't gain any additional (useful) information for the verification.

3.2.4 Losing Permission

We argued in the beginning of Section 3.2 that the read permissions in the pre- and postcondition of a method should match. That is, when we require a read permission to `o.f` and return a read permission to the same location, we actually have to return the same amount we got.

However, this is not always wanted as not all methods return the same amount of permission they got initially. Even if no permission is destroyed, the method might start a new thread that requires some permission and which has not yet finished. Or the method might store some permission in a monitor, transfer permission via a channel or fold it into a predicate.

In such cases we can just require a permission and then don't return anything via the postcondition, but rather give this permission away. However, this is not always what we want, as the example in Listing 14 illustrates: if we get a read permission to `o.f` and only fork another method `n` (that also requires a read permission), then we can't return the same amount of permission we got. But just returning no permission at all is not precise: in the end we still have some permission left that we can return, it is just less than what we got initially via the precondition.

To solve this issue, we introduce the special permission **rd***`(o.f)` which we call *starred read permission*. This version of a read permission also grants read access, but gives no guarantees about the associated amount. In particular the amount is not guaranteed to match with the amount any other permission.

```
class A {
  var f: int;

  method m()
    requires rd(this.f);
    ensures rd*(this.f);
  {
    fork n();
  }
  method n()
    requires rd(this.f);
    ensures rd(this.f);
  { /* ... */ }
}
```

Listing 14: The starred read permission allows the return of less permission than the amount received initially.

Listing 14 also shows how we use this new concept to solve the problem. By putting **rd***`(this.f)` in the postcondition, we make explicit that we do not give any guarantees about the returned permission. It will be some positive fraction, but it does not have to be the same amount that we got in the precondition.

Note that the starred read permission does not make much sense in the precondition of a method. It does not provide more flexibility than the standard read permission, as the fraction corresponding to **rd(this.f)** is constrained dynamically depending on the amount available at the call site anyway. Similarly, using **rd(this.f)** in the postcondition is not in all cases sensible. More precisely, **rd(this.f)** is only useful in a postcondition if a abstract read permission also appears in the precondition of the same method. However, even if the usefulness of these cases

is debatable, the semantics are precisely defined in those cases as well, which is why we do not exclude these border cases.

3.2.5 Verification of While Loops

The verification of a while loop is rather straight forward. The verification of a while loop corresponds to checking that the loop invariant holds before the loop and that the loop body preserves this invariant. More precisely, the loop invariant is exhaled first to check that it holds before the loop. Then, to check that the invariant is preserved, the invariant is inhaled before the loop body, followed by the verification of the loop body and then in the end the invariant is exhaled again.

The same verification procedure was also used by the previous version of Chalice. However, a few things are need to be considered additionally. We use a variable π_{loop} to refer to the amount of any abstract read permission of in the loop invariant. Analogous to a methods specification, all abstract read permissions are interpreted as the same fraction.

First, it is checked that the loop invariant holds before the loop by inexactly exhaling the invariant. This asserts that we have some permission to the required memory locations and it constraints π_{loop} to be smaller than the amount of permission we currently hold to these locations. The exhaling after the loop body however is still a normal exhale. This means that a loop that mentions a abstract read permission in its invariant has to ensure that it does not leak permission, but that the same amount is available after the loop body as well.

Again, we can use a starred abstract read permission to express the fact that we do not care about the exact amount of permission. This allows us for instance to verify the example shown in Listing 15. Even though in every iteration of the loop we give away some fraction of our permission to `x`, we still have a positive amount left over at the end of the loop body.

```
class A {
  var f: int;

  method m(count: int)
    requires rd(this.f);
    ensures rd*(this.f);
  {
    var i: int := 0;

    while (i < count)
      invariant rd*(this.f);
    {
      fork n();
      i := i + 1;
    }
  }
  method n()
    requires rd(this.f);
    ensures rd(this.f);
  { /* ... */ }
}
```

Listing 15: Using a starred read permission in a loop invariant.

3.2.6 Verification of Monitors

When we share an object, the monitor invariant is exhaled and we need to provide the necessary permission as specified by the monitor invariant. The same is true when we release a monitor. At the time we unshare or acquire a monitor we then get this permission back by inhaling the monitor invariant.

A natural but naïve idea is to take an analogous approach as for the asynchronous method calls: we constrain a fraction at the time we give away the permission, that is when we store permission inside the monitor via a **share** or **release** statement. Then, when we regain permission, we try to recall the fraction we fixed by looking at some ghost field in the heap. However, even though this approach seems reasonable, it is not sound. Consider two threads `T1` and `T2`, that concurrently operate on the monitor `lck`, and suppose the variable `lck` is some global variable of type `Lock`, as is shown in Listing 16. The monitor invariant of `Lock` mentions a read permission to field `x`.

We now consider one particular execution and show where the problem arises. Thread 1 first releases the monitor and, following this initial naïve idea, at this point we pick a fraction π_1. Then, some time later, the thread acquires the lock again, and we expect to get the same fraction π_1 back. After all, from the point of view of `T1` the two statements **release** and **acquire** are used in a scoped way.

On the other hand, after thread 1 has released the lock, another thread `T2` might acquire the monitor in between. At that point, this thread gets some (statically unknown) fraction π_1' from the monitor (in our particular execution this would be fraction π_1). Then, when thread 2 releases the monitor later on again, by our idea we can fix again a fraction π_2, and this fraction need not be the same as π_1. In particular, it might be the case that $\pi_2 < \pi_1$. However, then it is no longer correct for thread `T1` to assume that it got back the same fraction that it put inside the monitor earlier. This particular execution of the two threads is depicted in Listing 16.

```
class Lock {
  var f: int;
  invariant rd(this.f) && this.f > 0;
}
var lck: Lock
```

```
                                    |
  // Thread 1                       |  // Thread 2
  release lck; /* put π1 in monitor */ |
                                    |  acquire lck; /* get π1' */
  /* ... */                         |  /* ... */
                                    |  release lck; /* put π2 in monitor */
  acquire lck; /* get π1 back */    |
```

Listing 16: Execution to illustrate the unsoundness of the naïve approach of dynamically constraining the fraction at the **release** statement.

A very similar line of reasoning can be used to show that fixing the fraction at the point of the **share** statement is not sound either.

As we can see, the problem is really that at any point in time another thread might use the monitor as well. If we want to assume that we get the same permission back by a call to **acquire** than what we stored inside the monitor when releasing or sharing the object, we have to make sure that the fraction used to interpret any **rd**-expression stays the same for the complete execution of the program.

Therefore, we need to fix the fraction at latest at the point where two or more threads can access the monitor concurrently. This point in time is when the monitor is first leaked or

shared. However, between object creation and the first time the monitor is leaked or shared, the object is not used anyway. Therefore, we fix the fraction π_{monitor} that corresponds to a particular monitor already at object creation. Unfortunately, at this point we do not have any useful information about how to fix the fraction other than $0 < \pi_{\text{monitor}} \leq 1$. This is very different to a method call where we could fix the fraction to be smaller than the amount of permission we hold at the point of the call. That dynamic constraint on the fraction was very convenient, as it allows one to call the method whenever there is some permission to the necessary locations available, no matter how little. The problem with the monitors is that the point where we fix the fraction is different from where we need to show that we have at least this amount of permission available.

Consequences. The fact that we need to fix the fraction for a monitor very early has the consequence that we lose flexibility. Consider for instance the program shown in Listing 17. Method `m` simple releases the monitor `monitor` and therefore needs to provide the necessary permission. The monitor invariant contains an abstract read permission, and thus when we exhale this monitor invariant, we must prove that there is at least fraction π_{monitor} of permission available to `monitor.f`. The method `m` itself gets a abstract read permission to this location, which we interpret as π_{method}. However, at this point the only information we have is

$$0 < \pi_{\text{method}} \leq 1 \quad \wedge \quad 0 < \pi_{\text{monitor}} \leq 1$$

Clearly, this does not imply that $\pi_{\text{monitor}} \leq \pi_{\text{method}}$, which would be needed for the example to verify.

```
class Application {
  method m(monitor: Lock)
    requires rd(monitor.f) && holds(monitor);
    lockchange monitor;
  {
    // error, we cannot be sure that we have enough permission to monitor.f
    release monitor;
  }
}
class Lock {
  var f: int;
  invariant rd(this.f);
}
```

Listing 17: Monitors are less flexible than method calls.

