Constraint Semantics for Abstract Read Permissions

John Tang Boyland[*][†]
boyland@uwm.edu

Peter Müller[†]
peter.mueller@inf.ethz.ch

Malte Schwerhoff[†]
malte.schwerhoff@inf.ethz.ch

Alexander J. Summers[†]
alexander.summers@inf.ethz.ch

ABSTRACT

The concept of controlling access to mutable shared data via permissions is at the heart of permission logics such as separation logic and implicit dynamic frames, and is also used in type systems, for instance, to give a semantics to "read-only" annotations. Existing permission models have different strengths in terms of expressiveness. Fractional permissions, for example, enable unbounded (recursive) splitting, whereas counting permissions enable unbounded subtraction of the same permission amount. Combining these strengths in a single permission model appeared to increase the complexity for the user and tools. In this paper we extend our previous work on abstract read permissions by providing them with a novel constraint semantics, which retains the use of the domain of rational numbers but enables unbounded subtraction of identical amounts. Thus we can keep an intuitive model conducive to SMT solvers while enabling "counting."

Categories and Subject Descriptors

F.3.1 [**Logics and Meaning of Programs**]: Specifying and Verifying and Reasoning about Programs—*mechanical verification,specification techniques*

General Terms

Verification

1. INTRODUCTION

Access permissions enable sound, modular reasoning about shared mutable state. Program logics based on permissions, such as separation logic [12] and implicit dynamic frames [13], associate a permission with each memory location. A thread may access a shared location only if it has the corresponding permission. This rule prevents data races since at most one thread can hold permission to any location, and enables framing since no thread can modify a location while another thread holds permission to it. Many permission systems distinguish between read and write permissions to enable concurrent reads while enforcing exclusive writes. These systems allow a write (or *full*) permission to be split into several read permissions, which can later be re-combined into a write permission.

To be useful for automatic program verification, a permission system must have three key properties. First, the underlying permission model must be sufficiently *expressive*. Second, the permission system should require *low annotation overhead*, both in terms of complexity and verbosity of the permission assertions. Third, the permission model should be *amenable to automatic provers*, especially SMT solvers.

Bornat and others [1] identify two criteria for expressive permission models: Some programs require *unbounded divisibility* (or "infinite splitting"), for instance, when threads are forked recursively, and all sub-threads need read permission to a shared location. Other programs require *unbounded counting*, for instance, when one thread forks off an unbounded number of threads each with an identical permission and then waits for them to finish, in arbitrary order. Specification techniques that express data abstraction via abstract predicates [11] also require support for *multiplication* if one wishes to scale arbitrary predicates by fractions.

To our knowledge, no existing implementation of a permission system satisfies all of these requirements. For instance, fractional permissions [2] support unbounded divisibility and multiplication since one can use rational numbers (or real numbers) as a model, which also leads to low annotation overhead and good support from SMT solvers. However, unbounded counting seems impossible since if one starts with a write permission (fraction 1), then however small a positive fraction $q > 0$ one chooses to give to each sub-thread, there always is a point n after which $1 - nq$ is no longer positive.

Counting permissions [1] support unbounded counting by splitting a permission into an unbounded number of units and the remainder. The system then tracks how many units a thread holds (or how many it lacks for a full permission). However, counting permissions support neither unbounded divisibility (because units cannot be divided further), nor multiplication.

It is possible to compound fractional and counting models [1], for example, by representing permissions as a fraction plus a positive or negative number of units [10]. Dockins et al. [4] achieve this combination with a tree model for per-

[*]Dept. of EECS, University of Wisconsin, Milwaukee, USA
[†]Dept. of Computer Science, ETH Zurich, Switzerland

FTfJP '14, July 29 2014, Uppsala, Sweden
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2866-1/14/07...$15.00
http://dx.doi.org/10.1145/2635631.2635847

missions, but multiplication is not supported, and the encoding of counting imposes additional structure (each counting permission is represented differently), which we believe is challenging for an implementation. The implemented decision procedures by Le et al. based on this model [8] do not support counting. In our experience, an implementation of a compound model can lead to many disjunctions in proof obligations (since a thread may read a location if the fraction *or* the unit count is positive), which slows down SMT solving. Boyland [3] proposes $\mathbf{Z}[\epsilon]^+$ (positive polynomials over an infinitesimal) which satisfies all three criteria, but we are unaware of any implementation using this complex and subtle model.

