

# Polymorphic Fractional Permission Inference

Kevin Bierhoff   Nels E. Beckman   Jonathan Aldrich

Institute for Software Research, Carnegie Mellon University

{kevin.bierhoff,nbeckman,jonathan.aldrich}@cs.cmu.edu

## Abstract

Fractional permissions have recently received much attention for sound static reasoning about programs that rely on aliasing, but they are challenging to track automatically. This paper contributes an algorithm for proving permission-based assertions in a decidable fragment of linear logic. Unlike previous work, our inference approach supports polymorphism over fractions. The paper also describes our implementation of this algorithm as part of a tool, *Plural*, which checks compliance to typestate-based protocols in Java code. We report *Plural*'s performance on several benchmark programs.

**Categories and Subject Descriptors** F.3.1 [Logics and Meanings of Programs]: Specifying and Verifying and Reasoning about Programs—Mechanical verification

**General Terms** Languages, verification.

**Keywords** Fractional permissions, linear logic, typestate checking.

## 1. Introduction

Fractional permissions (Boyland, 2003) have recently received much attention for sound static reasoning about programs that rely on aliasing. They have been used for avoiding data races with locks (Terauchi and Aiken, 2008) as well as for verifying properties of multi-threaded (Beckman et al., 2008; Leino and Müller, 2009) and single-threaded programs (Bornat et al., 2005; Bierhoff and Aldrich, 2007), where fractional permissions are typically embedded into a substructural logic. In particular, we have used permissions for sound modular reasoning about typestate-based protocols in the presence of aliasing (Bierhoff and Aldrich, 2007; Beckman et al., 2008). However, fractional permissions are challenging to track automatically. This is because in practice we would like users to not be required to write down literal fractions, which in turn makes fraction inference necessary.

This paper contributes an algorithm for proving permission-based assertions in a decidable fragment of linear logic (Girard, 1987). We believe that this algorithm carries over into other substructural logics, although its use in undecidable logics such as separation logic (Reynolds, 2002) will in general require approximations. Unlike previous work, our inference approach supports *poly-*

*morphism* over fractions, i.e., universally and existentially quantified fractions, which is useful for reasons that we will describe.

Permissions are typically used to reason about access to shared objects in memory. Fractional permissions as they were originally proposed enforce that objects are either modified through a single, unique reference, or alternatively read through any number of immutable, read-only references.

Fractional permissions intuitively associate each program reference with a rational *fraction* in the range  $(0, 1]$ . unique permissions by definition carry a fraction of 1. immutable permissions are represented by fractions less than 1, and indicate that other permissions to the same object exist, without saying exactly how many. Fractional permissions are useful, like most systems for reasoning about aliasing, because they restrict and record statically the patterns of aliasing used in a program, which can in turn allow us to prove useful behavioral properties about a program with mutable state.

Fractional permissions complicate proof search in substructural logics because permissions are divisible resources that can theoretically be *split* an arbitrary number of times to satisfy different logical assertions. (Linear and separation logic traditionally treat resources as indivisible.) They can also be *merged* back together, which may be necessary to prove a stronger permission. Splitting means dividing the fraction associated with a permission among two new permissions, while merging means to sum up the fractions from a set of given permissions into one permission.

The permission inference algorithm presented in this paper extends conventional resource management techniques for linear logic (Cervesato et al., 2000) to additionally infer how fractional permissions should be split and merged over time. This process requires collecting linear constraints over fraction variables and ensuring that these remain satisfiable.

Our approach has at least two important features when compared with existing approaches for fraction inference:

1. Method signatures can quantify over fractions for incoming and outgoing permissions. The result is that method specifications need not refer to literal fractions (e.g., “ $1/4$ ”) but rather can be specified to require *any* fraction and produce *some* fraction in return. This fractional polymorphism promotes modularity between independent parts of the program.
2. Code executing inside loops is allowed to *consume* permissions.