We can use the starred read permission to partly regain the lost flexibility. If we use `rd*(o.f)` in a monitor invariant, we express the fact that we do not need the guarantees about the amount of permission. In particular, when we release (or share) a lock and need to put a read permission in this monitor, it suffices to have any positive fraction to the required location. Such an example is shown in Listing 18.

However, while we increase the flexibility of sharing and releasing monitors, we lose some of the guarantees. In particular, we cannot expect to get the same permission back if we acquired the lock `monitor` in Listing 18 again. We will show in Section 4.5 how this issue can be solved by an extension we defined for the new permission model.

One fraction for all monitors. The fraction that is stored in a monitor has to be fixed at the time where we create the monitor object. Two different monitors of course do not have to use the same fraction and so in principle it would be possible to use a different fraction for every monitor. However, we can increase the flexibility of our methodology by using the same fraction for all monitors in the system. First of all, this is sound, as the only requirement for these fractions is that per object they do not change during the lifetime of this object. This

```
class Application {
  method m(monitor: Lock)
    requires rd(monitor.f) && holds(monitor);
    lockchange monitor;
  {
    release monitor; // succeeds, we have some permission to monitor.f
  }
}
class Lock {
  var f: int;
  invariant rd*(this.f);
}
```

Listing 18: The starred read permission can be used in monitor invariants to increase flexibility.

clearly can still be guaranteed if every object in the system uses the same fraction and this fraction is globally fixed.

Furthermore, this allows us to verify more examples. For instance, this enables the verification of permission transfer between two monitors: if two different monitors store an abstract read permission to the same location, then we can acquire one monitor and release the other. If the two fractions could be different, then we could of course not prove such an example. Therefore we use a global constant π_{monitor} for all monitors where we know that

$$0 < \pi_{\text{monitor}} \wedge \pi_{\text{monitor}} <= 1$$

This uniform treatment of all monitors also removes the need to reason about aliasing; all abstract read permissions correspond to the same amount in a monitor, regardless of the monitor used.

3.2.7 Verification of Predicates

The situation for predicates is similar as for the monitors. We need to ensure that the fraction associated with a predicate is the same during the complete execution. Otherwise it would be possible to unfold a predicate (thereby getting some fraction of permission) and then later, possibly in a different method, fold the predicate again using another amount of permission. Therefore we use the same approach and fix a fraction $\pi_{\text{predicate}}$ for every predicate for the complete execution of the program. This introduces the same flexibility problems as with monitors and of course we can use the same solution: It is possible to use a starred read permission in a predicate to express the fact that the amount stored in the predicate is not important. This then also allows us to fold such a predicate, no matter how much permission we currently have, as long as it is some positive amount.

For the abstract read permissions, we use the same fraction for all predicates to allow unfolding one predicate and folding another seamlessly if the two contain the same location. Even more, we use the same fraction for both the predicates and the monitors, for the same reason. This allows us to transfer permission from a monitor to a predicate and vice versa without problems. Therefore we have $\pi_{\text{predicate}} = \pi_{\text{monitor}}$.

Permission scaling. In Chalice it is possible to obtain a read permission to a predicate, which effectively means scaling the permissions of the predicate's definition by the read permissions. For instance, consider a predicate `valid` with the following definition.

```
valid  ≡  acc(this.f) && this.f > 0
```

With this definitions, it is possible to write expressions like `acc(valid,10)` or `rd(valid,2)`. Intuitively, this expresses that only part of the predicates is held, e.g., 10% as in the first case,

or two epsilons as in the second case (using the syntax and semantics of the previous version of Chalice, with percentage and epsilon permissions). When such a (partial) predicate is unfolded, the definition of the predicate is multiplied by the specified permission. For instance, if an epsilon of a predicate that contains full access to some memory location is unfolded, we don't get a full permission, but rather one epsilon. A normal predicate like `valid` can now be seen as an abbreviation for the full access to this predicate, i.e., **`acc(valid)`**.

In the previous version of Chalice, it was essentially only possible to take a non-full part of a predicate if the definition of this predicate only contained full permissions[2]. In this case the permission multiplication is trivial, as at least one operand is the full permission, which means that the result is just the other operand. However, theoretically it would be possible to support more combinations. For instance, taking 10% of a predicate that stores 50% of the permission to some location is also well-defined. Such examples were not supported, however.

The new permission model uses fractions as permissions, and therefore we can easily support the multiplication of arbitrary permissions. In particular it is also possible to support the multiplication of two read permissions, something which was not possible before. To scale a permission by another permission in the new model, we can just multiply the two corresponding fractions. However, even if it is conceptually simple to multiply two fractions, we encountered some performance problems in this case. This will be explained in more detail in Section 5.3.2.

3.2.8 Verification of Channels

In Chalice, channels can be used to transfer messages and these messages can also contain permissions. Just like with monitors and predicates, we have the same issue of fixing the fraction with which we interpret any abstract read permission of a channel: if we want to assume that the permission we receive is the same as the permission we give away when sending a message, we have to fix the fraction for the complete lifetime of the channel. Therefore, we use the same ideas as for monitors or predicates, and we thus don't describe them here again.

3.2.9 Encoding Fractions as Integers

The use of fractions as permissions is very nice from a conceptual standpoint. However, Boogie currently does not support rational numbers, even though many SMT solvers (in particular, Z3) are capable of handling fractions. For this reason, we use an encoding that only relies on integers, without sacrificing any of the useful properties of fractions.

We represent all fractions π in our system using an integer k, such that

$$\pi = \frac{k}{D}$$

where D is a suitably large constant. This constant D is used to represent all permissions that we are interested in during the verification of a particular example. Similar to various previous variables such as π_{method}, we do not explicitly define the value of D, but rather give the only the following underspecification: $D > 0$. Such a constant D can be found to represent any (finite) number of permissions, and this is already enough for our purposes.

Note that D is really the same for all permissions, that is D is fixed for the verification of the whole program. Therefore, a permission can be represented solely by the integer k, where $k = D$ indicates full access (since $D/D = 1$, that is the full permission), and any other k with $0 < k < D$ corresponds to a permission that only grants read access. $k = 0$ indicates the absence of any access right.

Permission multiplication with integers. If we have a fractional permission a and scale it by another fraction b, this corresponds to the multiplication of a with b. In our encoding, a is

[2] Actually, it was also possible to have other combinations. However, those cases were not sound.

represented by an integer $\lceil a \rceil$ such that $a = \lceil a \rceil / D$, and similarly for b (we use $\lceil x \rceil$ to denote the integer encoding of a fraction x). In our encoding, multiplication can therefore be defined as

$$a \cdot b \quad \cong \quad \frac{\lceil a \rceil}{D} \cdot \frac{\lceil b \rceil}{D} = \frac{\lceil a \rceil \cdot \lceil b \rceil}{D^2} = \frac{\lceil c \rceil}{D} \quad \cong \quad c$$

Without using division (often, the support for division is limited in SMT solver), we cannot express `c` in terms of `a` and `b`, but we can use **havoc** and an assumption in Boogie to express the desired property:

havoc $\lceil c \rceil$ **;** **assume** $\lceil c \rceil * D == \lceil a \rceil * \lceil b \rceil$

3.2.10 Epsilon Permission

Even though the epsilon permissions are less flexible and can violate information hiding (as shown in Section 3.1.2), counting permissions such as the epsilons can still be useful. The problem with the fractional permissions of our model is that uniform splitting between a number of entities is not possible. This can make the book-keeping rather difficult.