In this paper, we present a permission system that supports unbounded divisibility, unbounded counting, and multiplication. Since its model is based on rational numbers, it is well suited for SMT solvers and offers a simple notation for permission assertions. However, as in our previous work on abstract read permissions (ARPs) [5], most permission assertions do not even have to indicate concrete fractions. It is typically sufficient to specify whether read or write permission is required. The actual fraction for a read permission is neither specified by the programmer nor determined by the verifier. Instead, when a read permission is transferred from a thread T_1 to a thread T_2, its fraction is suitably constrained to ensure that it is positive (to ensure that it permits read access) and that it is strictly less than the permission currently held by T_1 (to ensure that T_1 can still read). Appropriate proof obligations ensure that the constraints generated for an abstract read permission are satisfiable, that is, that there are fractions that could be chosen, although that never actually happens.

Like our earlier work on abstract read permissions, the proposed system supports unbounded divisibility and multiplication (since it is based on rational numbers), but unlike the previous work it also supports unbounded counting. The additional expressiveness is achieved by extending abstract read permissions with a more expressive constraint generation. Intuitively, if we have forked off n threads, each with an identical, not yet specified, fraction k, we have the constraint that $k < \frac{1}{n}$. If we fork off further threads, k is further constrained. But at all times, there are (an infinite number of) possible values for k that satisfy the constraints. In this work, we explain how this idea works in the context of a few illustrative examples, formalize the intuition, and discuss how to implement it in a program verifier.

2. MOTIVATION

The key idea of abstract read permissions (as presented in [5]) is to introduce *symbolic* names to represent read permission amounts (that is, strictly positive amounts of permission which are otherwise unspecified). When introducing a *fresh* symbolic amount, no information is known about its value, other than these bounds. The intuitive idea is that this amount may now be used when giving away permissions (e.g., when forking a new thread) and, whatever amount of permission is currently held (provided it is not already zero), the symbolic amount given away can be assumed to be strictly smaller than what is held, leaving both the caller and the callee with some permission left-over. Thus, constraints can be added *on-the-fly* to these symbolic amounts, which are never concretely chosen, neither at the source level, nor in an implementation such as Chalice; the symbolic amounts (abstract read permissions) can be seen as a kind of *prophecy variable*, always denoting a judiciously-chosen fraction.

Our and others' earlier work [5] employs this idea in a limited sense; extra constraints can be added only for method calls, for which a fresh symbolic amount is introduced and constrained only locally, that is, in the encoding of the call. The reasons for this restriction are elaborated in Sec. 5, but are essentially due to the difficulty of designing an encoding that yields constraints which are guaranteed to be satisfiable. Lifting these restrictions was difficult, since the essential property for soundness was not explained independently of the restrictions in [5]. We improve on this work by cleanly pulling out a property characterising sound constraint systems (Sec. 3), and by showing how this property can be used to lift previously necessary restrictions, which enables further examples to be verified, as shown in the following subsections.

2.1 Language

We use a Java-like language enriched with specification constructs such as method pre- and postconditions and permissions. The `ghost` keyword is used to declare fields, arguments or local variables that are needed only for verification and can be erased at run-time. We use `acc(`$x.g$`, 1)` (or just `acc(`$x.g$`)`) to denote full permissions to field $x.g$, and `acc(`$x.g, f$`)` to denote read access (with f permissions). The semantics of `acc(`$x.g, f$`)` are such, that $0 < f$ is assumed whenever permissions are gained, respectively, asserted when given away. The ghost-type `ARP` is used to declare permission-typed variables that can be constrained on-the-fly, as mentioned above, and `fresh()` is used to assign a fresh symbolic value to such a variable. For brevity, we ignore orthogonal issues such as exceptions, starvation and numerical overflows. Another simplification we allow ourselves is reading final fields without having the corresponding permissions, since such fields are immutable. This assumption ignores issues related to the initialisation of immutable data, but we regard these as orthogonal as well.

2.2 Example 1: Counting

Figure 1 shows a simple multiple-reader single-writer mutual exclusion class (an "RW controller") protecting a mutable cell. One can create an RW controller for a given cell by calling the constructor and transferring write permission to the cell that the controller is meant to protect. Any thread may request reads or writes to the controlled data; these requests block until access is granted and then proceed. Concurrent reads are supported by (temporarily) giving each reader a read permission to the protected data. This intuition is captured in the controller's invariant, which says that the controller holds all permissions to the shared data, minus what has been given to the currently active readers.

