# Fractional Permissions without the Fractions

Stefan Heule
ETH Zurich
stheule@ethz.ch

K. Rustan M. Leino
Microsoft Research
leino@microsoft.com

Peter Müller
ETH Zurich
peter.mueller@inf.ethz.ch

Alexander J. Summers
ETH Zurich
alexander.summers@inf.ethz.ch

## ABSTRACT

Fractional Permissions are a popular approach to reasoning about programs that use shared-memory concurrency. Abstractly, they provide a way of managing that either multiple readers or one writer thread can access a resource concurrently. Concretely, specification using fractional permissions typically requires the user to pick concrete mathematical values for partial permissions, making specifications overly verbose, tedious to write, and harder to adapt and re-use.

This paper contributes a flexible and expressive specification methodology for supporting fractional permissions while allowing the user to work at the abstract level of read and write permissions. The methodology is flexible and modular, and has been implemented in the verification tool Chalice.

## Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*formal methods, programming by contract*

## General Terms

Verification

## Keywords

Static verification, concurrency, fractional permissions

## 1. INTRODUCTION

An important part of reasoning about concurrent programs concerns their patterns of access to memory and other shared resources. A useful aid in the specification of such patterns is to use a model of resource *permissions* that can be transferred between program entities to specify how individual threads are currently allowed to access shared resources. By allowing permissions to be fractional [2], it is possible to distinguish between acceptable read and write behaviours, which is necessary for expressive reasoning about programs with shared-memory concurrency.


*FTfJP'11,* July 26, 2011, Lancaster, UK.


The traditional model of fractional permissions associates an access right with every memory location. The access right can be divided into fractions, which can be held by and transferred between threads and method activation records. An activation record can read a memory location only if it holds a non-zero fraction of the memory location's access right. To write to the memory location, the activation record must hold the entire access right. Since fractional permissions can only be divided and combined, but never forged or duplicated, this discipline ensures race freedom for memory updates while allowing concurrent reads.

Permissions are a fictional notion used for static reasoning about a program; they are used in specifications and during program verification, but are not present at program execution. The soundness argument of a successful verification guarantees that the program will be race-free, although the programming language itself allows memory races in general.

While fractional permissions give rise to a flexible model for reasoning, writing specifications in this model can be tedious and overly verbose due to the need to work with some concrete mathematical representation of the permissions. The problem is exacerbated by the fact that programmers are concerned with permissions only at the abstract level of reading or writing to a location; the concrete values representing these permissions are largely irrelevant.

In this paper we present a solution to this problem. Our approach gives a useful and flexible semantics to a clean syntax for talking about access rights, and allows the user to reason at the abstract level of read and write permissions. Our methodology is expressive, modular, and supports reasoning using locks/monitors, fork/join of threads, as well as loops and method calls. We have implemented a prototype in a modified version of the verification tool Chalice [3, 4].

### *Introductory Example.*

Let us use a simple example to illustrate the problem of writing specifications with fractional permissions. Consider a method `Twelve` that reads a field `f` and writes another field `g` as some function of `f`, here the product of `12` and `f`. The precondition of `Twelve` will thus require full (write) permission to `g`, which we will denote by **`acc(this.g, 1)`**, and read permission to `f`, written **`acc(this.f,`**$\pi$**`)`** for some non-zero fraction $\pi$. Here and throughout, we use a Chalice-like syntax, but there are other notations in use; for example, in separation logic [8, 6], these conditions are written:

$$\texttt{this.g} \overset{1}{\mapsto} \_ \;*\; \texttt{this.f} \overset{\pi}{\mapsto} \_$$

The postcondition of the method will ensure that these per-

missions are returned to the caller, so the full specification of the method has the following form:

```
method Twelve()
  requires acc(this.g,1) && acc(this.f,π)
  ensures acc(this.g,1) && acc(this.f,π)
```

The central problem is: what is $\pi$?

One option is to represent $\pi$ as a particular rational number, like `.18`. It is easy to see that this option can be rather limiting, because it forbids calls from a context where the caller has only, say, the fractional permission `.17` to `f`, despite the fact that such a fraction does imply read access.