For instance, consider a method `m` that requires and ensures a read permission. If we fork multiple threads to execute `m` and join them later on to regain all permissions, we cannot easily express a suitable loop invariant that allows us to regain a full permission in the end. Such an example is shown in Listing 19 for the case when we use epsilon permissions. We fork `n` threads in method `work` in a while loop, and join all threads in a second loop afterwards.

The problem of this example is that we have to provide a loop invariant for the two main loops. Intuitively this invariant should (among other things) say, that we have already forked i threads, and therefore given away the corresponding amount of permission (or, for the second loop, that i threads have been joined, and that we regained the corresponding permission).

If we use the fractional permissions as we have introduced them so far, this example cannot be verified. In our model, every call to method `m` gives rise to a new corresponding fraction π_{fork} which can be different every time. Therefore, the loop invariant would essential require a summation, and we would need to have a way of referring to the amount that has been given away to a particular thread.

On the other hand, if we use counting permissions such as shown in Listing 19[3], the problem becomes simpler. Method `m` requires one epsilon permission now, which means that the permission given away at the **fork** statement is the same, regardless of which thread is forked. Therefore, the loop invariant can be specified rather easily. Note however, that this particular example can only be encoded because we use a helper method `work` and do not have the loop directly in method `main`. The reason is, that in the main loop we would have to express in the first loop invariant, that we have "100% minus i epsilon permissions". An extension to the permission model presented in Chapter 4 will also allow this.

Summarizing, the example shown in Listing 19 can be encoded in a relatively simple way using counting permissions, and currently cannot be expressed with fractional permissions. For that, we would need summation over permission amounts, and a way to refer to the permission that has been given away in a fork. The latter will be provided by an extension to the permission model presented in Chapter 4, but not the former. However, even if summation is available, such an encoding would be significantly more difficult and would likely be more difficult to prove for an automated theorem prover.

Various such examples exist that can be written more easily when using counting permissions rather than fractional permissions. Therefore, it would hurt the expressiveness and simplicity of the verification methodology, if the epsilon permissions would be removed altogether.

[3] Counting permissions have not yet been introduced to the new model, but Listing 19 can be seen as a program from the previous version of Chalice. We will show in Section 3.2.11 how counting permissions can be reintroduced to the new model as well, in which case Listing 19 works with both the new and previous version of Chalice.

3.2.11 Reintroducing Counting Permissions

For the reason shown in Section 3.2.10, we reintroduce epsilon permissions essentially without modification. Syntactically, we cannot use **rd(o.f)** any more to represent one epsilon, since this expression already has a different meaning in the new model. However, **rd(o.f,n)** can still be used to refer to `n` epsilons and thus, **rd(o.f,1)** stands for a single epsilon.

As in the old version of Chalice, a permission is again encoded as a pair of integers (k, n), where the first component corresponds to the fractional part (that is the integer `k` we use to encode the fraction as `k/D`) and the second component counts the number of epsilons. The first component has integer values in the range $[0, \mathtt{D}]$, while the second component can take arbitrary integer values. The definitions of `CanRead` and `CanWrite` are very similar to their old versions:

`CanWrite(o,f)` $\equiv$ *let* (p, n) be the permission to `o.f` *in* $p = \mathtt{D} \wedge n = 0$
`CanRead(o,f)` $\equiv$ *let* (p, n) be the permission to `o.f` *in* $p > 0 \vee n > 0$

Other than the change of valid values for the fractional part of a permission from $[0, 100]$ to $[0, \mathtt{D}]$ and the resulting change in the definition of `CanWrite`, the encoding, compared with the old version of Chalice is completely unchanged. In particular, we can use the same mechanism for the interaction of epsilons and fractional permissions: any positive fraction is larger than any amount of epsilon permission. Therefore, we can split of any finite number of epsilons off any fractional permission.

3.2.12 Backwards Compatibility: Percentage Permissions

In the previous implementation of Chalice, percentage permissions were supported. They are denoted by **acc(o.f,p)** for a percentage `p` between 1 and 100. To allow the verification of programs written previously to this project, we reintroduce these permissions in the new permission model as follows.

A percentage permission of $p\%$ corresponds to a fraction $\frac{p}{100}$. Therefore, in our encoding, we can easily support such percentage permissions. They correspond to the integer $\frac{p}{100} \cdot \mathtt{D}$, as

$$\frac{\frac{p}{100} \cdot \mathtt{D}}{\mathtt{D}} = \frac{p}{100}$$

To guarantee that $\frac{p}{100} \cdot \mathtt{D}$ indeed is an integer and not a fraction, `D` must be divisible by 100. One simple way to ensure this property is to replace the constant `D` by $100 \cdot \mathtt{d}$ for another constant `d`. Now p percent can be simply encodes as $p \cdot \mathtt{d}$. Another nice consequence of this is that we do not need a division, something which is often not very well supported by many automatic theorem provers.

To improve the integration with the new abstract read permission, we slightly change the way we constrain the fraction π_{method}. The bounds on π_{method} are chosen as follows

$$0 < \pi_{\text{method}} \leq d$$

This means that the permission with which we interpret a abstract read permission is smaller than one percent. This allows us to give away a abstract read permission if we have one percent of permission to that location. This is not particularly important, but it allows us to verify a method that requires one percent of permission and ensures a abstract read permission. We can even go one step further and make π_{method} even smaller. This will also allow us to return multiple abstract read permissions. We use the following bounds, where the constant 1000 is chosen rather arbitrarily:

$$0 < 1000 \cdot \pi_{\text{method}} \leq d$$

3.3 Steps Towards the Final Permission Model

In this section we describe several intermediate stages of the project, most of which are limited in some way. The purpose of this is first of all to document our work. But also, and maybe more importantly, these intermediate stages and the conclusions we drew from them are helpful in understanding how we arrived at the final version and why earlier attempts did not work.

3.3.1 Initial Idea

The initial idea for this project is due to Peter Müller and Rustan Leino, and was more simplistic. The permission model was designed from scratch, completely removing the notion of counting permissions. Rather, **acc(o.f)** was used to denote a full permission, and **rd(o.f)**, similar to the model presented in this report, denotes an abstract read permission. However, the semantics of **rd(o.f)** were different from our abstract read permissions. For the synchronous or asynchronous method calls, there is no difference, but for monitors, the initial idea for the abstract read permissions was that the amount corresponding to the permission can be different every time the monitor is used. Essentially, this is the behaviour that our model supports using **rd***(o.f). This approach is less precise and, as we have seen in 3.2.6, cannot guarantee that a permission that is taken out of a monitor corresponds to the same amount as the permission that has been put into the monitor. We have also seen that this guarantee is useful, for instance to allow recombining permissions to obtain a full permission again eventually.

Furthermore, the initial idea did not include permission expression and it was not yet clear, how predicates or channels would be handled.

3.3.2 Combining Fractions and Epsilons

Early on we were already looking for a way to support the epsilon permissions in the new model. As we have argued in Section 3.2.10, they can be used to verify some examples more easily. Also, existing programs often use epsilon permissions, which means that removing the epsilon permissions from the model would damage backward compatibility.

An epsilon permission can be seen as some infinitesimal small amount of permission to a certain location. In particular, it is not possible to split such a permission into two or more parts. We had already decided to represent all fractions f in our system in the following way:

$$f = \frac{k}{D}$$

for some appropriate constant D. Therefore, an intuitive idea is to define an epsilon permission as $1/D$. This is in some sense the smallest amount of permission we can represent in our encoding, and it certainly cannot be split without changing the representation.

However, Peter Müller soon pointed out that this idea is highly problematic at best. The problem is that we now cannot easily determine whether we have some fraction of permission, or only a bounded number of epsilons. In the latter case we are only allowed to split such a permission into a finite number of pieces (if at all). More precisely, the problem manifests itself as follows: When we call a method that requires an abstract read permission to some location, we check that we have some positive amount, and then we fix π_{call} to be smaller what we currently have. Listing 20 shows a Boogie encoding of this.

