Abstract Read Permissions: Fractional Permissions without the Fractions

Stefan Heule[1], K. Rustan M. Leino[2], Peter Müller[1], and Alexander J. Summers[1]

[1] ETH Zurich, Switzerland
`stheule@ethz.ch`, `peter.mueller@inf.ethz.ch`,
`alexander.summers@inf.ethz.ch`
[2] Microsoft Research, USA
`leino@microsoft.com`

Abstract. Fractional Permissions are a popular approach to reasoning about programs that use shared-memory concurrency, because they provide a way of proving data race freedom while permitting concurrent read access. However, specification using fractional permissions typically requires the user to pick concrete mathematical values for partial permissions, making specifications overly low-level, tedious to write, and harder to adapt and re-use. This paper introduces *abstract read permissions*: a flexible and expressive specification methodology that supports fractional permissions while allowing the user to work at the abstract level of read and write permissions. The methodology is flexible, modular, and sound. It has been implemented in the verification tool Chalice.

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 [4], it is possible to distinguish between acceptable read and write accesses, which is necessary for expressive reasoning about programs with shared-memory concurrency. 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 traditional model of fractional permissions associates an access permission with every memory location. The permission can be divided into fractions, which can be held by and transferred between threads and method activation records. An activation record may read a memory location only if it holds a non-zero fraction of the memory location's permission. To write to the memory location, the activation record must hold the entire permission. 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. In the verification of both sequential and concurrent programs, permissions also enable *framing*; as long as the caller of a method holds on to a non-zero fraction of the permission for a memory location, it may soundly assume that the call will not affect the value stored in the location, because the callee method cannot obtain the full (write) permission to modify it.

While fractional permissions give rise to a flexible model for reasoning, writing specifications in this model can be tedious and overly low-level 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 in the abstract sense of reading or writing to locations; the concrete values representing these permissions are largely irrelevant.

In this paper, we present *abstract read permissions*; a novel specification methodology that allows the programmer to reason at the level of read and write permissions. Our methodology is expressive, modular, and sound. We present it in the context of implicit dynamic frames [20], but it also applies to other permission logics; in particular, separation logic [19]. This paper builds on our previous workshop presentation [8]. Our methodology is implemented in two verifiers for the class-based, concurrent language Chalice [13]; one based on verification condition generation [12] and one on symbolic execution [11].

Outline. The next section motivates abstract read permissions through an example. Sec. 3 summarizes the background concerning permission-based verification that is used in the rest of the paper. We present the core ideas of abstract read permissions in Sec. 4, and the syntax of permission expressions in Sec. 5. Sec. 6 explains the encoding of permissions in the program verifier, and Sec. 7 provides an informal soundness argument. Sec. 8 discusses issues and solution approaches for the use of abstract read permissions in monitor invariants. We discuss related work in Sec. 9 and conclude in Sec. 10.

2 Motivation

To understand the problem of writing specifications with fractional permissions, consider a class `Expr` for arithmetic expressions with a method `eval(s: State)`, which evaluates an expression in the state `s`. The method reads the state's mapping of variables to values, denoted by `s.map`. The precondition of `eval` requires read permission to `s.map`, written **`acc`**`(s.map,` π`)` for some non-zero fraction π. Here and throughout, we use a Chalice-like syntax, but there are other notations in use; for example, in separation logic [17, 3, 18] this condition is written as `s.map` $\overset{\pi}{\mapsto}$ _. The postcondition of the method ensures that this permission is transferred back to the caller, allowing it to re-assemble a full permission and to update the state. The resulting specification is displayed in Listing 1. The problem that we address is how to specify the permission amount π. To illustrate the challenge, we present three existing options and discuss their shortcomings.

```
  method eval(s: State)
    requires acc(s.map, π) && s.map ≠ null
    ensures acc(s.map, π)
```

Listing 1. Specification of method `eval`. The method also requires read permission to the fields of its receiver, but we ignore this aspect here for brevity.

```
class Add extends Expr {
  var left, right: Expr

  method eval(s: State) {
    leftVal := call left.eval(s)
    rightVal := call right.eval(s)
    return leftVal + rightVal
  }
}
```

Listing 2. An implementation of method `Add.eval`. The specification of `eval` is presented in Listing 1.

Concrete Fractions. One option is to represent π as a *concrete fraction*, such as `.18`. This approach has four major shortcomings:

(1) *Re-usability*: Using a concrete fraction forbids calls from a context in which the caller has a smaller fractional permission to `s.map`, say `.17`. If the specifications cannot be changed to use different fractions (for instance, when `eval` and its caller are library methods), the call is not permitted even though the caller holds a read permission, and `eval` needs read permission.

(2) *Abstraction*: For many implementations, the permission amount is irrelevant so long as it provides read access. Nevertheless, this approach forces programmers to choose a specific value and, thus, to clutter up the specification.

(3) *Framing*: Consider a subclass `Add` of `Expr` that represents the addition of two sub-expressions. Its implementation of `eval` recursively evaluates the left and right operands, and returns the sum of the results, as shown in Listing 2.

The recursive call `left.eval(s)` consumes the entire permission amount that the caller has (here, `.18`) before returning the permission to the caller. Therefore, modular verification of `Add.eval` must make the worst-case assumption that the callee (which is dynamically bound to a statically-unknown implementation) might obtain full permission to `s.map` and modify the field. Thus, the first recursive call removes any information that the caller has about the value of `s.map` before the call: in particular, that the field has a non-null value. Verification of the second recursive call then fails, since the second conjunct of the precondition (i.e., `s.map` $\neq$ **`null`**) cannot be proved. Failing to frame this information even though the calls require only read permission weakens verification considerably.