A second option is to let the calling context decide what $\pi$ should be for a particular call. This can be achieved by making $\pi$ a parameter of the method. The parameter can be considered a *ghost* parameter, since it is needed only for the proof and can be omitted in the executing program.

```
method Twelve(ghost π: rational)
  requires 0 < π && π <=1 &&
           acc(this.g,1) && acc(this.f,π)
  ensures acc(this.g,1) && acc(this.f,π)
```

In separation logic, such a parameter $\pi$ is typically coded as a logical variable that is bound for each invocation of the method, but that is really the same thing as passing a ghost parameter. This option provides flexibility for callers, but comes at the cost of cluttering up the specification.

A third option is to use *counting permissions* [1], which allow one to split a write permission into any positive number of *indivisible* units. A unit then grants read access. Chalice supports counting permissions by providing a unit $\varepsilon$, which is a positive, arbitrarily small, and indivisible constant [3]. To indicate one $\varepsilon$ of permission, one can now use a notation like **rd(this.f)**, which yields the following specification:

```
method Twelve()
  requires acc(this.g,1) && rd(this.f)
  ensures acc(this.g,1) && rd(this.f)
```

The trouble with this option is that the indivisible nature of $\varepsilon$ does not respect procedural abstraction, that is, the property that an implementation can choose *how* to establish *what* the specification says. Consider an implementation of `Twelve` that first calls subroutines `Eight` and `Four` (each of which requires one $\varepsilon$ permission to `f`), and then combines the results of these. Since these subroutines just read `f`, it seems reasonable that `Twelve` could call them in parallel. But to make the calls in parallel would require 2 $\varepsilon$ permissions, which `Twelve` does not have. Hence, the indivisible nature of $\varepsilon$ would necessitate different specifications of methods, depending on how many threads need concurrent read access, which goes against the grain of procedural abstraction.

In this paper we present a specification methodology for permissions that provides most of the flexibility and expressiveness of fractional permissions, without requiring a concrete mathematical representation. In a nutshell, we allow programs to use abstract read permissions with a meaning similar to **acc(this.f,π)** as in the second option above, but without requiring the user to manually specify the $\pi$ values.

## 2. BACKGROUND

Chalice is a concurrent programming language and a program-verification tool [3]. Chalice associates permissions with memory locations, as described in the introduction, and the permissions (notionally) held by a method activation can be fractional [2]. The Chalice language builds in specification constructs, like the pre- and postconditions shown in the introductory example above. The specifications are written in the style of implicit dynamic frames [9], which means they include access predicates like **acc** and **rd**. Access predicates instigate the transfer of permissions at the point where the specification takes effect. For example, a method precondition **acc(this.f,p)** says that, at the time of a call to the method, a fraction `p` of the permission[1] to **this.f** is transferred from caller to callee, and the caller meets the precondition only if it possesses at least this amount of permission to **this.f** at the time of call.

The semantics of Chalice [3] is formalized using, in a standard way, a variable $\mathcal{H}$ that maps memory locations to values. To keep track of permissions, the formalization also associates with each method activation a map $\mathcal{P}$ from memory locations to the method activation's permission to that location. Initially, the permission mask is empty, that is 0 for all locations. The semantics of read and write statements in Chalice are then defined as follows: for `x := o.f;`,

$$\textbf{assert } o \neq \mathit{null} \wedge \mathcal{P}[o,f] > 0; \quad x := \mathcal{H}[o,f];$$

and for `o.f := x;`,

$$\textbf{assert } o \neq \mathit{null} \wedge \mathcal{P}[o,f] = 1; \quad \mathcal{H}[o,f] := x;$$

where an **assert** indicates a proof obligation that has to be checked by the verifier.

The treatment of specifications is formalized using two operations *inhale* and *exhale*. Analogously to sequential verification, in which a method precondition is checked at a call site and assumed inside the method body, Chalice says that the caller exhales the precondition and the callee inhales it.