The second assumption in this listing is not justified any more. Previously we knew that `Mask[o,f]` corresponds to a fraction if it contains a positive value, even if this fraction was encoded as an integer. In contrast it is now possible that `Mask[o,f]` has the value 1, and really represents an amount of permission for which we cannot find a smaller, but still positive amount.

The solution to this problem is to separate the two concepts of fractions and epsilons and keep track of them separately. This can actually be done in a very similar way as in the previous

version of Chalice by representing permissions as a pair of integers: one for the fractional part, and another for the number of epsilons.

3.3.3 Splitting Permissions and the Meaning of `rd*(o.f)`

We explained in Section 3.2.4 that not all methods return the same amount of permission that they got via their precondition. One idea that we explored was to introduce syntax to precisely specify how much permission is returned. For instance a specification could say that the method returns half of the permission it got via the precondition, or one third, etc. It seemed that this would increase the expressiveness of our methodology.

However, the possibility of having such specific relations between various amounts of permission goes against the intuition behind the new model of providing a more abstract way of reasoning about permission, without the need to pick concrete values. Furthermore it is not clear why a method should return half the permission it required. Why not a third, or 99 percent of the initial amount?

Nevertheless, there are cases where we want to express more precisely how much permission a method returns. For instance, if a method returns part of its permission via the postcondition, and passes the rest on to another thread (whose token is returned by the method), then the caller might want to recombine the permission by joining the new thread. This is not possible with **rd*** predicates. However, Chapter 4 will present a solution for this problem.

```
class C {

  var f: int;

  method main(n: int)
    requires n > 0 && acc(this.f)
    ensures acc(this.f)
  {
    // fork all threads, and join them afterwards
    call work(n);

    this.f := 100; // we want a full permission in the end
  }

  method work(n: int)
    requires rd(this.f,n)
    ensures rd(this.f,n)
  {
    var tks:seq<token<C.m>> := nil<token<C.m>>;

    // first loop; fork all threads
    var i := 0;
    while (i < n)
      invariant i <= n && |tks| == i;
      invariant i < n ==> rd(this.f,n-i);
      invariant acc(tks[*].joinable);
      invariant forall k in [0..|tks|] :: tks[k] != null && tks[k].joinable;
      invariant forall k in [0..|tks|] :: eval(tks[k].fork this.m(), true);
      invariant forall k,j in [0..|tks|] :: k < j ==> tks[k] != tks[j];
    {
      fork tk := m();
      tks := tks ++ [tk];
      i := i+1;
    }

    // second loop; join all threads
    i := n;
    while (i > 0)
      invariant i >= 0 && |tks| == i;
      invariant i < n ==> rd(this.f,n-i);
      invariant acc(tks[*].joinable);
      invariant forall k in [0..|tks|] :: tks[k] != null && tks[k].joinable;
      invariant forall k in [0..|tks|] :: eval(tks[k].fork this.m(), true);
      invariant forall k,j in [0..|tks|] :: k < j ==> tks[k] != tks[j];
    {
      var tk: token<C.m>;
      tk := tks[i-1];
      join tk;
      i := i-1;
      tks := tks[0..i];
    }
  }

  method m()
    requires rd(this.f,1);
    ensures rd(this.f,1);
  { /* do some computation */ }

}
```

Listing 19: Counting permissions can be useful to uniformly split permissions.

```
var pi_call: int;
havoc pi_call;
assume pi_call > 0;

// check that rd(o.f) holds
assert Mask[o,f] > 0;
assume pi_call < Mask[o,f]; // might be problematic
Mask[o,f] := Mask[o,f] - pi_call;
```

Listing 20: If the permission mask can store 1 to indicate one epsilon, the assumption that there is always a smaller π_{call} is not justified any more.

4 Extension

In previous chapters we have already identified various examples that cannot be supported by the new permission model so far. A partial solution to many of them were **rd*** expressions. However, starred read permissions always result in the loss of preciseness, which is crucial if the permission should be recombined later on. In this chapter we describe an extension to Chalice, and to its permission model in particular, that overcomes these issues. We first motivate again what is missing, and then present the extension itself.

4.1 Motivation

The new permission model changes the semantics of **rd**(o.f) to a fractional permission of an unspecified, but positive amount. There is currently no way to refer to the amount of such a permission. For instance, consider an asynchronous method call. It is not possible to mention the amount of permission that has been given away, or that will be regained when joining the thread again (at least, if an abstract read permission is used in the specification). If the token is passed across method boundaries, this information is therefore completely lost, and one cannot hope to regain a full permission.

Another problem is that it is not possible to express differences between permissions in general. For example, we can again consider an asynchronous call to a method that requires an abstract read permission. If we start with a full permission, we cannot express the amount of permission that is left to the caller after the **fork** statement, as we cannot express the subtraction.

Listing 21 shows a simple example that forks a thread to do a long-running computation, and than joins it again later. We need to be able to express the exact amount of permission that is missing from 100% in order to get back a full permission in the end. It is also interesting to see that using counting permissions (i.e., using **rd**(o.f,1) instead of **rd**(o.f)) would not help here, even though we now can precisely express the missing permission. We still could not write the precondition of method `finish`, since we cannot express "100% minus one epsilon".

The only thing that would have worked, is to manually split the permission into percentages, e.g., giving away 10% and keeping 90%. If we use precise percentages, we can actually already do some permission arithmetic. However, this approach would use fixed amounts of permission, which is problematic as we have argued in Section 3.1.2. Furthermore, manual splitting of percentages is severely limited, as only integers between 1 and 100 are allowed. For instance, this means that such a permission cannot be split into more than 100 pieces.

4.2 New Permission Type

To overcome the current limitations, we generalize permission predicates to allow arithmetic expressions to specify the corresponding amount. The syntax of this generalization is as follows

acc(o.f, perm_expr)

This expression stands for a permission to the field `f` of object `o`, and represents the amount specified by `perm_expr`, a *permission expression*.

4.2.1 Permission Expressions

A permission expression allows one to perform arithmetic operations on permission amounts, as well as referring to certain abstract permissions (e.g., the one used in an asynchronous method call). Furthermore, the already available amounts (such as a certain number of epsilons, or a full permission) can be used. Note that permission expressions represent a permission *amount* rather than a permission. In particular this means that permission expressions do not mention

```
class C {
  var f: int;

  method main()
    requires acc(this.f);
    ensures acc(this.f);
  {
    // start long-running processing
    fork tk := processing();

    /* do some computation itself */

    // finish
    call finish(tk);
  }

  method finish(tk: token<C.processing>)
    requires /* access to this.f of "100% - one read permission" (cannot be
        expressed yet)*/;
    requires acc(tk.joinable) && tk.joinable && tk != null;
    ensures acc(this.f);
  {
    var res: int;
    join res := tk;

    // final write to this.f (requires full permission)
    this.f := res - 1;
  }

  method processing() returns (res: int)
    requires rd(this.f);
    ensures rd(this.f);
  {
    /* do some computation */
  }

}
```

Listing 21: Simple example that cannot be verified without the extension.

a memory location, and they can only be used as part of a permission, i.e., as `perm_expr` in `acc(o.f, perm_expr)`.

Permission expressions are inductively defined as follows:

- `rd(tk)` - represents the amount of an abstract read permission mentioned in the specification of a method that is called asynchronously with token `tk`. This expression can be used for any `rd(o.f)`, regardless of `o` and `f`: As we have seen in Section 3.2, all abstract read permissions in the specification of a method represent the same amount.
- `rd(monitor)` - amount of an abstract read permission mentioned in the monitor invariant of `monitor`. This is similar to `rd(tk)`, but for monitors instead of tokens.
- `rd(ch)` - amount of an abstract read permission mentioned in the `where` clause of channel `ch`.
- `rd(p)` - amount of an abstract read permission mentioned in the definition of a predicate `p`.