(4) *Permission splitting*: Since the recursive calls to `eval` only read `s.map`, it seems reasonable that they could be called in parallel, as shown in Listing 3.

```
leftTk := fork left.eval(s)
rightTk := fork right.eval(s)
return (join leftTk) + (join rightTk)
```

Listing 3. A parallel implementation of method `Add.eval`. In Chalice, the **`fork`** statement launches an asynchronous method call, which entails checking the method's precondition and transferring the specified permissions to the new thread. The local variables `leftTk` and `rightTk` store thread identifiers (*tokens*) that are used in the **`join`** statements to join the threads and to obtain the results of the forked method calls, as well as the permissions they return.

However, there is no concrete fraction that one could choose for π that lets one verify this implementation. The entire permission of the forking thread is transferred to a new thread during the first fork, meaning that no permission is left to satisfy the precondition of the second fork.

Counting Permissions. A second option is to use *counting permissions* [3], which allow one to split a write permission into any positive number of *indivisible* units. Such a unit then grants read access. So we could represent π in the specification of `eval` as a number of such units; say one unit.

Counting permissions address the first two shortcomings of concrete fractions, but not shortcomings (3) and (4): as with concrete fractions, the recursive call `left.eval` in Listing 2 consumes all of the permission to `s.map` that the caller has and, thus, doesn't permit the framing of any information about its value; the second call then does not verify for the same reason as before. Moreover, a specification using a fixed number of units does not permit the parallel implementation in Listing 3 since, again, the first fork consumes the entire permission of the forking thread such that no permission remains to satisfy the precondition of the second fork. The number of units used in a method pre-condition also imposes a static bound on the amount of parallelization possible in a method implementation, breaking abstraction and impairing re-use.

Ghost Parameters. A third option is to let the calling context decide which permission amount to transfer to a method call; that is, what π should be for a particular call. This can be achieved by making π 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. In separation logic, such a parameter π is typically represented as a logical variable that is bound for each invocation of the method. Both approaches are very similar: ghost parameters require programmers to provide a value for the ghost parameter when implementing a call, whereas logical variables require the verifier to provide a value when reasoning about a call. The specification for method `eval` using a ghost parameter is shown in Listing 4.

When this option is applied to *all* methods that require read permission, it addresses many of the shortcomings described above. It solves the problems of

```
method eval(s: State, ghost π: rational)
  requires 0 < π && π ≤ 1 && acc(s.map, π)
  ensures acc(s.map, π)
```

Listing 4. Specification of method `eval` using a ghost parameter π.

re-use, framing, and permission splitting by allowing a caller to choose a value for the ghost parameter that ensures that it has enough permission to make the desired calls (even in parallel), and that some permission remains, to enable framing. The abstraction problem is reduced, but specifications are still cluttered up with ghost parameters (or logical variables) and constraints over them.

The abstract read permissions presented in this paper provide most of the flexibility and expressiveness of the ghost-parameter option above, but without requiring the user to manually specify the π values or constraints over them. In particular, our specification methodology does not require a concrete mathematical representation of permission amounts; especially no concrete fractions.

3 Background

Abstract read permissions are independent of any particular permission logic or verification technique. For concreteness, we present them for the Chalice language and verifier [12], which is based on the verification condition generator Boogie [1]. This section introduces the background on Chalice that is needed in the remainder of the paper.

Permissions. Chalice associates permissions with memory locations, as described in the introduction, and the permissions held by a method activation can be fractional [4]. The Chalice language builds in specification constructs, such as the pre- and postconditions shown in the example above. The specifications are written in the style of implicit dynamic frames [20], which means they include *accessibility predicates*. Accessibility predicates instigate the transfer of permissions at the point where the specification takes effect. For example, a method precondition **acc(this**.f, π**)** says that, at the time of a call to the method, a (strictly positive) fraction π of the permission to **this**.f is transferred from caller to callee, and the caller satisfies the precondition only if it possesses at least this amount of permission to **this**.f at the time of the call.

To keep track of permissions, the formalization of Chalice [12] associates with each method activation a *permission mask*, that is, a map `Mask` from locations to the method activation's permission to that location. We assume here that permission amounts are represented by rational numbers, but other structures are also possible. Initially, the permission mask is empty: that is, 0 for all locations.

The semantics of read and write statements in Chalice include proof obligations that the current method's mask contains the necessary permissions. So for a read access to `e.f`, the verifier checks $\texttt{Mask}[[\![\texttt{e}]\!],\texttt{f}] > 0$, and for a write access it checks `Mask[⟦e⟧,f]` = 1, where ⟦e⟧ represents the translation of a Chalice expression into an appropriate Boogie expression.

Permission Transfer. 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, and vice versa for the postcondition.

The inhale and exhale operations are defined recursively over the syntax of specifications. Each sub-expression is encoded as a sequence of Boogie statements, which are composed sequentially. Conditions in a specification (such as implications) are encoded as conditional statements.

For an expression `e` not containing accessibility predicates, inhaling `e` means **assume** ⟦e⟧, where **assume** indicates a condition that the verifier is allowed to assume, and exhaling `e` means **assert** ⟦e⟧, where **assert** indicates a proof obligation. Inhaling an accessibility predicate **acc**`(e.f, p)` adds the permission amount `p` to the permission mask:

```
Mask[⟦e⟧, f] := Mask[⟦e⟧, f] + ⟦p⟧;
```

Exhaling **acc**`(e.f, p)` checks that the mask contains at least the permission amount `p` and then removes `p` from the permission mask:

```
assert Mask[⟦e⟧, f] ≥ ⟦p⟧;
Mask[⟦e⟧, f] := Mask[⟦e⟧, f] - ⟦p⟧;
```

In addition, exhaling an expression assigns an arbitrary value to those *heap* locations for which the mask contains no permission after the exhale. This "havoc" operation models possible state updates by other method activations, including those in different threads. It says that if a memory location `e.f` is no longer readable by the current method activation, then any knowledge about its value is removed. In contrast, if the current method activation still holds a non-zero permission to `e.f` after the exhale, then no other method activation can have the full (write) permission to `e.f`, and so whatever the current method activation already knows about the value `e.f` can be soundly retained; that is, can be framed.