The RW controller extends a Java-like non-reentrant `Lock` class, instances of which can be acquired (locked) and released (unlocked). The lock is exclusive and does thus not permit concurrent reads per se, which is therefore provided by the controller. Each lock has a lock invariant that, for simplicity, we require to be established at the end of the lock's constructor. A thread that acquires the lock gets to assume the invariant, and consequently, has to ensure it when it releases the lock.

Method `doWrite` is straight-forward: The thread busily waits for the lock to be available in a state where no reader

```
class Cell { public int val = 0 }

interface Reader {
   void read(Cell data, ghost ARP frac)
     requires acc(data.val, frac)
     ensures acc(data.val, frac)
}

interface Writer {
   void write(Cell data)
     requires acc(data.val)
     ensures acc(data.val)
}


class RWController extends Lock {
  private int rds = 0;     // No. of active readers
  private final Cell data; // Shared data
  private final ghost ARP frac = fresh();

  /* Lock invariant */
  invariant acc(rds) && rds >= 0 &&
            acc(data.val, 1 - rds * frac)

  public RWController(Cell data)
     requires acc(rds) && acc(data.val)
  {
    this.data = data
      // Lock invariant established
  }

  public void doWrite(Writer r)
    requires r != null
  {
    while(true) {
      acquire(this); // Gives acc(data.val, 1 - rds*frac)
      if (rds == 0) break;
      release(this); // Takes acc(data.val, 1 - rds*frac)
    }
    // We have acc(data.val, 1) since rds = 0
    r.write(data); // Takes and returns acc(data.val, 1)
    release(this); // Takes acc(data.val, 1)
  }

  public doRead(Reader r)
     requires r != null
  {
    acquire(this); // Let R = rds at this point
      // Gives acc(data.val, 1 - R * frac)
      // Due to def. of acc, we also get
      //       0 < 1 - R * frac
    rds++;
    release(this);
      // Takes acc(data.val, 1 - (R+1) * frac),
      // Due to def. of acc we have to show
      //          0 < 1 - (R+1) * frac
      //   <=> frac < 1 / (R+1)
      // We add this as a new constraint (assumption)
      // We still have:         acc(data.val, frac)
    r.read(x,frac); // Takes, returns acc(data.val, frac)
    acquire(this);  // Let R' = rds
      // Gives acc(data.val, 1 - R' * frac)
      // Together: acc(data.val, 1 - (R'-1)*frac)
    rds--;
      // rds = R'-1, ready to release
    release(this);
  }
}
```

Figure 1: Reader/Writer Interfaces and Controller. In our examples, access expressions `acc(`$x.g$`, 1)` and `acc(`$x.g, f$`)` imply that x is non-null.

is currently active (`rds == 0`), in which case it holds on to the exclusive lock, writes to the shared data, and finally releases the lock again.

Method `doRead` is more interesting: After locking the controller, and thus gaining the permissions stored in the invariant, the controller increases the number of currently active readers, followed by releasing the lock again. Afterwards, the reader's read operation is performed. The permission-related challenge in this situation is to ensure that the lock invariant can be re-established while still holding on to `frac` of the permission to the shared data `data.val`, which must be passed to `r.read`. This entails checking that we are actually able to subtract yet another `frac` of the permission from the amount stored in the invariant, which we achieve by constraining `frac` accordingly. Intuitively, this constraint is satisfiable because we never require any lower bound for `frac` other than zero, and thus, any arbitrarily small value for `frac` would suffice (we never actually have to make this choice, as will be explained in Sec. 3). When the reader is done, it returns the permissions to the shared data, which are placed back into the lock invariant after having acquired the lock again[1].

Note that the verification of this example requires unbounded permission counting because we want to give each reader the same amount of permissions, which simplifies permission bookkeeping and yields a simpler lock invariant. An alternative based on unbounded splitting would be to give each reader half of the remaining permission. This, however, would require the invariant to include a sum over a (ghost) list of permission amounts, which is less intuitive, and causes additional work both in the specifications and for the SMT solver [9].