- **rd** - amount of an abstract read permission of the current context. That is, **rd** corresponds to the amount of an abstract read permission of the monitor invariant, of the predicate, of the `where` clause of a channel, of the loop invariant, or of the specification of a method (depending on where it occurs).
- **rd**(n), e.g., **rd**(1) - amount corresponding to a counting permission of `n` epsilon.
- `n`, e.g., `42` - percentage of a full permission. `100` is the amount of a full permission.
- (perm_expr$_1$ $\pm$ perm_expr$_2$) - addition and subtraction of two permission expressions.
- (-perm_expr) - additive inverse of a permission expression.
- (n*perm_expr) or (perm_expr*n) - multiplication of an integer `n` with a permission expression.

Note that parentheses can also be dropped and that the normal rules of precedence apply in this case.

4.2.2 Peculiar Corner Cases

In Section 3.2.6, 3.2.7, and 3.2.8 we have explained why it is useful to use the same amount for all monitors, predicates and channels. One interesting and somewhat peculiar consequence of this equivalence is, that the amount described by **rd**(m) for a monitor `m`, **rd**(ch) for a channel `ch` and **rd**(p) for a predicate `p` is the same in all cases. While this might seem surprising, it does not cause any problems in our experience.

4.3 Encoding of Permission Expressions

As described in Section 3.2.9, a permission in Chalice corresponds to a pair (f, n) of two integers. f stands for the fractional part (encoded as an integer), and n represents the number of epsilons.

A permission that uses a permission expression to specify its amount shares the same representation as the existing permissions. In fact, the operations necessary to evaluate these expressions are not new either; permission addition and subtraction have been used already before when inhaling and exhaling permissions. The same also holds for expressions like **rd** or **rd**(tk), which have been used internally before as well. The extension merely gives the user the ability to use these constructs explicitly in specifications.

For this reason, the evaluation is straight forward, and is not explained in further detail here.

4.4 Valid Permissions

Because permission expressions allow to subtract permission amounts, and since it is possible to take the additive inverse of amounts, not all permissions are valid any more. For instance, it is not guaranteed that **rd**(tk)-**rd** denotes a positive amount of permission, and -**rd** is even negative in any case. For this reason, we demand that only positive permission amounts are used. More precisely, whenever a permission is exhaled, an additional proof obligation is generated to ensure that the permission corresponds to a *positive* amount. This is even the case for regular permissions (i.e., permissions that do not use a permission expression to specify its amount), such as percentage or epsilon permissions. This allows us then to assume that any permission we inhale denotes a positive amount. The precise details are described in Section 4.4.1.

This is different from what Chalice did before, even if we only consider the standard permissions. Previously, Chalice required that any permission mentioned in a specification denotes a

positive amount; not only when the permission is exhaled, but already the specification itself was required to ensure this property. For instance, it was not allowed to have a precondition `rd(o.f,n)` where `n` is the integer argument to the method. The definedness check rejected this specification, because it is not possible to prove that `n` is non-negative (0 was also allowed before). Instead, `n>=0 && rd(o.f,n)` could be used as precondition.

Such an explicit condition in the contract is not possible with the permission expressions. For example, it is not possible to include `rd-rd(tk)>0` in a specification, because `rd` and `rd(tk)` are permission amounts and only allowed inside a `acc` predicate. However, this would be needed to show that `acc(o.f,rd-rd(tk))` is a valid permission, which can be the case. For this reason, there is the *implicit* condition with any permission, that its amount has to be positive. We also require that it is *positive* (rather than just non-negative), because only a positive amount enables to read a location safely.

There is a further reason why such an implicit is preferable: Previously, Chalice required to explicitly write down the constraint on `n` in an expression such as `acc(o.f,n)`. This could easily be done as

```
n >= 0 && n <= 100 && acc(o.f,n)
```

However, if we add another permission to the same location, things are not so clear any more. The following expression can be used as precondition of a method:

```
n >= 0 && n <= 100 && acc(o.f,n) && m >= 0 && m <= 100 && acc(o.f,m)
```

Regardless of the explicit constraints on `n` and `m`, such a precondition hides a further, implicit constraint `n+m` $\leq$ `100`. That is, we can only call the method from contexts where this condition is satisfied. This condition is necessary and sensible, as no thread can hold more than 100 percent of permission to any memory location. However, it is unclear why `n >= 0 && n <= 100` (and the same condition for `m`) need to be written down explicitly, while `n+m` $\leq$ `100` does not.

Our approach removing these explicit constraint altogether does not suffer from such an inconsistency. Furthermore it makes specifications less verbose, as can be seen by the following example: Consider a method `start` that take an integer argument `n` and requires `n` epsilon permissions. Using our approach, the precondition `rd(o.f,n)` suffices, while previously it was necessary to use `n > 0 && rd(o.f,n)`.

4.4.1 Inhaling and Exhaling

As mentioned above, besides the obvious type-checking, there are verification conditions that need to be proven when working with permission expressions. In particular, the following steps are necessary when inhaling or exhaling such a permission:

- Exhaling
  - We have to ensure that permission expression actually stands for a valid permission. If we evaluate the permission expression to $(\mathtt{f}, \mathtt{n})$, we have to assert the following:

    ```
    assert f > 0 || (f == 0 && n > 0)
    ```

  - Update the permission mask of the particular memory location by subtracting $(\mathtt{f}, \mathtt{n})$.
- Inhaling
  - When inhaling, we can assume that the permission has a valid form, that is for $(\mathtt{f}, \mathtt{n})$

    ```
    assume f > 0 || (f == 0 && n > 0)
    ```

    This assumption is sound, because whenever we inhale such a permission, it has been exhaled earlier in the execution somewhere.

– Again, update the permission mask accordingly by adding (`f`, `n`).

Even though these verification conditions are required in general, for many cases it is possible to optimize the expressions slightly. For many permissions expressions such as **`rd-rd(tk)`** or **`rd(10)`** we can determine statically whether they only use the fractional part (i.e., $n = 0$ in the encoding (f, n), like in the former example), only consist of epsilons (i.e., $f = 0$ as in the latter example), or possibly use both parts of the encoding. In such cases, we can simply omit the redundant parts of the assertion or assumption. For instance, when exhaling **`rd-rd(tk)`**, it suffices to assert that $f > 0$, as n will be 0 for sure.

However, this is not possible in general, and our implementation only takes a very simple approximation. For instance, the permission expression **`rd+(rd(1)-rd(1))`** will not be considered to only consist of a fractional part (even though it clearly does), and the general assertion or assumption will be used. Note however, that our approximation is *conservative*, and when in doubt, always uses the general form. This is a very simple optimization that for a large number of practical example simplifies the generated output slightly.

4.4.2 Inexact or Exact Exhaling

As explained in Section 3.2.2, there are two possible version of the exhale operation; namely Exhale and InexactExhale. Recall that for an abstract read permission, InexactExhale only checks whether there is some permission available, and then constrains the corresponding variable. We call this behaviour *inexact checking*. For the full permission however, the behaviour of Exhale and InexactExhale is identical, and both check that at least fraction 1 is available (called *exact checking*).

To decide whether we have to check that at least the specified amount is available, or whether it suffices to check that some permission is available, the following recursive procedure is used.

- **`rd(tk)`** - exact checking.
- **`rd(monitor)`** - exact checking.
- **`rd(ch)`** - exact checking.
- **`rd(p)`** - exact checking.
- **`rd`** - if it occurs in a method specification, inexact checking is used, otherwise exact checking.
- **`rd(n)`** - exact checking.
- `n` - exact checking.
- `(perm_expr`$_1$ $\pm$ `perm_expr`$_2$`)` - if both `perm_expr`$_1$ and `perm_expr`$_2$ use inexact checking, then this permission expression uses inexact checking as well. Otherwise, we use exact checking.
- `(-perm_expr)` - the same behaviour as `perm_expr`.
- `(n*perm_expr)` or `(perm_expr*n)` - the same behaviour as `perm_expr`.