Because of the recursive definition of exhale, exhaling an expression such as **acc**`(e.f, p) &&` **acc**`(e.f, p)` will exhale **acc**`(e.f, p)` *twice*; the expression is essentially equivalent to **acc**`(e.f, 2p)` (the conjunction behaves *multiplicatively* with respect to permissions [18]). This conjunction can be formally related to the separating conjunction of separation logic [18].

4 Abstract Read Permissions

In this paper, we propose *abstract read permissions*, which allow the programmer to reason abstractly at the level of read and write permissions rather than concrete fractions. In this section, we introduce the main ideas behind abstract read permissions in method specifications. We discuss details of the encoding in Sec. 6 and extensions to other forms of specifications in Sec. 8.

We use two main kinds of accessibility predicates. A full permission (corresponding to a "1"-valued fractional permission) to a location `e.f` is denoted by **`acc`**`(e.f, 1)` and allows a method to both read and write the location. To specify read permissions, we introduce an accessibility predicate of the form **`acc`**`(e.f,` **`rd`**`)`, called an **`rd`**-*predicate*. The *abstract permission expression* **`rd`** denotes a positive amount of permission to `e.f` and, thus, permits read access, regardless of what the actual amount is. Every occurrence of **`rd`** in the specification of *one* method invocation denotes the same permission amount (regardless of the location mentioned), but the particular amount may be different for other method invocations. In particular, different activations of a recursive method may interpret **`rd`** differently.

Compared with the options mentioned in Sec. 2, our abstract permission expression **`rd`** is most similar to a ghost parameter that is passed to the method. But instead of the programmer having to compute the amount to be passed in, our approach handles the specification of the amount automatically. Intuitively, the positive amount chosen is small enough for the caller to handle (that is, it stays within a budget of the permissions that the caller has) and small enough that it does not completely rob the caller of all permissions to a memory location (thus enabling framing). The program verifier will produce an error if such a value does not exist.

Note that the exact amount chosen for a call is never revealed in our permission model. Indeed, as we shall see, we encode the amount as a symbolic value that satisfies certain constraints. The main challenge in our design is to identify where and how we should constrain the amount for an **`rd`**-predicate.

Example. Methods frequently require some permission to a location `e.f` and return this permission to their callers. That is, both the pre- and postcondition mention some **`rd`**-predicate such as in method `foo` in Listing 5. Because both occurrences of **`rd`** in the specification of `foo` denote the same amount of permission, method `main` is able to recombine this permission to obtain full permission again (as required by the assignment that follows the call). This part of the example motivates our design decision that every occurrence of **`rd`** in one method specification is interpreted as the same permission amount. Furthermore, the permission amount automatically chosen for the call to `foo` is strictly less than what the caller has. Therefore, the caller retains some permission to `c.val` across the call, which enables framing. In our example, this implies that the value of `c.val` cannot be changed by the call to `foo`; thus, the right-hand side of the last assignment to `c.val` evaluates to 5 (as required by the postcondition of `main`).

4.1 Method Implementations

For the verification of each *method implementation*, we introduce a new *permission constant* π_{method}, which is used to interpret every **`rd`**-predicate in the specification of that method for every location. No precise value is given to π_{method}; we

```
method main(c: Cell)
  requires acc(c.val, 1)
  ensures acc(c.val, 1) && c.val = 5
{
  c.val := 0
  call foo(c)
  c.val := c.val + 5
}
method foo(c: Cell)
  requires acc(c.val, rd)
  ensures acc(c.val, rd)
{ /* ... */ }
```

Listing 5. A simple example that illustrates the choice to interpret all **rd**-predicates as the same fraction in a method specification.

assume only that it is a proper read permission: $0 < \pi_{method} < 1$. The method is then verified as usual in Chalice: we inhale the precondition, execute the method body, and then exhale the postcondition. Because the assumption about π_{method} is so weak, a successful verification of the method implementation accommodates any permission amount between 0 and 1 chosen for the call and transferred by the caller. Of course, as usual, only those pre-states and parameter values — and now also the value of **rd** — that can feasibly satisfy the precondition need to be considered when verifying the implementation. For example, for a precondition `acc(e.f, rd) && acc(e.f, rd)`, the precondition can be satisfied only for values of **rd** that are bounded by `.5`.

4.2 Method Calls

A call to a method `m` is verified using `m`'s specification, which may mention **rd**-predicates. Since we verify that the implementation of `m` is correct for any permission amount that one might use to interpret these predicates, the caller is free to choose any fraction between 0 and 1 to interpret them. But we must take care when constraining this choice automatically, because if the constraints are too weak then it will cause callers to fail to verify, and our system would not be practical to use.

If the called method `m` requires an **rd**-predicate to some location `e.f`, we need to check only that the caller holds a positive amount of permission to `e.f`. If it does, intuitively we can always find a (positive) fraction that is smaller than the held amount, and we can transfer this fraction to the callee. This idea is reflected in the encoding of method calls as follows.

We first introduce a permission constant π_{call}, which is used to interpret every **rd**-predicate in the specification of the callee `m` for every location. π_{call} is constrained to be strictly positive (via a Boogie **assume** statement). When exhaling `m`'s precondition, we further constrain π_{call} to be smaller than the permission amount currently held by the caller, for any location for which `m` requires an **rd**-predicate. More precisely, for each **rd**-predicate to be exhaled (that is, each occurrence of **acc(e.f, rd)** for some `e.f`), we first check that the caller has a positive amount of permission to `e.f`, and we constrain π_{call} to be strictly smaller than this positive amount. Next, we subtract π_{call} from the permission mask (which we can do symbolically even if there will be further constraints on the value of π_{call}) and continue exhaling the precondition. This encoding is reflected in the following Boogie code:

```
assert Mask[⟦e⟧, f] > 0;
assume πcall < Mask[⟦e⟧, f];
Mask[⟦e⟧, f] := Mask[⟦e⟧, f] - πcall;
```

The encoding ensures that the callee `m` is provided with the required read permission 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; that is, it enables framing.