Regarding the first point, consider a method `wrap(Object x)` that wraps a reference  $x$  into a new object. It would be extremely inconvenient if we had to specify `wrap` to require a concrete fraction for  $x$  such as  $3/4 \cdot x$ : all callers of `wrap` would have to be able to provide that fraction for the reference they want to wrap, *regardless of what else they want to do*. For instance, they would not be able to also call `print(Object x)` if `print` requires  $1/2 \cdot x$ . These situations can obviously be resolved by changing either `print` or `wrap`, but not if both methods reside in re-usable libraries. Our approach

allows specifying these methods to require *some* (polymorphic) fraction  $\exists z.z \cdot x$ , which different callers may instantiate as they choose. It also simplifies changes in the implementation of these methods if they in turn also only depend on fraction-polymorphic methods: with concrete fractions, the fraction required by a method effectively becomes the sum of fractions consumed by the methods being called internally—and the method cannot be written if that sum is greater than 1—while polymorphic fractions in most all cases hide such details. Fraction polymorphism hence promotes independence between program modules and facilitates code re-use. In our view it is essential for reasoning about program modules independently and at the very least avoids arbitrary concrete fractions such as  $3/4$  to be written down by programmers, which Leino and Müller (2009) have already described as quite awkward.

With regard to the second point, consider the relatively common case of an application that spawns of a number of threads, passing each a modifying reference to some shared data structure. The following example illustrates this idea:

```
Blocking_queue createConsumers(int number) {
  Blocking_queue queue = new Blocking_queue();
  for( int i=0; i<number; i++) {
    (new Worker(queue)).start();
  }
  return queue;
}
```

The only previously published fraction inference system (Terauchi and Aiken, 2008) is a global analysis which does not support polymorphic fractions. This in practice leads to imprecisions in loops (see (Terauchi, 2008), chapter 5). Notably the above producer-consumer example could not be verified by their approach. They also do not support reasoning about *formulae* over permissions, whereas our inference system supports the typical operators of substructural logics.

We implemented our algorithm as a dataflow analysis that tracks permissions through conventional Java code. As a consequence of this strategy, loop invariants can be automatically inferred. This implementation is part of a tool, called Plural, which checks conformance to typestate-based protocols. Permissions allow Plural to check individual Java methods while making sound assumptions about the rest of the program even when that program contains aliasing. These assumptions are provided by developer specifications which describe the required and produced permissions. By successively checking every method in the program we can ensure the consistency of all assumptions.

Our experience with Plural (Beckman et al., 2008; Bierhoff et al., 2009) has been very positive for several reasons: developer-provided permission annotations have the flavor and extent of conventional static typing information, and our implementation is fast enough to be used by developers to check code as they write it. Our modular design allows the pin-pointing of errors to a particular program location, which simplifies debugging. It also enables the analysis of small program pieces, such as individual classes or libraries, independently from their clients and from the APIs upon which they depend. This is crucial for use in practice, and additionally allows us to verify the correct *implementation* of object protocols, which is not possible with previous global permission (Terauchi, 2008) and protocol analyses (Naeem and Lhoták, 2008; Bodden et al., 2008).

In summary, contributions of this paper include the following:

- A novel inference algorithm for formulae over fractional permissions in a large, decidable fragment of linear logic (section 2).
- An implementation of this inference algorithm as a dataflow analysis for Java programs. This modular implementation,

which tracks permissions as they flow through code, has the effect of inferring loop invariants (section 4).

- An evaluation of the tool’s performance on a variety of real programs. Our tool checks individual methods in less than 60ms on average (section 5).

Support for typestate checking required several extensions to the basic permission inference algorithm (section 3). Section 6 discusses related work before section 7 concludes.

## 2. Inference System

This section provides a deterministic type inference algorithm that collects linear constraints over fractions needed to satisfy permission-based linear logic predicates. We then discuss how the collected constraints can be solved.

Proof search for permission-based linear logic has several sources of non-determinism; in other words, finding a proof involves making several kinds of “guesses”:

- Context splits, such as in the linear logic proof rule for  $\otimes$ , “guess” how permissions have to be divided in order to satisfy different predicates. These non-deterministic context splits also appear in certain expression typing rules, for example in the `let` binding, in order to prove multiple subexpressions.
- Multiple proof rules can apply. For example, the rules for proving a disjunction  $P_1 \oplus P_2$  “guess” whether  $P_1$  or  $P_2$  can be proven.
- Permissions can be split an arbitrary number of times and merged back together. The type system “guesses” exactly how a given permission has to be split up and merged in order to satisfy different predicates. Notice that this additional non-determinism is not an issue in conventional linear logic proof search where resources are indivisible.