2.3 Example 2: Splitting and Counting

The example in Fig. 2 sketches a tree-based concurrent work division framework that parallelises a computation over a tree structure by recursively forking off a new task thread for each node in the tree. In order to perform its computation, each task requires read access to a shared resource. Eventually, the individual results are combined in order to yield the result of the overall computation. The example exercises unbounded permission splitting when recursively going from one level in the tree to the next deeper level, and unbounded permission counting when iterating over the direct children of a node.

We use a Java-like `Thread` class whose instances can be forked (`start`) and later on joined (`join`) again. Method `run` must be implemented by clients and is invoked when the thread starts. Hence, `start` requires `run`'s precondition, and `join` ensures `run`'s postcondition. We regard issues related to joining an already joined thread as orthogonal to our work.

The constructor is unremarkable, except that it already starts the new thread, and thus has to establish the precondition of `run`.

Method `run` performs its own computation – here trivial, but the important aspect is that it requires read access to the shared data `data.val` – and it also forks subtasks, joins them again and aggregates the results (again trivial). We assume that it is not possible, or efficient, to compute the number of subtasks upfront; Java's iterators, for example, don't even provide a way of querying the number of elements to come. For simplicity, we ignore the permissions required to use the list and the iterator in the loop invariant.

As done in the first example, we make use of unbounded permission counting to yield straight-forward assertions

[1]Note that `rds` cannot become negative before the release, because this would imply holding more than `1` permissions to `data.val`, which is impossible.

```
interface Node {
  public Iterator<Node> children()
}

class TreeTask extends Thread {
  private final ghost ARP frac;
  private int result;
  private final Node node;
  private final Cell data;

  public TreeTask(Node n, Cell c, ghost ARP f)
    requires acc(c.val, f) && acc(result)
  {
    node = n; data = c; frac = f;
    start() // Requires run()'s precondition
  }

  public void run()
    requires acc(data.val, frac) && acc(result)
    ensures  acc(data.val, frac) && acc(result)
  {
    if (node == null) result = data.val;
    else {
      ghost ARP f = fresh();
      List subs = new List<TreeTask>();
      for (Iterator it = node.children(); it.hasNext())
        invariant acc(data.val, frac - subs.size()*f) &&
                  forall s in subs ::
                    s.frac == f && s.data == data
      {
          // Let S = subs.size(). We have
          //   acc(data.val, frac - S * f)
        TreeTask sub = new TreeTask(it.next(), data, f);
          // Requires acc(data.val, f). Hence, that
          //       f < frac - S * f
          //   <=> f < frac / (S+1)
          // We add this as a constraint
        subs.add(sub);
      }

      // We have acc(data.val, frac - subs.size()*f)

      while (subs.size() > 0)
        invariant acc(data.val, frac - subs.size()*f) &&
                  forall s in subs ::
                    s.frac == f && s.data == data
      {
          // Let S = subs.size(). We have
          //   acc(data.val, frac - S * f)
        TreeTask sub = subs.removeLast();
        sub.join(); // Returns run()'s postcondition
          // We have acc(data.val, frac - S * f + f),
          // since sub.frac == f
        result += sub.result;
          // Show that we have at least
          //   acc(data.val, frac - (S - 1) * f)
          // which is exactly what we have
      }
    }
  }
}
```

Figure 2: A Recursive Tree Worker.

(here, loop invariants). That is, each subtask receives the same amount `f` of permissions to the shared data. Since we do not compute the number of subtasks upfront, we again need to constrain `f` on-the-fly. At the end of `run`, all permissions have been collected back again, and the task can itself return its `frac` permission.

3. CONSTRAINT SYSTEM

In this section, we introduce a notion of constraint system that can be used to explicitly explain (and justify) the assumptions made by techniques such as abstract read permissions. Recall that explicit values for these read permissions are not chosen in our approach [5], but rather constrained via assumptions at various points in the encoding. The constraint system introduced in this section is designed to reflect the accumulation of these constraints as upper bounds on *symbolic* permission amounts (we model these as permission-typed variables), such that the satisfiability of the constraints can be reduced to requiring an *ordering* on these symbolic amounts; we explain the role of this ordering in Sec. 3.1.