Note that our design interprets all **rd**-predicates in a method specification as the *same* permission amount, regardless of the corresponding memory location. This is not a restriction, because the amount is always implicitly chosen to be smaller than any corresponding amount held by the caller. Also, note that if the specification mentions multiple **rd**-predicates to the same location, then we effectively choose an amount that is small enough such that giving away all of those **rd**-predicates is allowed. This is achieved by constraining π_{call} multiple times, once for every **rd**-predicate. The main soundness argument shows that the generated constraints are satisfiable: see Sec. 7.

After exhaling the callee's precondition, our encoding of a call inhales the postcondition, using the same value π_{call} to interpret any **rd**-predicates mentioned. This allows the caller to regain the same amount of permission that it gave away, if **rd** is mentioned in both the pre- and postcondition. So, in the example from Listing 5, method `main` regains write permission to `c.val` after the call to `foo` and, thus, may write to the field.

Example. To illustrate abstract read permissions, we revisit method `eval` in Listing 1, with the placeholder π replaced by the abstract permission expression **rd**. When the method body of `eval` in Listing 2 is verified, the permission to `s.map` starts out as π_{method} after inhaling the precondition. The recursive call to `left.eval` succeeds by exhaling a strictly smaller fraction, which is then regained on inhaling `eval`'s postcondition. Analogously, a second fraction is given away and regained for the second recursive call. Thus, the permission to `s.map` at the end of the method is again π_{method}, which is required to successfully exhale the postcondition.

This example illustrates that abstract read permissions do not require the overhead of the ghost-parameter option in Sec. 2; a programmer neither has to declare ghost parameters nor provide concrete values for ghost parameters when a method is called. Moreover, they address the first three of the four shortcomings of concrete fractions and counting permissions that we discussed in Sec. 2: (1) Since the permission amount chosen is context-dependent, it is adjusted for each call, which lends itself to flexible re-use. (2) The specification expresses only which read and write permissions are requested, but not any concrete permission amounts. Therefore, they do not contain any irrelevant information. (3) For a call, **`rd`**-predicates are constrained to be strictly smaller than the permission held by the caller, which allows framing. In particular, `s.map` $\neq$ **`null`** is still known to hold after the first recursive call because the caller retains some permission to `s.map` during the call, which allows the verifier to prove the precondition of the second call. Abstract read permissions also address shortcoming (4), as we discuss next.

4.3 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, but with the inhale and exhale separated: when a thread is forked to execute a method, the method's precondition is exhaled; at the time a forked thread is joined, the postcondition of the corresponding method (which in Chalice is determined by the type of the token) is inhaled.

For the same reasons as for synchronous method calls, it is useful for asynchronous calls to interpret all **`rd`**-predicates in a method specification as 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 record 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 the **`fork`**/**`join`** statements for a token 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 Sec. 5, we show how to avoid this loss of information through additional specifications.

Example. Let us again consider method `eval` from Listing 1, with the placeholder π replaced by the abstract permission expression **`rd`**, and the parallel implementation from Listing 3. The verification of the parallel implementation is performed as follows. First, the precondition of `Add.eval` is inhaled, adding π_{method} to the mask for the location `s.map`. Then, at the first **`fork`** statement, our encoding introduces a fresh constant $\pi_{fork1} > 0$ to interpret all **`rd`**-predicates in the specification of `eval` for the first **`fork`** statement. When exhaling the precondition of the forked method, permission for `s.map` is (successfully) checked to be positive, and π_{fork1} is constrained to be strictly smaller than the currently held amount. Consequently, when π_{fork1} is transferred to the new thread, the forking thread still holds a positive permission amount for `s.map`. The verification of the second **fork** statement is analogous, with another constant $\pi_{fork2} > 0$. So after the two **fork** statements, the forking method is left with $\pi_{method} - \pi_{fork1} - \pi_{fork2}$. When inhaling the postcondition at the two **join** statements, the fractions π_{fork1} and π_{fork1} are regained; the verifier knows that these amounts are the same as for the two **fork** statements, since the amounts were recorded in a ghost field of the two tokens. Consequently, the forking method `Add.eval` now holds π_{method}, which allows it to exhale its postcondition.

This example illustrates that abstract read permissions enable flexible splitting of permissions. Since permission amounts are constrained to be strictly smaller than the amounts held by the current method activation, abstract read permissions even support *unbounded* permission splitting. That is, they also address the final shortcoming (4) of concrete fractions and counting permissions discussed in Sec. 2. Note that, in contrast to the ghost-parameter option from Sec. 2, programmers do not have to devise a strategy to determine how to split permissions (for instance by splitting the currently held permission in half whenever a fraction needs to be transferred) and how to re-combine the permissions, which is tricky when the order in which the permissions are regained is not statically known.

4.4 Losing Permission

Using the same permission amount to interpret all **rd**-predicates in a method specification is useful for most implementations. However, this can be too restrictive when a method `m` gives away some permission to a location (for instance, during a **fork**) and returns what is left. To handle these situations, we introduce an alternative abstract read expression **rd***, which gets interpreted as another positive, but otherwise unrelated permission amount. In particular, there is no guarantee that it corresponds to the same amount as any other permission expression used in the program. So we could specify `m` to require an **rd**-predicate and to ensure an **rd***-predicate.

When inhaling an **rd***-predicate, no information is assumed about the permission amount it denotes, other than it being positive. Therefore, we can exhale an **rd***-predicate by checking that the current method activation has *some* permission to the appropriate location, and then interpreting the **rd***-predicate as a strictly smaller amount.