Our previously published typechecking and proof judgments  $\Gamma \mid \Delta \vdash M : \exists x.P$  and  $\Delta \vdash P$  (simplified and excluding conventional typing information) make these guesses to prove a predicate  $P$  based on given resources  $\Delta$  (with free variables  $\Gamma$  in  $\Delta$ ,  $P$ , and expression  $M$ ) (Bierhoff and Aldrich, 2007; Beckman et al., 2008). They essentially treat  $\Delta$  as an input and produce one output, the predicate  $P$  that could be proven. For example, typing a `let` expression involves guessing a context split between the two subexpressions, which may involve splitting up fractional permissions:

$$\frac{\Gamma \mid \Delta_1 \vdash M_1 : \exists x.P \quad \Gamma, x \mid \Delta_2, P \vdash M_2 : \exists x_2.P_2 \quad \Delta = \Delta_1, \Delta_2}{\Gamma \mid \Delta \vdash \text{let } x = M_1 \text{ in } M_2 : \exists x_2.P_2}$$

This sort of nondeterminism is reasonable to have in the typing rules for a language, since they form a declarative description of the facts that must hold for any well-typed program. But in order to actually check the well-typedness of a program, we need to define a series of deterministic type-checking rules.

First, in order to make context splits deterministic, we take a page from linear logic proof search (Cervesato et al., 2000) and add an additional output,  $\Delta'$ , to these judgments. This additional output will return all of the items from the incoming context that were not consumed by the proof.

Next, we must deal with situations where multiple proof rules for linear logic are applicable. This is the case for proving an external choice,  $P_1 \oplus P_2$  and for using an internal choice,  $P_1 \& P_2$ . Typically a backtracking approach is used, where an analysis attempts to prove one predicate and, if that fails, tries to prove the other predicate instead. However, because of our desire to implement our algorithm as a dataflow analysis (for the purposes of loop

invariant inference) we have chosen a different approach. Instead of backtracking, we will simply carry all possible choices forward, pruning out choices as they become infeasible.

This leaves one final source of non-determinism: the splitting and merging of fractional permissions. Inspired by constraint logic programming (Jaffar and Lassez, 1987), we extend our proof search technique by associating constraint implications  $C_0 \rightarrow C$  (abbreviated  $I$ ) with our linear context  $\Delta$ . The constraints  $C$  will record which permissions were needed in earlier parts of the method body.  $C_0$  represents *assumptions* we can make when solving  $C$ . These assumptions come from quantified fractions. Together we will refer to  $\Delta | C_0 \rightarrow C$  as an *atomic context*.

The following sections discuss our permission inference system in greater detail.

## 2.1 Syntax

<i>Programs</i>	$\Lambda$	::=	•   $\Lambda, m(x_1, \dots, x_n) : P \multimap E = M$   $\Lambda, c(x_1, \dots, x_n) : P \multimap \exists x.1 \cdot x$
<i>Expressions</i>	$M$	::=	$x_0.m(x_1, \dots, x_n)$   $\mathbf{new} \ c(x_1, \dots, x_n)$   $\mathbf{let} \ x = M_1 \ \mathbf{in} \ M_2$
<i>Expression types</i>	$E$	::=	$\exists x.P$
<i>Predicates</i>	$P$	::=	$k \cdot x$   $P_1 \otimes P_2$   $P_1 \& P_2$   $P_1 \oplus P_2$   $\exists z.P$
<i>Fractions</i>	$k$	::=	$z$   $\mathbf{Z}$   $1$   $0$
<i>Variable contexts</i>	$\Gamma$	::=	•   $\Gamma, x$
<i>Proof contexts</i>	$\Psi$	::=	$\Delta \mid I$   $\Psi_1 \& \Psi_2$   $\Psi_1 \oplus \Psi_2$
<i>Implication</i>	$I$	::=	$C_0 \rightarrow C$
<i>Permission set</i>	$\Delta$	::=	•   $\Delta, P$
<i>Constraints</i>	$C$	::=	$\top$   $\perp$   $F$   $C_1 \wedge C_2$
<i>Formulae</i>	$F$	::=	$k \doteq k_1 + k_2$   $0 < k$
<i>Program variables</i>	$x, y$	::=	$\mathbf{this} \mid \dots$
<i>Fraction variables</i>	$z$		
<i>Class names</i>	$c$	::=	$\mathbf{Object} \mid \dots$
<i>Node names</i>	$n$	::=	$\mathbf{alive} \mid \dots$
<i>Method names</i>	$m$		