4.5 Previous Examples Revisited

In Section 4.1 we presented an example that requires us to express differences between permissions. Using the newly introduced permission expressions, this is easily possible, and an encoding of the example shown in Listing 21 is given in Listing 22.

```
class C {
  var f: int;

  method main()
    requires acc(this.f);
    ensures acc(this.f);
  {
    // start long-running processing
    fork tk := processing();

    /* do some computation itself */

    // finish
    call finish(tk);
  }

  method finish(tk: token<C.processing>)
    requires acc(this.f,100-rd(tk));
    requires acc(tk.joinable) && tk.joinable && tk != null;
    ensures acc(this.f);
  {
    var res: int;
    join res := tk;

    // final write to this.f (requires full permission)
    this.f := res - 1;
  }

  method processing() returns (res: int)
    requires rd(this.f);
    ensures rd(this.f);
  {
    /* do some computation */
  }

}
```

Listing 22: Simple example that cannot be verified without the extension.

In Section 3.2.6 we have seen that monitors are not very flexible if the corresponding monitor invariant mentions an abstract read permission. Using the starred read permission partly solve the issue, but at the expense of precision. That is, we cannot recombine the permission to a full permission again in such cases.

The extension allows us to precisely express the involved permission amounts. To illustrate this, we again look at Listing 17. As Listing 23 shows, we can use `rd(monitor)` to refer to the amount of permission stored in the monitor.

```
class Application {
  method m(monitor: Lock)
    requires acc(monitor.f,rd(monitor)) && holds(monitor);
    lockchange monitor;
  {
    release monitor;
  }
}
class Lock {
  var f: int;
  invariant rd(this.f);
}
```

Listing 23: Using the extension to precisely specify Listing 17.

5 Evaluation of the New Permission Model

In this chapter we critically evaluate the new permission model. In particular we consider backwards compatibility, simplicity, expressiveness and how well the new permission model performs in terms of execution time.

5.1 Backwards Compatibility

The encoding used in the new permission model is rather similar to the encoding of the previous model. Furthermore, the basic idea of having two different kinds of permissions (a fraction or percentage, and epsilons) is still present in the new model. This allowed us to retain very high degree of backward compatibility.

The full permission **acc**`(o.f)`, percentage permissions such as **acc**`(o.f,p)`, and epsilon permissions such as **rd**`(o.f,n)` are still fully supported, without changing their syntax or semantics. The short form of a single epsilon, **rd**`(o.f)`, now has a different interpretation. Also, the permission **rd**`(o.f,*)` is no longer supported. Previously, **rd**`(o.f,*)` informally had the meaning of "any number of epsilons", but has not been fully supported by Chalice. The new starred read permission, **rd***`(o.f)`, does not conflict with any existing syntax.

Even though the semantics of **rd**`(o.f)` have been changed, the new and old interpretation are still similar in practise. When we worked with Chalice programs written for the old permission model (such as the examples from the Chalice test suite), we found that in most cases the program and its contracts are still valid and can be verified without any changes.

In fact, the complete test suite of Chalice programs can be verified with only minor modifications to the contracts. The only cases where (trivial) modifications were necessary are the following:

- One test case checks that a single epsilon permission cannot be given away multiple times to other threads. More precisely, this test consists of a method `m` that requires **rd**`(o.f)` and tries to (recursively) fork two copies of `m`. Clearly, this should not be possible in the old model, where **rd**`(o.f)` corresponds to one epsilon. However, in the new model, this is supported.
- When abstract read permissions are used in monitor invariants, these permissions correspond to a fixed fraction. This requires the user in some cases to either use the mechanisms introduced in the extension (see Chapter 4), or replace the permission by **rd**`(o.f,*)` to indicate that the exact amount is not important.

Trivially, all old programs can be updated by simply replacing any occurrences of **rd**`(o.f)` (for any object `o` and field `f`) by **rd**`(o.f,1)`. This will work regardless of the program at hand, as in the old model the two programs before and after this transformation were equivalent, and in the new model no construct is used that has different semantics. Furthermore, such a transformation could even be done fully automatically by a simple program.

5.2 Expressiveness and Simplicity

We support essentially all of the features of Chalice that were present in the old version and therefore we can verify at least the examples that could be verified before. The new permission model and in particular the fact we use fractions as permission furthermore allows us to verify examples that could not be verified before.

5.2.1 Permission Splitting

The use of fractional permissions allow us to split permissions arbitrarily often. For instance let us look at the example from Section 3.1.1, where we work on a binary tree and want to start a new thread to work on each node. In Listing 24 we can see how we can use the new abstract read permission to encode the problem very intuitively. The new semantics of **rd(data.f)** allow us to split the permission into multiple parts and start the child threads. Furthermore it is possible to recombine the permission by joining the child threads. Note that the method and its specification can be used for any tree, even if it's depths is unbounded.

```
class Node {
  var l: Node; var r: Node;
  method work(data: Data)
    requires valid;
    requires rd(data.f);
  {
    unfold valid;
    var tkl: token<Node.work>
    var tkr: token<Node.work>
    if (this.l != null) { fork tkl := this.l.work(data) }
    if (this.r != null) { fork tkr := this.r.work(data) }
    /* ... (perform work on this node, using data.f) */
    if (this.l != null) { join tkl }
    if (this.r != null) { join tkr }
  }
  predicate valid {
    rd(this.l) && rd(this.r) && (this.l != null ==> this.l.valid) && (this.r !=
        null ==> this.r.valid)
  }
}
class Main {
  method main(tree: Node)
    requires tree != null && tree.valid;
  {
    var data: Data := new Data { data.f := 1 }
    fork tree.work(data);
  }
}
class Data { var f: int; }
```

Listing 24: Working on a binary tree with arbitrary permission splitting.

5.2.2 Information Hiding and Simplicity

The new abstract read permissions that correspond to unknown fractions also simplify the task of choosing the permissions to be used in the specification of a Chalice program. To illustrate this, let us again consider the example from Section 3.1.2 where we wanted to parallelize a method, but could only do so if the precondition required more than one epsilon permission.

In Listing 25 we can see how the new abstract read permission can be used regardless of whether the computation is done sequentially or in parallel.

```
class Worker {
  var f: int;

  method m1()
    requires rd(this.f)
    ensures rd(this.f)
  {
    call subtask_1()
    call subtask_2()
  }

  method m2()
    requires rd(this.f)
    ensures rd(this.f)
  {
    fork tk1 := subtask_1();
    fork tk2 := subtask_2(); // no problem, still permission left to start
        subtask 2
    join tk1; join tk2;
  }

  method subtask_1/2()
    requires rd(this.f)
    ensures rd(this.f)
  { /* perform some computation involving x ... */ }
}
```

Listing 25: No information hiding problem in the new permission model.

5.3 Performance

How long a verifier takes to verify challenging examples is an important measure when assessing a verifier. Since the new permission model is more expressive than the old model, it is interesting to see how much this increase of expressiveness costs in terms of performance.

5.3.1 Performance of Existing Features

To compare the performance of the implementation of the new permission model with the previous version of Chalice, we used the test suite that is available as a part of Chalice. More precisely, we measured the complete execution time of the invocation of the Chalice verifier for all examples in the test suite. The tests were run both with the most recent version of Chalice, which was HG changeset `1785795e3eae`, and with our implementation of the new permission model.

The tests were run on an otherwise idle machine with an Intel Core i7 620M processor and 4096 MiB of RAM. To ensure a low error, the measurements were repeated three times and the average over those runs has been used as a result.