5 Permission Expressions

Our design so far provides only two ways of specifying read permissions, **rd**-predicates and **rd***-predicates. The resulting expressiveness is insufficient for some interesting examples. Consider for instance a method `m` that requires full permission to some location, transfers an **rd**-predicate to a newly forked thread `tk`, and returns the remaining permission (along with the token `tk`) to its caller.

So far, we can specify that `m` returns *some* permission to the caller using an **`rd*`**-predicate, but we have no way of denoting the precise permission amount it returns (the *difference* between two permission amounts). However, the precise information is necessary for the caller to regain full permission by joining `tk`.

To provide sufficient expressiveness for such examples, we generalise the accessibility predicates `acc(e.f, 1)` and `acc(e.f, rd)` to the new form `acc(e.f,` p`)`, where p is a *permission expression*. Permission expressions p are defined inductively as follows:

$$\begin{array}{lll} p ::= & \mathtt{c} & \textit{concrete fraction} \\ \mid & \mathbf{rd} & \textit{abstract read permission} \\ \mid & \mathbf{rd}\mathtt{(tk)} & \textit{token read permission} \\ \mid & p_1 + p_2 & \textit{permission addition} \\ \mid & p_1 - p_2 & \textit{permission subtraction} \\ \mid & \mathtt{n} * p & \textit{permission multiplication} \end{array}$$

where `c` is a rational literal ($0 < \mathtt{c} \leq 1$), `tk` is a token, and `n` is an integer-valued expression

The literal permission expression `c` subsumes full permissions, and **`rd`** is used as before. The expression `rd(tk)` refers to the amount of permission associated with an **`rd`**-expression for a particular asynchronous call, via its token. Finally, we support addition, subtraction, and integer multiplication of permission expressions. This allows us in particular to specify the exact permission amount returned by method `m` above, using the permission expression `1 - rd(tk)` in its postcondition.

In addition to the generalised accessibility predicates, we continue to support **`rd*`**-predicates. However, we decided not to support permission expressions containing **`rd*`** because the meaning of such expressions is sometimes un-intuitive (for instance, `1 - rd* + rd*` does not necessarily denote a full permission since the two occurrences of **`rd*`** may be interpreted differently) and because they do not provide extra expressiveness (for instance, `1 - rd*` and **`rd*`** express the same information, namely an unknown, positive permission amount).

6 Encoding

The introduction of permission expressions leads to new subtleties in the encoding of abstract read permissions. This section revises the encoding sketched in Sec. 4 to address these subtleties.

First, because permission expressions p can include subtraction and multiplication, it is possible to write expressions such as `rd-1` that 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, leading to unsoundness. Therefore, we impose an additional well-formedness constraint that each permission expression p must provably denote a strictly positive amount of permission. The exact implementation of this check varies for different kinds of permission expression, as shown later in this section.

```
method test(c: Cell)
  requires acc(c.f)
  ensures acc(c.f, rd*) && c.f == 3
{
  c.f := 3;
  call bar(c);
}

method bar(c: Cell)
  requires acc(c.f, 1-rd) && acc(c.f, rd)
{
  c.f := 4;
}
```

Listing 6. Occurrences of **rd** in negative positions need to be treated with care to avoid inconsistencies.

Second, permission expressions require more sophisticated rules for constraining the permission constants π_{call} during exhale operations. So far, exhaling an **rd**-predicate led to an upper bound on π_{call} by assuming that π_{call} is *smaller* than the positive amount currently stored in the mask for a particular location. With the introduction of permission expressions, the abstract permission expression **rd** can also occur in *negative* positions, for instance in `acc(e.f, 1-rd)`. If we were to adopt the same behaviour when exhaling accessibility predicates with **rd** in negative positions, then we also impose lower bounds, leading to potentially unsatisfiable assumptions.

Exhaling negative occurrences of **rd** may also lead to difficulties with subsequent *positive* occurrences, as the example in Listing 6 shows. At the call to `bar` in the body of `test`, the mask contains full permission to `c.f`. Exhaling the first conjunct of `bar`'s precondition leaves us with exactly π_{call}. When the second conjunct `acc(c.f, rd)` is exhaled, the encoding from Sec. 4 checks that some permission is available and then assumes π_{call} to be strictly smaller than that amount. This would lead to the inconsistent assumption $\pi_{call} < \pi_{call}$.

In our solution to these difficulties, we differentiate between three different types of permission expressions:

Type 1: Those in which **rd** does not occur (e.g., `1-rd(tk)`).
Type 2: Those in which **rd** occurs, but only in positive positions (e.g., `rd-rd(tk)`).
Type 3: Those in which **rd** occurs in negative position(s) (e.g., `1-rd`).

We classify accessibility predicates `acc(e.f, p)` in the same way, according to the type of `p`. The special **rd***-predicates are handled like type-2 accessibility predicates, but with respect to a fresh permission constant for each occurrence.

We now describe how to encode the exhale of a precondition at a (synchronous or asynchronous) method call, in terms of how we generate appropriate Boogie code. Inhales are encoded as described earlier; no constraints on π_{call} are generated. To encode a method call, we first introduce a fresh permission constant π_{call} and constrain it to denote a proper read permission:

```
havoc $\pi_{call}$; assume $0 < \pi_{call} < 1$;
```

If the method call is asynchronous, we additionally store π_{call} in a ghost field of the corresponding token. (Since the value of this ghost field never changes, no permission management is necessary for that field.) Then, we exhale the precondition in three phases, each handling one type of accessibility predicate.