**Figure 1.** Syntax for permission inference system

The syntax for the permission inference system is summarized in Figure 1. It is based on Featherweight Java (Igarashi et al., 1999), but for simplicity we are not keeping track of object types. Instead, we are assuming that conventional typechecking has already ensured type safety. Programs  $\Lambda$  are simply lists of methods (suitably renamed to avoid name clashes) and class constructors with their declared pre- and post-condition predicates. Expressions  $M$

<b>T-CALL</b>	$\frac{\text{mtype}(m) = \forall x_0, x_1, \dots, x_n. P \multimap E \quad \Psi \vdash P \Rightarrow \Psi'}{\Gamma \mid \Psi \vdash x_0.m(x_1, \dots, x_n) : E \Rightarrow \Psi'}$
<b>T-NEW</b>	$\frac{\text{init}(c) = \forall x_1, \dots, x_n. P \multimap E \quad \Psi \vdash P \Rightarrow \Psi'}{\Gamma \mid \Psi \vdash \mathbf{new} \ c(x_1, \dots, x_n) : E \Rightarrow \Psi'}$
<b>T-LET</b>	$\frac{\Gamma \mid \Psi \vdash M_1 : \exists x. P \Rightarrow \Psi' \quad \Gamma, x \mid \Psi', P \vdash M_2 : E \Rightarrow \Psi''}{\Gamma \mid \Psi \vdash \mathbf{let} \ x = M_1 \ \mathbf{in} \ M_2 : E \Rightarrow \Psi''}$

**Figure 2.** Typechecking rules for permission inference (helper judgments in Figure 3)

are standard; however, notice that we introduce a let-binding construct to define intermediate variables and only allow variables as arguments to methods and object constructors. This simplifies the theory.

Expression types  $E$  existentially bind a variable that represents the value computed by an expression and include a linear logic predicate  $P$ . Fractional permissions  $k \cdot x$  with fraction  $k$  for variable  $x$  represent the only kind of atomic linear predicate. Fractions include *unknown* fractions  $z$  as well as *logic variables*  $\mathbf{Z}$  which can be freely instantiated. All variables are implicitly at least 0 and at most 1; we will omit constraints to that effect for readability.

Proof contexts  $\Psi$  are built up from atomic contexts which include a linear context  $\Delta$  and the constraint implication  $I$ . Whenever we can make a choice between two rules we introduce a *choice context*  $\Psi_1 \& \Psi_2$  that carries the two possibilities forward (to avoid backtracking). When one of the choices fails to prove a predicate, we will simply drop it, and if no context remains then our proof fails. Dually, we introduce *all contexts*  $\Psi_1 \oplus \Psi_2$  to carry two possibilities forward that both have to be able to prove subsequent predicates. When one of the options of an *all* context fails to prove a predicate then the entire context is dropped, failing the proof if no other choices (see above) are available. This is useful for proof rules that are dual to the ones where we use *choice* contexts.

## 2.2 Typechecking

Expressions are typechecked with the following judgment (Figure 2):

$$\Gamma \mid \Psi \vdash M : \exists x. P \Rightarrow \Psi'$$

This judgment reads, “in context  $\Psi$ , expression  $M$  (with free variables defined in  $\Gamma$ ) will produce a value  $x$  with predicate  $P$  as well as a new context  $\Psi'$  for the remainder of the program”. For example, consider the new typing rule for let expressions first seen in the beginning of this section.  $\Psi$  is passed into the rule and used to prove the first subexpression. This sub-proof may generate more constraints and returns any unused permissions, both of which are in  $\Psi'$ .  $\Psi'$  is in turn used to prove the second subexpression, producing  $\Psi''$  in the process which is returned to the next part of the program.

The other typing rules effectively thread permission contexts through the program and invoke the judgment for proving a linear logic formula (see next section) as needed. Most interestingly, method and constructor pre-conditions will need to be proved. Field accesses will be discussed in section 4.1.7.