The inhale and exhale operations are defined recursively over the syntax of specifications. For an expression $E$ without access predicates, exhaling $E$ means **assert** $E$ and inhaling $E$ means **assume** $E$, where **assume** indicates a condition that the verifier is allowed to assume. Exhaling an access predicate **acc(o.f, p)** means

$$\textbf{assert } p \leq \mathcal{P}[o,f]; \quad \mathcal{P}[o,f] := \mathcal{P}[o,f] - p;$$

Inhaling **acc(o.f, p)** is its dual, but with a twist:

$$\textbf{if } (\mathcal{P}[o,f] = 0) \ \{ \ \textbf{havoc } \mathcal{H}[o,f]; \ \} \quad \mathcal{P}[o,f] := \mathcal{P}[o,f] + p;$$

where **havoc** sets the given location to an arbitrary value. The conditional **havoc** in the definition of inhale is crucial to the correct modelling of state updates by other method activations, including those in different threads. It says that if a memory location `o.f` was not previously readable by the current method activation, then nothing is known about its value (other than any properties that are assumed in the rest of the inhale operation). In contrast, if the current method activation already holds a non-zero permission to `o.f`, no other method activation can have the full (write) permission to `o.f`, and so whatever the current method activation already knows about the value `o.f` can be soundly retained.

Because of the recursive definition of exhale, to exhale an expression such as **acc(o.f,p) && acc(o.f,p)** we exhale **acc(o.f,p)** *twice*; the expression is essentially equivalent to **acc(o.f,2p)** (the conjunction behaves *multiplicatively*). This

[1]Permission amounts are strictly positive; see Section 5.

accurately reflects that permissions should be treated as resources, avoiding the duplication or accidental merging of permissions. This conjunction can be formally related to the separating conjunction of separation logic [7].

The current version of Chalice denotes fractional permissions by `acc(o.f, p)`, where `p` is an integer percentage between 1 and 100, and counting permissions by `rd(o.f)`, which denotes one $\varepsilon$ permission, discussed as the third option in the introductory example. `acc(o.f)` is an abbreviation of `acc(o.f, 100)` and denotes a write permission. In this paper, we abandon percentages and $\varepsilon$ permissions; the syntax of our new permission model distinguishes only between read and write permissions, and redefines **rd** to have a more flexible semantics than that used before.

# 3. THE NEW PERMISSION MODEL

The motivation for our permission model is to allow the user to reason abstractly, at the level of read/write permissions. To enable this view, we use two main kinds of permissions. A *full permission* (corresponding to a "1"-valued fractional permission) to a location `o.f` is denoted by `acc(o.f,1)` or, for brevity, simply `acc(o.f)`. Such a full permission allows a thread to both read and write the location. A corresponding *read permission* is denoted by `acc(o.f,rd)` or simply `rd(o.f)`. These read permissions are abstract in the sense that they do not concretely specify an associated mathematical fraction of the full permission. Such a permission can be understood to represent a *fixed, positive and unknown* amount of permission to `o.f`. The fact that a read permission represents a positive amount of permission means it is sound to assume that we can read from the location, regardless of what the actual amount is.

In our permission model, the concrete permission amounts associated with `rd(o.f)` permissions are never chosen, but these values are *constrained* at various points, to give useful properties. Note that we do not insist that the amount of permission associated with each **rd** predicate is the same; in general, two such read permissions correspond to different fractional permissions, regardless of whether or not they are concerned with the same location. The main challenge in our design is to identify where and how we should constrain these values to express useful specification cases.

## 3.1 Method Implementations

One common idiom is that a method requires some permission to a location `o.f` and returns this permission to its caller. That is, both the pre- and postcondition mention some read predicate `rd(o.f)`. In this case, we want to be able to express the common situation that we return the *same* amount of permission to the caller, who may then be able to recombine this permission to obtain full permission again. For instance, consider the following example:

```
method main(c: Cell)
  requires acc(c.val)
{
  c.val := 0
  call m(c)
  c.val := 1
}
method m(c: Cell)
  requires rd(c.val)
  ensures rd(c.val)
{ /* ... */ }
```

Even though method `m` requires read access to `c.val`, it is desriable that it is still possible to write to the location after a call to `m` if we started with write access to this location. For this reason, if `rd(o.f)` occurs in both the pre- and postcondition of a method, it should be interpreted as the same (fixed, unknown) amount. Because we cannot statically know full aliasing information, this would not be enough to prove methods that return the same permission to the caller, but via some other reference to the same object. For this reason, we use a simple rule: *For any given method call, every* ***rd*** *predicate occurring in the method contract is interpreted as holding the same fixed amount of permission.*

To verify a particular method implementation, we first fix a fraction $\pi_{\text{method}}$, which is used to interpret any **rd** predicate in the contracts of that method. No precise value is given to $\pi_{\text{method}}$, we assume only that it is a valid permission:

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

Otherwise, the method is verified as usual: we inhale the precondition, execute the method body and then exhale the postcondition. Because we do not choose a specific value form $\pi_{\text{method}}$, a successful verification actually proves that the method is correct for *any* permission amount $\pi_{\text{method}}$.