Figure 2 shows an overview of the performance comparison, and there are several things to point out. First of all, the test `iterator` sticks out, as our implementation performs 1042.58% worse on this example. We tried to investigate the cause, and it seemed that the theorem prover is searching for a proof in the wrong direction first. In particular, changing the random seed of the theorem prover changes its performance drastically, ranging from 10 seconds up to 90 seconds. To investigate the issue we contacted Rustan Leino, one of the developers of Chalice. He [Lei11] pointed out that the bad performance is due to the use of non-linear arithmetic when percentages are used. In the encoding, we use `n*d` to represent `n` percentage

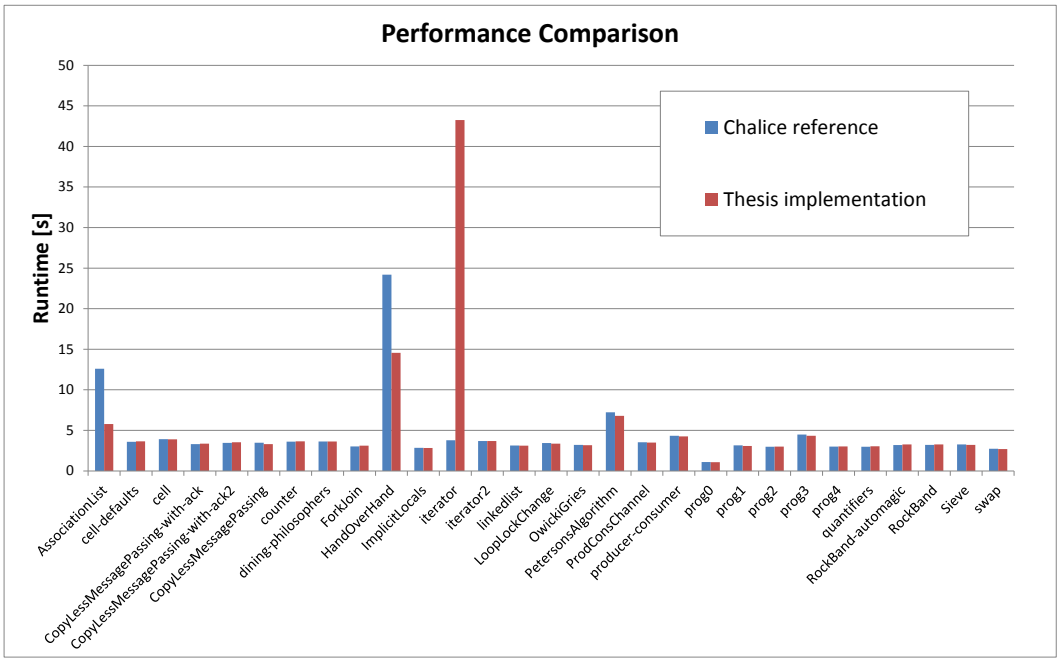

Figure 2: Performance comparison of the examples from Chalice's test suite.

	Chalice reference	Thesis implementation	Improvement
AVLTree	*859.22 s*	*819.44 s*	*4.63%*
AVLTree.iterative	9.99 s	8.55 s	14.43%
AVLTree.original	*1342.42 s*	*1135.81 s*	*15.39%*
Composite	93.33 s	97.67 s	-4.65%
AssociationList	12.59 s	5.78 s	54.06%
HandOverHand	24.19 s	14.56 s	39.80%

Figure 3: Performance comparison of the more challenging examples.

of a permission. The use of various heuristics when dealing with non-linear arithmetic in Z3 can cause large variation and might even be much worse in cases where the verification conditions are not valid. Rustan Leino also suggested several solutions to this problem which either eliminate the bad effects of non-linear arithmetic, or work completely without non-linear arithmetic. Because this was not work done as part of this thesis, we do not further describe these solutions here. However, the suggested solutions have been implemented later, and all of them solve the performance problems.

Figure 2 also shows that for all the small examples in the test suite the performance is almost identical. More precisely, the differences are all smaller than 6% and for the cases where our implementation is slower, the differences are even below 3.7%. For the two more challenging examples, `HandOverHand` and `AssociationList`, our implementation performs significantly better. To see whether our tool handles complicated examples better in general, we also tested several examples that are not part of the test suite. We looked at a Chalice encoding of the composite design pattern which we took from the Master thesis of Filip Wieladek [Wie10]. We also tested several variants of an AVL tree implementation in Chalice.

Figure 3 shows the average execution time (again on the same machine, using the same measurement procedure) of all challenging examples, including the two examples from Chalice' test

suite. The two test cases called AVLTree and AVLTree.iterative are very challenging for the theorem prover, and running further tests with different random seeds for the prover causes their performance to vary a lot. For those reason, those two tests appear in italics in Figure 3 and have to be considered with caution. The performance of our implementation for the composite pattern is about 5% slower than with the previous version of Chalice. All other examples however run faster with our implementation with improvements up to over 50%.

In summary, our implementation provides very competitive performance compared to the previous version of Chalice with the old permission model. It seems that for the more challenging examples, our implementation even provides better performance.

5.3.2 Performance of the New Features

It is difficult to test the performance of the new features, but the abstract read permissions do not seem to cause any performance problems. After all, various tests of the test suite use them, and the performance of those test is very competitive.

However, permission multiplication is rather slow. For instance, the verification of the program shown in Listing 26 takes 30.98 seconds on average. If we replace `rd(valid)` by the full predicate `valid` (such that no permission multiplication is necessary), the performance suddenly increases to 2.82 seconds on average.

Again it was Rustan Leino [Lei11] who pointed out that the performance discrepancy is due to the use of non-linear arithmetic. However, his suggestions for the percentages cannot be easily adapted to this more general case of multiplication.

```
class Cell {
  var f: int;

  predicate valid { rd(f) && f > 0 }

  method m()
    requires rd(valid);
  {
    unfold rd(valid);
    assert(rd(this.f,*));
    assert(rd(this.f)); // this assertion should fail
  }
}
```

Listing 26: Example that uses permission scaling.

6 Additional Work

Beside the core of this project and the extension, work has also been carried out that is not directly related to the project, but can still be considered part of this Bachelor thesis.

6.1 Bug Reports and Bug Fixes

During our work on Chalice we have found a number of bugs in Chalice and Boogie, for some of which we directly provided a patch. All patches and most errors have been submitted to the Chalice website at `http://boogie.codeplex.com`, and they include:

- Chalice: Duplicated member declarations (both fields and methods) in classes are not correctly detected in Chalice (see patch #7685 on boogie.codeplex.com).
- Chalice: Under certain circumstances, read locks allow write access (see workitem #10148).
- Chalice: Duplicated method parameters and return values are not checked by Chalice, which can lead to unexpected output from Boogie (see workitem #10147 and patch #8151).
- Boogie: Nested lambda expressions cause Boogie to crash (see workitem #10175).
- Chalice: If a **`fork`** statement is used inside of a loop, a `NullPointerException` can occur (see workitem #9978 and patch #7636).
- Chalice: The command-line switch `-noDeadlockChecks` introduces contradictory assumption, which allows the verification of **`assert false`** (see workitem #10007).
- Chalice: There is a copy-paste error in the implementation of the method `EpsilonsOf`, leading to erroneous behaviour (see patch #8152).
- Chalice: The pretty printer functionality contained several errors and used some outdated syntax. This has been fixed.

6.2 Improvements to Chalice' Prelude

The Boogie file that Chalice generates during the verification starts with the "prelude", which contains important definitions, axioms and global variables. This prelude was defined as a static piece of text in Chalice, always containing all information, regardless of the program to be verified. Depending on the features the program at hand uses, this prelude often contained information that is not needed for the verification. For instance, the sequences in Chalice are based on a sequence axiomatization that is part of the prelude. If a program did not use sequences at all, the Boogie file produced still contained the lengthy sequence axiomatization.

For this reason, the prelude is now built from smaller parts which are be included on demand. In particular, sections such as the sequence axiomatization are now only included if needed for the verification of the program at hand.