## 2.3 Proof Rules

The judgment for proving a predicate  $P$  from a given context  $\Psi$  is the following:

$$\begin{array}{c}
\text{SPLIT} \\
\frac{C' = k \doteq k' + \mathbf{Z}'' \wedge 0 < k \wedge 0 < k' \quad \mathbf{Z}'' \text{ fresh}}{\Delta, k \cdot x \mid C_0 \rightarrow C \vdash k' \cdot x \Rightarrow \Delta, \mathbf{Z}'' \cdot x \mid C_0 \rightarrow (C \wedge C')} \\
\\
\text{MERGE} \\
\frac{\Delta, \mathbf{Z}'' \cdot x \mid C_0 \rightarrow (C \wedge \mathbf{Z}'' \doteq k + k') \vdash P \Rightarrow \Psi \quad \mathbf{Z}'' \text{ fresh}}{\Delta, k \cdot x, k' \cdot x \mid C_0 \rightarrow C \vdash P \Rightarrow \Psi} \\
\\
\text{FAIL} \\
\frac{\text{no permission for } x \text{ in } \Delta}{\Delta \mid C_0 \rightarrow C \vdash k \cdot x \Rightarrow \Delta \mid C_0 \rightarrow (C \wedge \perp)} \\
\\
\otimes \text{L} \\
\frac{\Delta, P_1, P_2 \mid I \vdash P \Rightarrow \Psi}{\Delta, P_1 \otimes P_2 \mid I \vdash P \Rightarrow \Psi} \\
\otimes \text{R} \\
\frac{\Delta \mid I \vdash P_1 \Rightarrow \Psi_1 \quad \Psi_1 \vdash P_2 \Rightarrow \Psi_2}{\Delta \mid I \vdash P_1 \otimes P_2 \Rightarrow \Psi_2} \\
\\
\oplus \text{L} \\
\frac{\Delta, P_1 \mid I \vdash P \Rightarrow \Psi_1 \quad \Delta, P_2 \mid I \vdash P \Rightarrow \Psi_2}{\Delta, P_1 \oplus P_2 \mid I \vdash P \Rightarrow \Psi_1 \oplus \Psi_2} \\
\oplus \text{R} \\
\frac{\Delta \mid I \vdash P_1 \Rightarrow \Psi_1 \quad \Delta \mid I \vdash P_2 \Rightarrow \Psi_2}{\Delta \mid I \vdash P_1 \oplus P_2 \Rightarrow \Psi_1 \& \Psi_2} \\
\\
\& \text{L} \\
\frac{\Delta, P_1 \mid I \vdash P \Rightarrow \Psi_1 \quad \Delta, P_2 \mid I \vdash P \Rightarrow \Psi_2}{\Delta, P_1 \& P_2 \mid I \vdash P \Rightarrow \Psi_1 \& \Psi_2} \\
\& \text{R} \\
\frac{\Delta \mid I \vdash P_1 \Rightarrow \Psi_1 \quad \Delta \mid I \vdash P_2 \Rightarrow \Psi_2}{\Delta \mid I \vdash P_1 \& P_2 \Rightarrow \Psi_1 \oplus \Psi_2} \\
\\
\forall \text{L} \\
\frac{\Delta, [\mathbf{Z}/z]P' \mid C_0 \rightarrow (C \wedge \mathbf{Z} > 0) \vdash P \Rightarrow \Psi' \quad \mathbf{Z} \text{ fresh}}{\Delta, \forall z. P' \mid C_0 \rightarrow C \vdash P \Rightarrow \Psi'} \\
\forall \text{R} \\
\frac{\Delta \mid (C_0 \wedge z > 0) \rightarrow C \vdash P \Rightarrow \Psi'}{\Delta \mid C_0 \rightarrow C \vdash \forall z. P \Rightarrow \Psi'} \\
\\
\exists \text{L} \\
\frac{\Delta, P' \mid (C_0 \wedge z > 0) \rightarrow C \vdash P \Rightarrow \Psi'}{\Delta, \exists z. P' \mid C_0 \rightarrow C \vdash P \Rightarrow \Psi'} \\
\exists \text{R} \\
\frac{\Delta \mid C_0 \rightarrow (C \wedge \mathbf{Z} > 0) \vdash [\mathbf{Z}/z]P \Rightarrow \Psi' \quad \mathbf{Z} \text{ fresh}}{\Delta \mid C_0 \rightarrow C \vdash \exists z. P \Rightarrow \Psi'}
\end{array}$$

**Figure 4.** Proof rules for linear logic formulae

$$\begin{array}{c}
m(x_1, \dots, x_n) : P \multimap E = M \in \Lambda \\
\text{mtype}(m) = \forall \text{this}, x_1, \dots, x_n. P \multimap E \\
\\
c(x_1, \dots, x_n) : P \multimap E \in \Lambda \\
\text{init}(c) = \forall x_1, \dots, x_n. P \multimap E
\end{array}$$

**Figure 3.** Helper judgments for permission inference in Figure 2

$$\Psi \vdash P \Rightarrow \Psi'$$

This judgment will be presented in several steps. We first discuss the rules for deriving constraints from splitting and merging individual permissions. Afterwards, remaining rules for proving predicates from atomic contexts  $\Delta \mid I$  are discussed. Then we show how contexts  $\Psi$  are broken down into such atomic contexts. Finally, we discuss how focusing resolves remaining non-determinism.