## 3.2 Method Calls

A call to a method `m` is verified using `m`'s contract, which may mention read permissions. Since we verify that the implementation of `m` is correct for any permission amount that one might use to interpret its read permissions, the caller is free to choose any fraction to interpret those permissions.

If the called method `m` requires a 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, intuitively we can always find a (positive) fraction that is smaller than what we currently hold, and we can give this permission amount to the method. We again use an underspecified fraction $\pi_{\text{call}}$ to interpret all **rd** predicates in the specification of such a called method `m`. $\pi_{\text{call}}$ is constrained to be a positive amount that is less than the permission currently held, for any location for which read access is required by `m`.

At a method call we first introduce a variable $\pi_{\text{call}} > 0$. Then, we exhale the called method's precondition. For each read permission to be exhaled (i.e., each occurrence of `rd(o.f)` for some `o.f`), we check that we have a positive amount of permission in our mask to `o.f`, and we constrain $\pi_{\text{call}}$ to be smaller than this positive amount we currently hold. Next, we subtract $\pi_{\text{call}}$ from the permission mask and continue through the precondition. This encoding ensures that the called method `m` is provided with the required read permissions for each relevant location while `m`'s caller also retains read permissions to those locations. The latter lets the caller prove that `m` does not modify those locations.

We can now see why interpreting all read permissions in a method specification as representing the *same* amount (regardless of the corresponding memory location) is not a restriction: the amount is always implicitly chosen to be smaller than any corresponding amount held by the caller. If multiple read permissions to the same location are mentioned, then we effectively just assume that we can find a fraction that is small enough such that giving away all of those read permissions is allowed. This is achieved by constraining $\pi_{\text{call}}$ multiple times, once for every read permission.

After a call to a method `m`, we inhale `m`'s postcondition,

using the same value $\pi_{\text{call}}$ to interpret any **rd** permissions mentioned. This allows us to regain the same amount of permission we have given away, if it is mentioned in both the pre- and postcondition. So in the above example, method `main` regains write permission to `c.val` after the call to `m` and thus, may update the field.

Note that, because a different fraction is used for each method activation, the scheme described above works even for recursive method definitions (which would not easily be the case using concrete fractions).

## 3.3 Asynchronous Method Calls

Chalice supports asynchronous method calls, using **fork** and **join** statements. A statement `tk := fork m()` forks off a new thread that executes the method `m`, and returns a *token* which can be used to *join* the thread and wait on the result `r` of the call, using a statement `r := join tk`. The verification of asynchronous calls is analogous to synchronous calls; when a thread is forked to execute a method `m`, the method's precondition is exhaled; its postcondition is inhaled only when the forked thread is joined.

For the same reasons as for synchronous method calls, it is convenient to interpret all **rd** permissions mentioned in a method specification as representing the same permission amount. In particular, if a **fork** and its corresponding **join** occur in a scoped fashion (in the same method body), we would like to be able to express that we can match up the same permission amounts from corresponding **rd** predicates. We encode this by adding a *ghost field* to tokens, which represents the permission amount used to interpret **rd** predicates for the associated asynchronous call. We store this value in the ghost field when a token is created at a **fork** statement, and refer to it when interpreting the permissions returned at a **join**. This value is never changed; if we encounter **fork**/**join** statements on the same path through a method body, the same permission fraction is known to be used for both. However, if the **join** takes place in a different method body, no information will be known about this fraction; it is effectively arbitrary (although positive). In Section 5 we show how to avoid this loss of information.

## 3.4 Monitors

Chalice allows any object to be a *monitor*. Initially, any object is thread-local, and it can be made available using the statement **share**. A monitor has an associated *monitor invariant*, describing the permissions held and properties guaranteed while the monitor is shared and unlocked. A thread that holds the monitor of an object `o` can also make the object thread-local again, using a statement **unshare** `o`. For verification purposes, when an object is being acquired or unshared, the monitor invariant is inhaled, and when the lock is released or shared, it is exhaled. Permissions to a location are sometimes split over several monitors, such that each monitor stores only a read permission; consequently, we need our methodology to give an appropriate semantics to **rd** predicates in monitor invariants.