Definition 1. *A* positive fractional constraint system *over a set of variables V (we use v to range over V) is a finite set C of inequalities of the form $v < E$ where E is an* expression *formed from the following grammar (in which q is a positive rational constant: an element of $\mathbf{Q}^+$).*

$$E ::= \quad q \ | \ v \ | \ E\texttt{+}E \ | \ E\texttt{*}E \ | \ E\texttt{-}E \ | \ E\texttt{/}E$$

A constraint system is satisfiable *if there exists a partial mapping $\sigma : V \rightharpoonup \mathbf{Q}^+$ such that for all $(v_i < E_i) \in C$, we have that $\sigma(v_i) < \sigma(E_i)$ is (defined and) true, where σ is lifted from variables to expressions in the obvious way, except that if an intermediate result of a subtraction were to be non-positive, the entire result is undefined, and the enclosing constraint is considered not satisfied.*[2]

3.1 Layered Fractional Constraint Systems

It is easy to define constraint systems that are unsatisfiable, for example $C = \{v < v - \frac{1}{2}\}$. In this section, we define a sufficient condition for satisfiability: it must be possible for the constraints to be "layered", with respect to a partial ordering of the occurring variables. In particular:

- We need a partial order $\prec$ over the variables.
- All constraints must respect the order, in that variables can be constrained only with respect to "larger" variables in the ordering: if v' occurs in the expression E in a constraint $v < E$, then we must have $v \prec v'$.
- Furthermore, the only form of subtraction permitted is where a variable v' is subtracted from an expression E, where E contains only variables that may (according to $\prec$) constrain v'. Effectively, the subtraction implicitly prescribes an additional constraint $v' < E$ that ensures that the result is positive.

Formally, we extend partial orders on variables $\prec$ to expressions, according to the rules of Fig. 3.

L-Const $\dfrac{q \in \mathbf{Q}^+}{v \prec q}$ L-Plus $\dfrac{v \prec E_1 \qquad v \prec E_2}{v \prec (E_1 \texttt{ + } E_2)}$ L-Mult $\dfrac{v \prec E_1 \qquad v \prec E_2}{v \prec (E_1 \texttt{ * } E_2)}$

L-Div $\dfrac{v \prec E_1 \qquad v \prec E_2}{v \prec (E_1 \texttt{ / } E_2)}$ L-Minus $\dfrac{v \prec v' \qquad v' \prec E}{v \prec (E \texttt{ - } v')}$

Figure 3: Extending $\prec$ to Expressions: $v \prec E$.

[2]Since we do not permit 0 as a constant, or as a binding of a variable, then as long as the results of subtractions are always positive, division by zero will not happen.

Definition 2 (Layerable Constraint Systems). *A constraint system C is* layered by a partial order $\prec$ *if, for every constraint $(v < E) \in C$, we have $v \prec E$.*

A constraint system C is layerable *if there exists a partial order $\prec$ such that C is layered by $\prec$.*

Note that the choice of a partial order $\prec$ for layering a particular constraint system need not be unique.

Theorem 1. *If a positive fractional constraint system C is layerable then it is satisfiable.*

Proof. (Sketch) First we augment the set of constraints with $v < E$ for every subterm $E - v$ that exists in an existing constraint. We also add the constraint $v < 1$ for every variable, to make sure every variable is constrained. Since this process creates no new subtraction subterms, it will terminate. Let C^* be this augmented set of constraints. By the definition of layering, C^* is also layerable.

Then we extend the partial order of variables to a total order of the variables occurring in C, and assign concrete values to the variables in descending order, according to this total order. That is, for each variable, we collect its constraints, evaluate all of the constraining expressions, and assign the variable *half* of the minimum of these values.

$$\sigma'(v) = \frac{1}{2}\left(\min\{\sigma(E) \mid (v < E) \in C^*\}\right)$$

Layering ensures we never try to use the value of a variable before it is defined. The only danger is subtraction but, by augmentation, every subtraction is reflected in a constraint that ensures that the result is positive. □

3.2 Extending Layerable Constraint Systems

Given an existing layerable constraint system, one can add a fresh symbolic constant to the system, by extending the original set of variables V with an additional variable v'. The resulting constraint system is still layerable, since none of the constraints in C will mention v'. More usefully, one can also extend the partial order $\prec$, such that $v' \prec v$ for all $v \in V$; this is guaranteed to produce a partial order, since v' was not previously mentioned. In particular, the resulting layered constraint system can now be extended further by adding additional constraints to C of the form $v' < E$ where E is an expression of the *original* constraint system (i.e., v' does not occur in E). The resulting constraint system will *still* be layerable; routine inductive arguments can show that this property is preserved by all of the above steps.