**Atom Rules.** We only need two rules for dealing with atomic permissions, one for splitting and one for merging (Figure 4). The basic idea behind them is to delay splits until they are needed for proving a permission, while permissions for the same variable are merged eagerly. Both rules introduce constraints to relate the various permissions involved.

Constraint formulae have two forms (Figure 1): they can equate a fraction sum with another fraction or they can define a fraction to be strictly greater than 0. The former is used to relate fractions from different permissions. The latter type of formula will force fractions to be “real” as explained below.

In general a fraction cannot have a value of 0 if it is to be used to prove anything in our system. Intuitively, this is because fractions track how often permissions were split. A fraction of 0 would rep-

resent infinite splitting. However, sometimes extra permission for a reference may be left around in a method body even after the post-condition of the method is satisfied. Because our analysis will not know until constraints are solved whether or not there are permissions left over, it still must create a constraint variable representing this permission. This variable may end up being greater than 0 (i.e., “real”) or it may end up being equal to 0 after the constraints have been solved. If a fraction is used to prove something, we force it to be real by adding a constraint  $0 < k$ .

Let us see how these constraints are generated by the splitting and merging rules, shown in Figure 4.

- **SPLIT** proves a permission for some variable  $x$  if there is a permission for that same variable in the context (otherwise, the permission is trivially not provable). In this case we can add a “fresh” permission (a permission with a fresh fraction  $\mathbf{Z}''$ ) for the variable into the output context, which represents the “leftover” permission after splitting off the needed from the existing permission. We introduce constraints that force the fractions in the needed and leftover permissions to sum up to the fraction of the given permission. We also introduce constraints for enforcing the given and proven permissions to be “real”. Notice that the leftover permission is not required to be “real”.
- **MERGE** is used to eagerly merge permissions for the same variable. This ensures that we always have the strongest possible permission available to prove the next one. Two permissions for the same variable occur when permissions are injected into the context from the post-condition of an invoked method.
- **FAIL** makes trivial failure to prove a permission for a variable  $x$  explicit: the rule inserts “false” ( $\perp$ ) into the constraints if no permission for  $x$  whatsoever is available. Another way of thinking about this rule is that any variable not explicitly mentioned

$$\begin{array}{c}
\text{CTX-}\& \\
\frac{\Psi_1 \vdash P \Rightarrow \Psi'_1 \quad \Psi_2 \vdash P \Rightarrow \Psi'_2}{\Psi_1 \& \Psi_2 \vdash P \Rightarrow \Psi'_1 \& \Psi'_2} \\
\hline
\text{CTX-}\oplus \\
\frac{\Psi_1 \vdash P \Rightarrow \Psi'_1 \quad \Psi_2 \vdash P \Rightarrow \Psi'_2}{\Psi_1 \oplus \Psi_2 \vdash P \Rightarrow \Psi'_1 \oplus \Psi'_2}
\end{array}$$

**Figure 5.** Context proof rules

in  $\Delta$  implicitly has a zero ( $0 \cdot x$ ) fraction associated with it. Proving a permission for such a variable  $x$  using SPLIT would then result in the unsatisfiable constraint  $0 < 0$ .

**Proving Predicates.** Figure 4 also summarizes the judgments for proving linear logic formulae. They are described below:

- **Linear Connectives.** The rules for proving linear logic connectives ( $\otimes$ ,  $\&$ ,  $\oplus$ ) are a straightforward adaptation of resource management for linear logic. In particular, constraints are simply threaded through the proof in the obvious way. Notice that we use our “choice” and “all” contexts for handling connectives other than  $\otimes$  to avoid backtracking (see section 2.1).
- **Quantifiers.** The quantifier rules ( $\forall$ ,  $\exists$ ) allow us to support polymorphic fractions using logic variables and “unknowns”. When quantified fractions are introduced ( $\forall R$  and  $\exists L$ ) we simply strip the quantifier, turning the quantified variable into a parameter (which we will call an “unknown”) that can be thought of as a constant with unknown value. Of course, the quantified variable needs to be suitable alpha-converted to not capture existing parameters. We arrange that quantified fractions are always “real” by inserting suitable assumptions. When quantifiers are used ( $\forall L$  and  $\exists R$ ) we introduce logic variables to find a suitable instantiation. We ensure that our assumption that quantifiers contain “real” fraction holds by inserting the corresponding constraint on the logic variable being introduced.