Phase 1: The first phase handles logical expressions without accessibility predicates and those accessibility predicates that denote permission amounts that were already fixed before the call; that is, predicates of type 1. To do this, we pass over the precondition, generating assertions for all logical expressions, and ignoring all accessibility predicates of types 2 and 3. For each accessibility predicate `acc(e.f, p)` of type 1, we encode the exhale as described in Sec. 3; that is, by generating the following code:

```
assert ⟦p⟧ > 0;
assert Mask[⟦e⟧, f] ≥ ⟦p⟧;
Mask[⟦e⟧, f] := Mask[⟦e⟧, f] - ⟦p⟧;
```

Phase 2: The second phase generates constraints on π_{call} that ensure that accessibility predicates of type 2 can be exhaled (if possible), leaving some remainder. To do this, we pass over the precondition again, ignoring logical expressions and accessibility predicates of types 1 and 3. For each accessibility predicate `acc(e.f, p)` of type 2, we assume a rewriting of the form `p = p' + n * rd` (for $n > 0$) such that `p'` is some permission expression not mentioning **rd**. We then generate code that constrains the value of π_{call}:

```
assert ⟦p'⟧ ≥ 0;
assert Mask[⟦e⟧, f] > ⟦p'⟧;
assume n * $\pi_{call}$ < (Mask[⟦e⟧, f] - ⟦p'⟧);
Mask[⟦e⟧, f] := Mask[⟦e⟧, f] - (⟦p'⟧ + n * $\pi_{call}$);
```

Phase 3: The third phase exhales the remaining accessibility predicates, without introducing further constraints on π_{call}. To do this, we pass over the precondition a third time, ignoring logical expressions and accessibility predicates of types 1 and 2. For each accessibility predicate of type 3, we generate the same code as in Phase 1.

Constraining π_{call} after accessibility predicates of type 1 have been exhaled results in stronger assumptions, since we assume the value of π_{call} is smaller than the amounts held after Phase 1 is finished. A permission expression of type 2 comes with the requirement that the part that does not mention **rd** (the `p'` above) is non-negative; thus, the whole expression denotes a positive amount. Exhaling accessibility predicates of type 3 only after all constraints have been generated in Phase 2 solves the problems mentioned at the beginning of this section: exhaling **rd** in negative positions does not generate any constraints and, thus, introduces no lower bounds on π_{call}. Moreover, the precondition of method `bar` (Listing 6) is now exhaled soundly; we first exhale the second conjunct (in Phase 2), which constrains π_{call} and leaves $1 - \pi_{call}$ in the mask, and then exhale the first conjunct (in Phase 3). This does not lead to additional constraints and, thus, does not introduce inconsistent assumptions.

Any conditionals in the precondition are handled in each phase. However, the evaluation of such conditionals are unaffected by our manipulation of permissions, because conditionals in assertions are syntactically restricted not to depend on permissions (assertions such as **acc**(x.f) ⇒ **acc**(y.f) are forbidden).

After all phases are complete, our encoding removes all knowledge about locations to which no permission remains in the mask, as we explained in Sec. 3.

7 Soundness

In this section, we give a brief argument for the soundness of our encoding. A soundness proof for an entire verification methodology using our permission model is beyond the scope of our paper, and for the most part involves arguments that are orthogonal to the contributions of this paper.

Compared to the standard fractional permission model, we introduced abstract read permissions and encode them as underspecified constants in Boogie. The most relevant concern for soundness with respect to this paper is that the assumptions that we introduce about the constants used to interpret abstract permission expressions must not lead to contradictions. Apart from the points in our encoding where these assumptions are generated, abstract read permissions are treated just as any other permission amounts in permission expressions.

A new constant π_{call} to denote the underspecified amount is introduced at each method call in the encoding of the source program. From the end of Phase 2 (as described in the previous section), these amounts are treated just like any other permission amount with a fixed interpretation. Therefore, it is sufficient for us to justify that, for each method call, the assumptions generated in Phase 2 are always satisfiable, provided that none of the assertions in the generated code fail. Operationally (though this is never required in the verifier), one can think of this amount as being chosen (by some oracle) at the point of the method call, in such a way that the assumptions are all satisfied; we just need to justify that this is possible. As we show below, each assumption during Phase 2 imposes a (strictly positive) upper bound on the possible values of π_{call}. Since the only lower bound imposed is 0 and since we assume that permission amounts are rational numbers, the assumptions are then guaranteed to be satisfiable. In the following, we focus on the permission expressions presented in Sec. 5, but the arguments apply equally to **rd***-predicates.

Let us consider the exhale of the method precondition for any given method call, and let π_{call} be the permission constant introduced for that call. Let us further consider the amount of permission (for any location) stored in the `Mask` up to the start of Phase 2 of exhaling the precondition as a formula representing the arithmetic performed so far during the verification. We first apply an inductive argument that during Phase 2 of the exhale, this formula remains expressible in the form $\rho - \mathtt{m} * \pi_{call}$, for some formula ρ not mentioning π_{call} and some integer $\mathtt{m} \geq 0$: since π_{call} is chosen to be a fresh constant for each call, it is clear that up to the start of Phase 2, the formula representing the current permission amount in the mask for any location cannot depend on the fresh π_{call} constant. Each exhale of an accessibility predicate during Phase 2 subtracts multiples of π_{call} from the mask, and so the amount stored remains expressible in the form $\rho - \mathtt{m} * \pi_{call}$.

Now consider the handling of an arbitrary type-2 accessibility predicate **`acc`**`(e.f, p)` during Phase 2. Just as in the encoding presented in the previous section, we assume a rewriting of the form `p = p' + n *` **`rd`** (for some $\mathtt{n} > 0$) such that `p'` is some permission expression not mentioning **`rd`**. Using the argument so far, we may assume that there exist an integer $\mathtt{m} \geq 0$ and a formula ρ not mentioning π_{call}, such that:

$$\mathtt{Mask[}[\![\mathtt{e}]\!]\mathtt{,\ f]} = \rho - \mathtt{m} * \pi_{call}$$