So long as one always moves from one layerable system of constraints to another layerable system of constraints then the set of constraints will always be satisfiable. Concrete values for the symbolic permission amounts need never actually be chosen, even in an implementation (see Sec. 4); in a soundness argument one can argue that, at any given program point, one *could* assign a suitable value to every variable and evaluate all symbolic expressions to positive rational numbers. The critical property that an implementation must therefore satisfy, is that the constraints generated must always be guaranteed to be layerable.

The format of constraints directly supported by our layered constraint systems is very restricted (particularly for subtractions). However, constraints that are not of the permitted form can be accepted, so long as they have the same *meaning* as some rewritten constraint that would be permitted. For example, the constraint $v < 1 - (v + \frac{1}{2})$ can be rewritten as $v < \frac{1}{4}$, which is allowed according to Fig. 3.

4. IMPLEMENTATION

It may seem that a model of permissions based around symbolic permission amounts (and expressions over them), along with a constraint system of assumptions, is much more complex to implement than some more direct mathematical structure (such as rationals). However, the very fact that these values need not be fixed at any point, along with the argument that such constraints can be constructed to be satisfiable, means that the symbolic amounts themselves can be treated by a verification tool just like any other symbolic value (e.g., a program variable), about which limited information is known. In other words, a symbolic permission amount v can be simply encoded as a program variable v with a particular unknown rational value. This would not be the case if we used a more complex explicit mathematical domain, such as $\mathbf{Z}[\epsilon]^+$ as the domain. As a result, we retain the simplicity of rational numbers and arithmetic, while achieving the expressive power of a more powerful domain.

On the other hand, the constraint system can still exploit the staging differences; the value of an expression that does not depend on a constraint variable can be considered a constant for the purposes of the constraint system. Thus for example, the expression `1 / (rds + 1)` in our first example, which is used as a bound for `frac`, is a constant at the (dynamic) program point where the constraint is (notionally) added to the constraint system. Furthermore, since the monitor invariant ensures `rds >= 0`, it can be considered a *positive* constant, thus meeting the requirements for layering. Similar reasoning applies to our second example with the result returned by the `size` method.

The fact that permission expressions must have a specific shape in order to be layered, and thus, that certain expressions such as $v < 1 - v$ have to be rewritten in order to be layered (as discussed in the previous section), does not impose additional complexity in an actual implementation. The required rewriting is purely conceptual: as long as it is guaranteed that a constraint *can* be appropriately rewritten – which needs to be established in the soundness proof of the verifier – it is sound to add any constraints which are semantically equivalent to those permitted by our definition of layering.

It is also worth noting that constraints of the form "v is less than the current amount of permission held" can be soundly added even though a static verifier will in general not be able to track the precise amount of permissions held, for example, because of unknown aliasing relations, or incompleteness in the underlying prover. In such cases, a sound verifier is forced to work with weaker assumptions about the permissions currently held; provided that it knows that *strictly positive* amounts are held, this results only in stronger, but nonetheless satisfiable constraints.

We implemented the constraint permission system in our intermediate verification language Silver, which is part of our Viper verification infrastructure [7]. Silver provides a built-in permission type that can be used to declare permission-typed variables, a construct to assign fresh symbolic values to such variables, and a construct that instructs the verifiers to generate appropriate constraints when giving away read permissions. Following the argumentation above, we make it the responsibility of front-ends, that is, of tools that generate Silver code, to ensure that these constructs are used in a way such that all generated constraints are layerable.

5. RELATED WORK

We discussed several permissions models in the introduction. In this section, we focus on the permission models supported by our own verifier Chalice and by VeriFast.

5.1 Abstract Read Permissions in Chalice

Chalice implements our previous model of abstract read permissions [5]. This model is a special case of the model presented in this paper. The basic idea of ARPs in Chalice is as explained in Sec. 2. The restriction that constraints on a symbolic permission amount can be added only together with the symbolic variable being constrained (in the encoding of method calls) is, as already explained, a consequence of the fact that our previous work did not precisely work out a property that guarantees that a permission constraint system is satisfiable.