To our knowledge, the handling of quantifiers is novel compared to previous work on fraction inference. Quantifiers motivate the introduction of assumptions  $C_0$  in constraint formulae  $C_0 \rightarrow C$ , which in previous work were not present. Resolving these constraints is discussed in Section 2.4.2.

Notice that the rules for  $\&$  and  $\oplus$  as well as the rules for  $\forall$  and  $\exists$  are exactly dual to each other, as expected.

**Context Rules.** The rules for breaking down “choice” and “all” contexts in order to invoke the linear logic proof judgment are shown in Figure 5. The rules stipulate that predicates are proven separately for compound contexts and put back together into a new compound context afterwards.

**Focusing.** The rules as presented are still not fully deterministic: we can still choose to break down a premise (and which premise) or to break down the conclusion. *Focusing* is the standard technique for making the choices for breaking down premises and conclusions deterministic (Andreoli, 1992). Since the formalization of focusing for linear logic is lengthy and standard we omit it here. Also note that we have not included the rules for solving linear implication ( $\multimap$ ). In practice we have found that the need for linear implication was made unnecessary by dynamic state tests (section 4.1.6) and Plural’s built-in notion of pre and post-conditions. Full support for implication would require the standard use of focusing in linear logic proof search.

## 2.4 Solving Constraints

The collection of constraints during typechecking allows us to effectively accumulate information about all the permissions needed

$$\begin{array}{c}
\text{T-METH} \\
\frac{\text{this}, x_1, \dots, x_n \mid P \mid \top \rightarrow \top \vdash M : E \Rightarrow \Psi \quad \models \Psi}{m(x_1, \dots, x_n) : P \multimap E = M \in \Lambda} \\
\hline
\text{SAT-ATOM} \quad \text{SAT-}\&\text{-L} \quad \text{SAT-}\&\text{-R} \\
\frac{C_0 \rightarrow C}{\models \Delta \mid C_0 \rightarrow C} \quad \frac{\models \Psi_1}{\models \Psi_1 \& \Psi_2} \quad \frac{\models \Psi_2}{\models \Psi_1 \& \Psi_2} \\
\hline
\text{SAT-}\oplus \\
\frac{\models \Psi_1 \quad \models \Psi_2}{\models \Psi_1 \oplus \Psi_2}
\end{array}$$

**Figure 6.** Well-defined methods

in a piece of code. But a program should only typecheck if constraints are satisfiable.

### 2.4.1 Checking Method Definitions

We could check satisfiability, written  $C_0 \rightarrow C$ , after typechecking every program expression. But there is no point in doing so since constraint satisfiability for the surrounding expression implies that the constraints for its sub-expressions are satisfied (because constraints are conjunctions that only ever grow in length). Therefore, we can *delay* constraint resolution to the outermost expression, which in our case are the method bodies. The relevant rule for ensuring that a method body is well-defined is as shown in Figure 6. The rules for context satisfiability require constraint satisfiability for atomic contexts and let us choose a component context in “choice” contexts, while both components must be satisfiable in “all” contexts. In practice, we balance delaying constraint solving with identifying and pruning unsatisfiable contexts early. In particular, it is easy to prune trivially unsatisfiable constraints (those containing  $\perp$ ) eagerly.