For read permissions in monitor invariants, it is useful to be able to prove that when a monitor is released and later re-acquired, the same fraction is exhaled and inhaled. The same is true for sharing and later un-sharing the monitor. Using the same fraction in the **release**/**acquire** (or **share**/**unshare**) pair is sound only if each **acquire**/**release** pair that might be executed in between leaves the permission stored in the monitor unchanged. We enforce this by fixing the fraction $\pi_{\text{monitor}}$ used for interpreting read permissions in a monitor invariant when the corresponding object is created. It is constant throughout the object's lifetime and used whenever the invariant is exhaled or inhaled.

Choosing the interpretation of read permissions as soon as the object is created is less flexible than our approach for method calls, where we choose an amount for each individual call. For instance, if we have a read permission to a location `o.f`, we cannot share an object that mentions **rd**`(o.f)` in its monitor invariant. The problem is that the fraction we currently hold might be smaller than $\pi_{\text{monitor}}$, which is constrained only by $0 < \pi_{\text{monitor}} \leq 1$. An example that does not verify because of this inflexibility is the following:

```
class C {
  method main(l: Lock)
    requires rd(l.f)
  {
    release l // Err: possibly insufficient permission
  }
}
class Lock { var f: int; invariant rd(f) }
```

We present two solutions to this problem in this paper.

# 4. LOSING PERMISSION

It is usually beneficial to use the same permission fraction to interpret **rd** permissions for the pre and postcondition of one method call, and similarly for the monitor invariant of one object. However, this can be too restrictive when methods give away permissions during execution, or similarly when permission is given away between acquiring and releasing a monitor (perhaps to other threads, or to a newly-shared monitor invariant). For example, consider a method `m` that requires read access to fields `f` and `g` of some object `c`. Suppose that `m` forks off a worker thread that requires read permission to `c.g`, but does not join that thread. After the fork, `m` still has read permission to `c.g`. However, with the encoding introduced so far, we cannot return this permission to the caller. Putting **rd**`(c.g)` in `m`'s postcondition would require `m` to return the exact fraction that it received from its caller. However, that is not possible when part of that fraction has been passed to the worker thread.

To handle these situations, we introduce a *starred* read permission, denoted by **rd**$*$`(c.g)`. This has the meaning of introducing another positive, but otherwise unrelated fractional amount. In particular there is no guarantee that it corresponds to the same amount as any other permission used in the same method signature or monitor invariant. Because no information about this permission amount (other than it being positive) is ever assumed, when exhaling such a **rd**$*$ predicate it is sufficient just to check that *some* permission to the appropriate location is available, and then to assume the actual amount to be smaller than this.

Starred read permissions are useful for method postconditions and monitor invariants, but not for method preconditions, for which the caller is free to choose an interpretation anyway. Similarly, the standard read permission **rd** is useful in method preconditions and monitor invariants. In method postconditions, it makes only sense when the precondition also contains a standard read permission. Otherwise, the fraction used for the interpretation is arbitrary, and it would be clearer to use a starred read permission **rd**$*$.

Using this new permission syntax, we can then specify the example described above as:

```
method m(c: Cell)
  requires rd(c.f) && rd(c.g)
  ensures rd(c.f) && rd*(c.g)
{
  tk := fork worker(c)
}
method worker(c: Cell)
  requires rd(c.g)
  ensures rd(c.g)
{ /* ... */ }
```

# 5. PERMISSION EXPRESSIONS

Using our design so far, we cannot denote the amounts of permission associated with **rd** predicates in a particular monitor, or in the specification of a particular forked thread. In situations where we do not need to be precise about these amounts, the **rd***  predicates allow us to abstract them away, but this is necessarily at the expense of precision; as soon as a **rd*** predicate is used, the permission to that location cannot be recombined to give a full permission any more. For example, in cases where we use **fork**/**join** or **release**/**acquire** across method boundaries, so far we cannot preserve the information about these permission amounts. We also have no way in the model to express the *difference* between two permission amounts. For instance, if we start with a full permission and give away a **rd** permission, we cannot yet express what we have left over, which (as we will see) is essential for some examples. In this section we introduce *permission expressions*, which amend these defects.