We can now use this equality to rewrite the relevant code generated in Phase 2:

```
assert Mask[⟦e⟧, f] > ⟦p'⟧;
assume n * π_call < (Mask[⟦e⟧, f] - ⟦p'⟧);
Mask[⟦e⟧, f] := Mask[⟦e⟧, f] - (⟦p'⟧ + n * π_call);
```

into the following (equivalent) form:

```
assert ρ - m * π_call > ⟦p'⟧;
assume π_call < (ρ - ⟦p'⟧) / (m + n);
Mask[⟦e⟧, f] := ρ - ⟦p'⟧ - (m + n) * π_call;
```

Since `m` and π_{call} are non-negative, the assertion implies that $(\rho - [\![\mathtt{p'}]\!]) > 0$. Combined with the facts $\mathtt{m} \geq 0$ and $\mathtt{n} > 0$, this gives us that $(\rho - [\![\mathtt{p'}]\!])/(\mathtt{m} + \mathtt{n})$ is strictly positive. Therefore, the assumption only imposes an extra strictly positive upper bound on the permitted values of π_{call}. Since this argument applies to each accessibility predicate generated in Phase 2, all assumptions generated impose strictly positive upper bounds on π_{call}, and thus, combined with the only other assumptions about this value, that $0 < \pi_{call} < 1$, the assumptions about π_{call} are always satisfiable.

8 Monitors

Chalice supports monitors, which have an associated *monitor invariant*, describing the permissions held and properties guaranteed while the monitor is unlocked. When a monitor is acquired, the monitor invariant is inhaled, and when the monitor is released, it is exhaled [12]. It can be useful for a monitor invariant to provide read permissions to the fields it describes. A typical example is a single-writer, multiple-reader scenario, which can be handled by splitting a full permission between the writer thread and the monitor. By acquiring the monitor, a thread can obtain read permission. The writer can combine the fraction from the monitor with the fraction it already holds to obtain full permission.

Supporting abstract read permissions for monitor invariants is more difficult than for methods. If we allowed a thread to choose a permission amount for **rd**-expressions when exhaling the monitor invariant upon release (analogously to choosing the amount when exhaling a method precondition) then one could not soundly assume that the next acquire in the same thread will inhale the same amount — other threads might have acquired and released the monitor in the meantime and interpreted the **rd**-expressions in the monitor invariant differently. Similar issues arise with folding and unfolding abstract predicates [17] as well as with sending and receiving messages [14].

If the threads interacting with a monitor never need to know that the amount of read permission that they inhale from a monitor invariant is related to some other amount earlier in the program execution, then we can employ **rd**∗-predicates in the monitor invariant and handle them just as for method calls.

However, in situations like the single-writer, multiple-reader example above, we must associate a persistent amount of permission with a monitor invariant to allow the writer thread to obtain full permission. We have considered various solutions to this problem. One is to fix the permission amount that **rd**-expressions are interpreted with "once and for all", either for all monitors (which has the advantage that such permissions can be transferred between monitors), or when a new monitor is created (which has the advantage that the amount can be chosen with respect to what is held at that point). An alternative is to store the permission amount in a ghost field of the object, similarly to the ghost-parameter option in Sec. 2. This is more flexible, but permissions to this ghost field must then be appropriately handled, and information about the field value needs to be communicated in specifications for this to really give an advantage.

In our implementation, we currently take the simplest approach described above; we generate one (underspecified) permission constant to interpret all **rd**-expressions in monitor invariants, abstract predicates, and message invariants. This has been sufficiently expressive, but we are also evaluating the other options.

9 Related Work

Fractional permissions were proposed by Boyland [4]. He uses them in a type system to check non-interference of the branches of a parallel composition and to show that non-interfering parallel compositions have deterministic results. Zhao [21] uses fractional permissions to prevent data races in concurrent Java programs. Neither of the two systems supports the verification of a program w.r.t. to a programmer-supplied specification.

The use of fractional permissions for program verification was first explored in the context of separation logic. Bornat *et al.* [3], Gotsman *et al.* [7], and Hobor *et al.* [9] all employ separation logic with fractional permissions. They use concrete fractions and logical variables in specifications, which has the drawbacks discussed in Sec. 2.

Our original verification methodology for Chalice [12] supports fractional permissions (expressed as integer percentages) and infinitesimal permissions, which are similar to counting permissions. We built on implicit dynamic frames [20], which allows us to generate first-order verification conditions, which can be handled by automatic SMT solvers. The permission model used in this work suffers from the shortcomings discussed in Sec. 2. To solve these problems, we introduced the idea of underspecified, constrained fractions in a workshop paper [8]. The present paper extends our earlier work with more detailed explanations (especially of the encoding) and with a soundness argument.

Other systems have also aimed to package fractions in more abstract ways. In his original work on fractional permissions, Boyland uses a type system that allows permission polymorphism [4]. A non-deterministic type checker determines the possible ways fraction variables used in method signatures can be instantiated. Only a sketch of an algorithm for coming up with the instantiation is provided; it is not clear how the sketched approach deals with repeated fraction variables or with fraction variables occurring in negative positions.

The separation-logic based verifier VeriFast supports fractional permissions and logical variables [10]. When a logical variable specifies a permission amount, there is some limited support to set it automatically at a call site. However, only the first use of the variable is considered (so fractions cannot be correlated) and that first use will soak up all the permission that the caller has (so framing is not supported).

Bierhoff *et al.* [2, 16] recently presented fraction-free permission type systems. Their permissions, which have intuitive names such as "unique" and "local immutable", are held by variables, whereas our permissions are held by method activation records (and monitors, etc.). Permission transfers happen at assignments and parameter passing. If the target of the transfer only needs a fraction, the type system automatically carves up the permission held. Permissions that cross method boundaries can be *borrowed* (meaning they will be transferred back upon return of the method) or *consumed.* Borrowing is related to our rule of interpreting all **`rd`**-expressions in one method specification as the same permission amount, while consuming is related to our **`rd*`**-predicates.

Bierhoff [2] presents a verification system that makes use of permissions as a subsequent step after permission type checking. The integration of both steps in our system provides more expressiveness, for instance, because permissions can be denoted conditionally, using logical implication.