6.3 Compatibility with Scala 2.8.

Chalice is written in Scala and most parts of it were developed in 2009. With later releases of Scala, in particular with Scala version 2.8, the code of Chalice used more and more deprecated features. This resulted in many warnings during compilation. Some of these shortcomings have been removed during our work on Chalice, including:

- The method `flatten` in object `List` is deprecated in Scala 2.8. All uses of `List.flatten(xss)` have been replaced by `xss.flatten`.
- The method `first` in trait `IterableLike` is deprecated in Scala 2.8. All uses of this method have been replaced by `head`.
- The object `Math` in package `scala` is deprecated in Scala 2.8. Package `scala.math` is used now.

6.4 Improved Error Reporting

Under some circumstances, Chalice abruptly exits by throwing an exception. For instance, if the user tries to access a feature that has not yet been implemented, or when the program to be verified contains unsupported constructs such as scaling epsilon permissions with another epsilon permission. While these cases indeed are exceptional, it does not seem sensible to print a full stack trace.

Therefore, we now catch such exceptions and print a user-friendly error message, indicating what has gone wrong. It appears much more appropriate to have a clear error message, rather than a full stack trace in these cases.

Nevertheless, during debugging of Chalice or when new features are implemented, a full stack trace might still be useful. For this reason we introduced the command line switch `-showFullStackTrace`.

6.5 Submission to FTfJP 2011

Based on the work of this Bachelor thesis, K. Rustan M. Leino, Peter Müller, Alexander J. Summers and myself wrote a paper called *Fractional Permission without the Fractions*, which has been accepted at the workshop Formal Techniques for Java-like Programs at the 25th European Conference on Object-Oriented Programming 2011 [HLMS11].

7 Conclusions

7.1 Status of the Implementation

The Chalice verifier has been modified to use the new permission model, as described in this report. As a basis we used the SVN revision 57305 from `boogie.codeplex.com` (later renamed to HG changeset `77645fdef3e9`), the most recent revision at the time the project started. All major aspects of the new permission model have been implemented, with the exception of one recent extension to Chalice, which is called stepwise refinements [LY10]. It is unclear what will happen to that extension, and therefore we decided not to implement the new permission model in this case. At a later point in time support for this extension can be easily added. To simplify this even more, we used comments to mark all locations in the source code of Chalice that need to be considered when implementing support for these refinements.

To ensure the quality of the implementation, a new test suite with a large number of small test cases has been put together. These test cases typically cover one single aspect of the new model and examine its correctness. Furthermore, the existing test suite of Chalice has been ported to the new permission model (though only minor changes were necessary, as described in Section 5.1). All of these test cases pass.

One feature that will not work fully any more is the command line option `-autoFold`. This tries to automatically close a predicate when it cannot find it in the heap. There was not enough time to implement this feature again for the new model.

Another rarely used feature for which the new permission model could not yet be implemented is read locks. Again, this has not been implemented due to lack of time.

7.2 Future Work

A possible direction for the future is to investigate more closely how important the epsilon permissions are and whether it would be possible to remove them from the permission model. This is interesting for several reasons. First of all, the fractions as permissions are nicer from a conceptual point of view. But it would also be very interesting to see how an implementation that does not use epsilon permission performs. In particular, currently every proof obligation that checks read access introduces a disjunction. More precisely, we use the function `CanRead(o,f)`, which is defined as

```
let (p,n) be the permission to o.f in (p > 0 ∨n > 0)
```

We suspect that this disjunction might have negative effects on the performance, which is why a model without epsilon permissions might be interesting to consider.

Another possibility is to introduce comprehension expressions such as summation, product, minimum, maximum and count. This could be done in a similar way as in the Spec# program verifier [BLS05]. Specifically, it would be interesting to introduce at least summation for amounts of permission. For instance, this would allow the verification of examples which fork a dynamic number of n threads, whose tokens are stored in a sequence. The summation expression would then enable us to express the amount of permission that corresponds to sequence of tokens, even if the threads each hold a (possibly different) fractional amount of permission to some location.

A further direction would be improving the Boogie output that Chalice generates. In particular there are various assertions, assumption and statements in general that are superfluous, and which could be fairly easily identified as such. For instance, an assertion that **`this`** is not **`null`** does not have to be output at all, as this will always be the case. Another example are duplicated assumptions. Our implementation of the new permission model is also not optimal in this respect. In particular, the fractions are fixed regardless of whether they are actually

needed or not. For instance, if a method does not use read permissions in its contract, then fixing the fraction π_{method} at the beginning of the method's verification is not necessary, and could be omitted. Even though it should be fairly easy to prove such trivial assertions and ignore unnecessary assumptions, it might still be a significant performance problem. First of all, the number of such superfluous statements is quite significant, and lots of additional assumptions might mislead the prover and cause needless instantiations of axioms.

7.3 Conclusions

We presented a new permission model for Chalice based on the idea of fractional permissions. This new model allows us to split permissions arbitrarily often. Furthermore it provides better abstraction and information hiding by using a single expression for all read permission instead of forcing the user to specify precise amounts.

We then extended the model further by generalizing permissions such that they allow arithmetic operations on permission amounts. Furthermore it is possible to refer to the amount of the newly introduced abstract read permissions.

All these improvements and changes to the permission model have been implemented in the Chalice verifier. Our implementation is almost fully backwards compatible and provides roughly the same performance, even though many more examples can now be verified due to the increased expressiveness.

7.4 Acknowledgements

I would like to thank my supervisor Dr. Alexander J. Summers for his continuous support during the project, the many fruitful and interesting discussions and his valuable feedback. I would also like to thank Prof. Dr. Peter Müller for giving me the opportunity to work on this exciting research project and for giving me a glimpse of state-of-the-art software verification. My thanks also go to Dr. K. Rustan M. Leino, who helped identify and solve the performance issue.

Even though writing the paper about fractional permissions was not part of this Bachelor thesis any more, I would particularly like to thank Rustan Leino, Peter Müller and Alexander Summers for giving me this amazing opportunity to work together on a paper. This was a great honour and a very rewarding experience.

List of Figures

List of Listings

References

[BLS05] Mike Barnett, Leino, and Wolfram Schulte. *The Spec# Programming System: An Overview*, volume 3362/2005 of *Lecture Notes in Computer Science*, chapter 3, pages 49–69. Springer, Berlin / Heidelberg, January 2005.

[Boy03] John Boyland. Checking interference with fractional permissions. In *Proceedings of the 10th international conference on Static analysis*, SAS'03, pages 55–72, Berlin, Heidelberg, 2003. Springer-Verlag.

[ByECD+06] 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 *Formal Methods for Components and Objects: 4th International Symposium, FMCO 2005, volume 4111 of Lecture Notes in Computer Science*, pages 364–387. Springer, 2006.

[dMB08] Leonardo de Moura and Nikolaj Bjørner. *Z3: An Efficient SMT Solver*, volume 4963/2008 of *Lecture Notes in Computer Science*, chapter 24, pages 337–340. Springer Berlin, Berlin, Heidelberg, April 2008.

[HLMS11] S. Heule, K. R. M. Leino, P. Müller, and A. J. Summers. Fractional permissions without the fractions. In *Formal Techniques for Java-like Programs (FTfJP)*, 2011.

[Lei08] K. Rustan M. Leino. This is Boogie 2, 2008.

[Lei11] K. Rustan M. Leino. personal communication, 2011.

[LM09] K. R. M. Leino and P. Müller. A basis for verifying multi-threaded programs. In G. Castagna, editor, *European Symposium on Programming (ESOP)*, volume 5502 of *Lecture Notes in Computer Science*, pages 378–393. Springer-Verlag, 2009.

[LMS09] K. R. M. Leino, P. Müller, and J. Smans. Verification of concurrent programs with Chalice. In A. Aldini, G. Barthe, and R. Gorrieri, editors, *Foundations of Security Analysis and Design V*, volume 5705 of *Lecture Notes in Computer Science*, pages 195–222. Springer-Verlag, 2009.

[LY10] K. Rustan M. Leino and Kuat Yessenov. Automated stepwise refinement of heap-manipulating code, 2010.

[Wie10] Filip Wieladek. Automatic verification of concurrent programs. Master's thesis, ETH Zurich, 2010.