We generalise accessibility predicates to the new form **acc**(o.f, $P$) where $P$ is a *permission expression*. Permission expressions $P$ are defined inductively as follows:

| | | | |
|---|---|---|---|
| $P$ | ::= | 1 | *full permission* |
| | \| | **rd** | *read permission* |
| | \| | **rd**(tk) | *token read permission* |
| | \| | **rd**(o) | *monitor read permission* |
| | \| | $P_1 + P_2$ | *permission addition* |
| | \| | $P_1 - P_2$ | *permission subtraction* |

*where* tk *is a token and* o *is a reference-typed expression*

The permission expressions 1 and **rd** are used as before, to denote full and read permissions in a specification (where the read permissions are interpreted depending on the context; e.g., per method call if used in a method specification). The expressions **rd**(tk) and **rd**(o) are new, and can be used to refer to the amount of permission associated with read permissions for a particular asynchronous call (via its token) or monitor. Finally, we support addition and subtraction of permission expressions. This allows us to express *differences* such as 1 - **rd**(o), meaning that we hold a full permission minus the permission amount associated with **rd** predicates for the monitor o (in such a case, acquiring the monitor would be enough to regain a full permission).

There is one technical difficulty to consider for this extension: because the permission expressions $P$ can include subtractions, it is possible to write expressions such as **rd** - 1 or **rd** - **rd**(tk) which are not guaranteed to denote valid (that is, positive) permission amounts. Exhaling such permission amounts naïvely could result in a total permission of more than 1 in a method activation record, leading to unsoundness. In addition, if we allowed permission expressions to potentially denote *zero* permission, we would effectively introduce the need, whenever such a permission is inhaled, of considering cases for whether or not any positive permission is received. To avoid these complications, we impose an additional well-formedness constraint, that a permission expression $P$ must provably denote a *strictly positive* amount of permission, whenever it is exhaled.

The example in Listing 1 illustrates the use of permission expressions. It shows pseudo-code for a program in which some optimization problem should be solved. The class Solver takes a problem instance and, via the method start, forks off a probabilistic search for a solution (method solve) and returns a token to its caller with which the result can be retrieved. This way several threads can work concurrently on the same problem to increase the probability of finding an optimal solution.

```
class Client {
  method main(p: Problem, s : Solver) returns (r: int)
    requires acc(p.f)
    ensures acc(p.f)
  {
    tk1 := call s.start(p)
    tk2 := call s.start(p)

    r1 := join tk1; r2 := join tk2
    r := max(r1,r2)
  }
}
class Solver {
  method start(p: Problem)
    returns (tk: token<Solver.solve>)
    requires rd(p.f)
    ensures acc(p.f, rd-rd(tk))
  { /* initialize, etc. */ tk := fork solve(p) }

  method solve(p: Problem) returns (r: int)
    requires rd(p.f); ensures rd(p.f)
  { /* ... */ }
}
class Problem { var f:int; }
```

**Listing 1: Example using generalized permissions.**

In our example, the client uses two threads and combines their results after joining the tokens. We assume that the attempts to solve the Problem require read access to the problem data, represented by the field f. The client starts with full access to this field, and would like to regain full access before it terminates. Since the method solve requires and returns read access to the problem data, we specify this with appropriate **rd** predicates. The method start has to fork a call to solve, and thus also needs read permission to p.f in its precondition. However, it does not return all of this permission to its caller; some is passed to the forked thread. It returns to the caller the "left over" permission, along with the token for the forked thread, using permission expressions: **acc**(p.f,**rd**-**rd**(tk)).

This specification supports a flexible concurrent implementation without concrete permission values, and provides enough information to the client code to guarantee that the full permission can be regained after joining the two threads. The starred read permission we have seen earlier is still useful in cases where the same amount is not desired.

Note that in our new permission model, permission expressions replace Chalice's percentage-expressions. A common idiom in Chalice is to split a full permission over two

monitors (or over a monitor and a method precondition), for instance, by specifying `acc(x.f,40)` for one and `acc(x.f,60)` for the other. The full permission is then obtained by acquiring both monitors. We achieve the same effect by specifying `acc(x.f,rd)` for one and `acc(x.f,1-rd)` for the other, without committing to arbitrary percentages. Consequently, we drop percentage-permissions from the language.

# 6. IMPLEMENTATION

We have implemented our new permission model in the Chalice program verifier. To evaluate the performance of our approach we compared execution times of existing examples using our prototype implementation and the original Chalice version. We used the 29 programs from the Chalice test suite. These have an average verification time of 4.5 seconds on our test machine. Four of the test cases required trivial changes to reflect the changes in the permission model.