Recent work of Militão et al. [15] generalizes fractional permissions to user-defined *views*, which can describe permissions to sets of fields, and properties such as reference uniqueness. They also employ fractions whose values are hidden from the user, but do not have an analogue to permission expressions (cf. Sec. 5).

There are other ways of specifying that a method will treat something as read-only. The C verifier VCC employs *claims* [5], which can specify a certain set of objects which cannot be modified while the claim exists. Using reference counting, an object keeps track of how many outstanding claims it has. Claims are themselves (ghost) objects, so there can be claims on claims. This allows programs like our `eval` example in Sec. 2 to be specified and verified, but at the cost of having to write the (ghost) code that sets up and destroys the claims.

10 Conclusions

We have presented a novel methodology for specifying sequential and concurrent programs based on fractional permissions. Our abstract read permissions allow programmers to specify access permissions at the level of read and write permissions, without the need to reason using a concrete mathematical model or syntax. Our methodology avoids shortcomings of working with concrete fractions and, by picking a judicious interpretation for abstract read permissions in method specifications, imposes less specification and verification overhead than solutions based on ghost parameters or logical variables. In cases where the differences between permission amounts are important, our permission expressions provide a natural and abstract way of conveying the relevant information.

We have explained our methodology in terms of implicit dynamic frames and verification condition generation. However, abstract read permissions are independent of any particular permission logic or verification technique. So far, our methodology has been implemented in two verifiers for Chalice: one based on verification condition generation and one on symbolic execution. Both verifiers make use of the support for real numbers in Z3 [6].

Our permission expressions support the addition and subtraction of a bounded number of **rd**-expressions. As future work, we plan to handle the (statically) unbounded case, for instance, by supporting mathematical sums over unbounded sets or sequences. We could then specify a method that forks an unbounded number of threads (each requiring read permission to some shared data) and stores the tokens in a list. By removing tokens from the list, we could easily support a specification for rejoining the threads, regardless of the order of joins. This kind of example would be difficult to support with concrete fractional permissions.

We are also exploring other possible uses of abstract read permissions; exploiting the use of permission amounts which can be freely constrained (from above). We are considering the possibility of manually introducing such amounts in other verification situations, and also basing an approach to handling immutable data on this novel specification concept.

Acknowledgements

We would like to thank the attendees and reviewers of the Formal Techniques for Java-like Programs 2011 workshop, as well as 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. We thank John Boyland for discussions of the comparisons with fractional permissions. We are grateful to Malte Schwerhoff for many discussions about the details of our model and to Martin Vechev for useful feedback on a draft of this paper.

References

1. M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *FMCO*, volume 4111 of *LNCS*, pages 364–387. Springer, Sept. 2006.
2. K. Bierhoff. Automated program verification made SYMPLAR: symbolic permissions for lightweight automated reasoning. In *ONWARD*, pages 19–32. ACM, 2011.
3. R. Bornat, C. Calcagno, P. O'Hearn, and M. Parkinson. Permission accounting in separation logic. In *POPL*, pages 259–270. ACM, 2005.
4. J. Boyland. Checking interference with fractional permissions. In *SAS*, volume 2694 of *LNCS*, pages 55–72. Springer, 2003.
5. E. Cohen, M. Moskal, W. Schulte, and S. Tobies. Local verification of global invariants in concurrent programs. In *CAV 2010*, volume 6174 of *LNCS*, pages 480–494. Springer, 2010.
6. L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.
7. A. Gotsman, J. Berdine, B. Cook, N. Rinetzky, and M. Sagiv. Local reasoning for storable locks and threads. In *APLAS*, volume 4807 of *LNCS*, pages 19–37. Springer, 2007.
8. 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.
9. A. Hobor, A. W. Appel, and F. Z. Nardelli. Oracle semantics for concurrent separation logic. In *ESOP*, volume 4960 of *LNCS*, pages 353–367. Springer, 2008.
10. B. Jacobs, J. Smans, and F. Piessens. A quick tour of the VeriFast program verifier. In *APLAS 2010*, volume 6461 of *LNCS*, pages 304–311. Springer, 2010.
11. I. T. Kassios, P. Müller, and M. Schwerhoff. Comparing verification condition generation with symbolic execution: an experience report. In *VSTTE*, volume 7152 of *LNCS*, pages 196–208. Springer, 2012.
12. K. R. M. Leino and P. Müller. A basis for verifying multi-threaded programs. In *ESOP*, volume 5502 of *LNCS*, pages 378–393. Springer, 2009.
13. K. R. M. Leino, P. Müller, and J. Smans. Verification of concurrent programs with Chalice. In *FOSAD Lectures*, volume 5705 of *LNCS*, pages 195–222. Springer, 2009.
14. K. R. M. Leino, P. Müller, and J. Smans. Deadlock-free channels and locks. In *ESOP*, volume 6012 of *LNCS*, pages 407–426. Springer, 2010.
15. F. Militão, J. Aldrich, and L. Caires. Aliasing control with view-based typestate. In *FTfJP*, pages 7:1–7:7. ACM, 2010.
16. K. Naden, R. Bocchino, J. Aldrich, and K. Bierhoff. A type system for borrowing permissions. In *POPL*, pages 557–570. ACM, 2012.
17. M. Parkinson and G. Bierman. Separation logic and abstraction. In *POPL*. ACM, 2005.
18. M. J. Parkinson and A. J. Summers. The relationship between separation logic and implicit dynamic frames. *Logical Methods in Computer Science*, 2012. To appear.
19. J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*. IEEE, 2002.
20. J. Smans, B. Jacobs, and F. Piessens. Implicit dynamic frames: Combining dynamic frames and separation logic. In *ECOOP*, pages 148–172. Springer, 2009.
21. Y. Zhao. *Concurrency Analysis based on Fractional Permission System.* PhD thesis, The University of Wisconsin–Milwaukee, 2007.