An interesting aspect of ARPs in Chalice is the treatment of symbolic variables that occur *negatively* in permission expressions, as in $1 - v$. Taking a fresh symbolic read permission v and then giving away $1 - v$ permission (leaving exactly v behind), followed by giving away a further v amount of permission, would, in a naïve encoding, generate the unsound assumption $v < v$ (to guarantee that some permission is left after the last transfer). To avoid such unsound constraints, without making the constraints overly weak, Chalice rearranges method preconditions to handle all permission expressions in which the constrained variable v occurs negatively last. Constraints are only added while handling permission expressions with positive occurrences of v. Consequently, while introducing the constraints for the positive occurrences of v, the current permission amount remains expressible by expressions of the form $E - n * v$, where v does not occur in E. Every time v is given away from such an expression, the assumption generated is effectively $v < E - n * v$. This assumption is equivalent to $v < \frac{E}{n+1}$, which matches our definition of a layered constraint system.

5.2 VeriFast

Another automated verifier that supports fractional permissions is the separation-logic-based verifier VeriFast [6]; fractions are encoded as real numbers. This allows unbounded splitting, but not unbounded counting. However, VeriFast's standard library contains an encoding of counting permissions as *tickets*, supported by ghost methods operating on auxiliary separation logic predicates. The idea is as follows: counting starts by calling a ghost method `start_counting(`r`, `f`)` that consumes f permissions to a field r. Intuitively, the initially consumed permission amount f is used as a "pool" from which an unbounded number of tickets can be taken by calling a ghost method `create_ticket(`r`)`, which requires a predicate provided by `start_counting`. Unlike counting permissions however, the tickets cannot be "summed up" easily (e.g. to represent holding n such tickets), because they are encoded as predicate instances. To sum up ticket predicates, ones needs additional recursive predicates that essentially encode a list of tickets.

Extending VeriFast with abstract read permissions as presented in this paper could potentially simplify bookkeeping in situations where tickets are currently used. Our constraint system could also be used to prove the soundness of ticket permissions, because the constraint implicitly generated for each call to `create_ticket(`r`)` is that the fraction associated with the new ticket is strictly less than the permission currently in the "pool" and, thus, satisfies our rules for layering.

6. CONCLUSIONS

We have presented a permission model based on the novel concept of layered constraints that supports unbounded divisibility and counting, as well as unrestricted multiplication. The formal model of constraints over symbolic rational variables is actually quite complex; however, it is possible to treat it as if it were the rational numbers with their usual arithmetic laws, which makes it intuitive to understand by users and straight-forward to implement. We have shown two non-trivial examples that illustrate the expressiveness of our model, and we have argued how it can be used in a proof of soundness of the permission models used in Chalice and VeriFast. Moreover, we have implemented support for it in the intermediate verification language Silver [7].

7. REFERENCES

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

[2] J. Boyland. Checking interference with fractional permissions. In *SAS*, volume 2694 of *LNCS*, pages 55–72. Springer, 2003.

[3] J. Boyland. Fractional permissions. In *Aliasing in Object-Oriented Programming*, volume 7850 of *LNCS*, pages 270–288. Springer, 2013.

[4] R. Dockins, A. Hobor, and A. W. Appel. A fresh look at separation algebras and share accounting. In *APLAS*, volume 5904 of *LNCS*, pages 161–177. Springer, 2009.

[5] S. Heule, K. R. M. Leino, P. Müller, and A. J. Summers. Abstract read permissions: Fractional permissions without the fractions. In *VMCAI*, volume 7737 of *LNCS*, pages 315–334. Springer, 2013.

[6] B. Jacobs and F. Piessens. The VeriFast program verifier. Technical Report CW-520, KU Leuven, Aug. 2008.

[7] U. Juhasz, I. T. Kassios, P. Müller, M. Novacek, M. Schwerhoff, and A. J. Summers. Viper: A verification infrastructure for permission-based reasoning. Technical report, ETH Zurich, 2014.

[8] X. B. Le, C. Gherghina, and A. Hobor. Decision procedures over sophisticated fractional permissions. In *APLAS*, volume 7705 of *LNCS*, pages 368–385. Springer, 2012.

[9] K. R. M. Leino and R. Monahan. Reasoning about comprehensions with first-order SMT solvers. In *SAC*, pages 615–622. ACM, 2009.

[10] 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.

[11] M. J. Parkinson and G. M. Bierman. Separation logic and abstraction. In *POPL*, pages 247–258. ACM, 2005.

[12] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, pages 55–74. IEEE, 2002.

[13] J. Smans, B. Jacobs, and F. Piessens. Implicit dynamic frames: Combining dynamic frames and separation logic. In *ECOOP*, volume 5653 of *LNCS*, pages 148–172. Springer, 2009.