The comparison shows that our implementation provides very similar speed; on average our implementation is 3.6% faster, and the performance changes range from -3.1% up to 50.6%. At this point it is not entirely clear why some of the examples perform significantly better with the new permission model. It appears that especially larger examples can be verified faster with our new permission model.

# 7. CONCLUSIONS AND FUTURE WORK

We have presented a methodology for specifying concurrent programs using a notion of splittable permissions, but without the need for the specifier to reason using a concrete mathematical model or syntax. This makes specifications less verbose and tedious to write, but more importantly allows them to express the programmer's intentions at the correct level of abstraction; ultimately it is sufficient to know the locations a thread can currently read and write to, in order to understand (and verify) its intended behaviour. By picking a judicious default semantics for read permissions in method contracts, we are generally able to retain the expressiveness afforded by concrete specifications. In implementations which pass read permissions between threads and monitors in flexible ways, while eventually recombining them into full permissions, our simple permission expressions allow for specifications which are precise about the important relationships between permission values, while still freeing the specifications from a concrete model or values. This also allows for greater flexibility when building verifiers such as Chalice; underlying design decisions about the concrete handling of permissions are invisible at the source level, allowing the underlying implementation to be seamlessly changed. As an example, we plan for future work to experiment with encoding permissions as rational and real numbers (as supported by the Z3 theorem prover) and compare performance. These experiments will not require any change to our specification methodology or surface syntax.

Our permission expressions support a bounded number of read permissions to be added/subtracted. A natural extension is to handle the unbounded case. For example, our randomised algorithm example in Listing 1 could be extended to fork off some statically-unknown number of solvers in a loop, and then rejoin all of the corresponding tokens. This would require permission expressions to express mathematical sums over permission amounts. In the example we would sum over indexed sequences of `rd(tk`$_i$`)` expressions.

Recent work of Militão et al. [5] generalises fractional permissions to user-defined *views*, which can describe permissions to sets of fields, and extra properties such as uniqueness of references. They also employ fractions whose values are hidden from the user, but do not have an analogue to permission expressions in specifications (cf. Section 5).

The possibility of reasoning about permissions in the absence of *any* concrete mathematical model is itself interesting. We are interested in further exploring our methodology in its own right, without the need *per se* of any interpretation of the underlying values of read permissions. Our work seems potentially to provide a means of explaining and formalising the use of permissions (which are merely a reasoning tool themselves), at a level of abstraction which corresponds with programmers' thinking about their code.

## Acknowledgements

We would like to thank the attendees of the Dublin Concurrency Workshop 2011, particularly Andrew Butterfield and Peter O'Hearn, for encouraging feedback on a preliminary presentation of this work.

# 8. REFERENCES

[1] R. Bornat, C. Calcagno, P. O'Hearn, and M. Parkinson. Permission accounting in separation logic. In *Principles of Programming Languages (POPL)*, pages 259–270. ACM, 2005.

[2] J. Boyland. Checking interference with fractional permissions. In R. Cousot, editor, *Static Analysis (SAS)*, volume 2694 of *LNCS*, pages 55–72. Springer, 2003.

[3] 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 *LNCS*, pages 378–393. Springer, 2009. Open source code at `boogie.codeplex.com`.

[4] 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: FOSAD 2007/2008/2009 Tutorial Lectures*, volume 5705 of *LNCS*, pages 195–222. Springer, 2009.

[5] F. Militão, J. Aldrich, and L. Caires. Aliasing control with view-based typestate. In *Formal Techniques for Java-Like Programs*, FTFJP '10, pages 7:1–7:7. ACM, 2010.

[6] M. Parkinson and G. Bierman. Separation logic and abstraction. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '05, 2005.

[7] M. J. Parkinson and A. J. Summers. The relationship between separation logic and implicit dynamic frames. In G. Barthe, editor, *European Symposium on Programming (ESOP)*, volume 6602 of *LNCS*. Springer, 2011.

[8] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Logic in Computer Science (LICS)*. IEEE, 2002.

[9] J. Smans, B. Jacobs, and F. Piessens. Implicit dynamic frames: Combining dynamic frames and separation logic. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 148–172. Springer, 2009.