### 2.4.2 Constraint Satisfiability

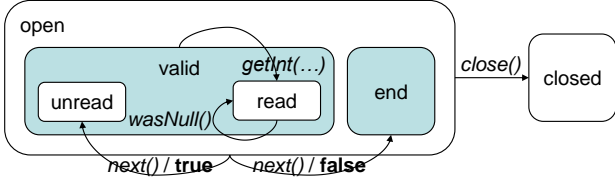
Constraints contain “unknown” variables, written  $z$ , and *logic variables*  $\mathbf{Z}$ . The quantifier proof rules ( $\forall$ ,  $\exists$ ) of Figure 4 introduce both kinds of variables, while the atomic predicate rules only introduce logic variables. Logic variables can be instantiated arbitrarily to satisfy constraints. Unknown variables, on the other hand, are assumed to have any value. Hence, in a given constraint implication  $C_0 \rightarrow C$ , unknowns can be thought of as universally quantified variables, while logic variables are existentially quantified. Our typing rules keep these quantifications implicit for notational convenience. If we make these quantifications explicit then determining constraint satisfiability  $C_0 \rightarrow C$  is equivalent to deciding the following logical formula:

$$\forall z_1, \dots, z_n. (C_0 \implies (\exists \mathbf{Z}_1, \dots, \mathbf{Z}_m. C)) \quad (1)$$

The *assumptions*  $C_0$  about unknowns introduced by  $\exists L$  and  $\forall R$  are always of the form  $0 < z$ , guaranteeing that quantified permissions are “real”, while  $C$  is an arbitrary constraint formula (see the constraint syntax in Figure 1). Notice that unknown and logic variables are always quantified in the same way, regardless of whether they result from universally or existentially quantified predicates.

Constraints of this form (that are not trivially true or false) can be checked for satisfiability using Fourier-Motzkin elimination.<sup>1</sup> In a nutshell, Fourier-Motzkin allows eliminating a given variable

<sup>1</sup>Linear programming (Schrijver, 1998) can unfortunately not deal with alternately quantified variables directly.



**Figure 7.** Simplified Java `ResultSet` protocol. Rounded rectangles denote states refining another state. Arches represent method calls, optionally with return values.

from a conjunction of linear constraints by re-writing the input formula into an equivalent formula that does not include the eliminated variable (Schrijver, 1998). In particular, the input formula is satisfiable if and only if the new formula is. We can successively eliminate all variables from our constraints using Fourier-Motzkin, which will result in a final constraint formula that is equivalent to the input formula and contains only rational constants but no variables. Equality and inequality between rational constants (such as  $0 < 1$  or  $1 = 1$ ), however, is decidable.

In order to deal with the alternating quantification in (1) we use Fourier-Motzkin to first eliminate the existentially quantified variables, then rewrite the outer universal quantifier into an existential quantifier (which introduces negations), and finally use Fourier-Motzkin to eliminate the remaining and now existentially quantified variables. (The last step could be accomplished with linear programming as well.)

Thus, Fourier-Motzkin elimination can be used to check our fraction constraint formulae (1) for satisfiability. Nipkow (2008) discusses more sophisticated algorithms for eliminating linear quantifiers, which are applicable to our constraint formulae. We leave the evaluation of these algorithms to future work.

### 3. Tpestate Tracking with Permissions

In this section we describe how we use fractional permissions to statically check tpestate-based object protocols in the presence of aliasing (Bierhoff and Aldrich, 2007). Doing so requires several extensions to our basic fraction inference approach.

Tpestates (Strom and Yemini, 1986) are a programming language concept that associates objects with finite state machines to define their *usage protocols*. For instance, the Java standard library defines a protocol shown in Figure 7 for the JDBC `ResultSet` interface, which represents the rows returned by a query over a relational database. In particular, `next` advances to the next *valid* row or reaches the *end* of the result and `getInt` retrieves a cell from the current *valid* row. As this figure indicates, our tpestates are arranged in a hierarchy, where mutually exclusive sets of states, such as *valid* and *end* refine a “bigger” state, such as *open* (Bierhoff and Aldrich, 2007).

Tpestate checking means to make sure that client programs such as the one shown in Figure 10 follow this protocol. It involves tracking the tpestate of references to objects with protocols. Aliasing complicates this task a great deal because other references may change the state of objects without our knowledge.

We employ permissions to control these interferences through other references. To this end, we extended the conventional notion of fractional permissions in several ways (Bierhoff and Aldrich, 2007). These extensions impact the permission inference system presented in the previous chapter in a subtle but important way. This section summarizes these extensions.

First off, not all references in our approach are either unique or immutable. We added share permissions, which allow modifi-

cations when objects are aliased (as opposed to immutable permissions, which forbid modifications). This is the effective permission associated with references in conventional programming languages. We also added the full / pure pair of permissions which grant exclusive modifying access (with possible reads through other references using pure permissions) and read-only access (with possible modifications through other references using full or share permissions), respectively. This pair of permissions allows asymmetric sharing of objects, where one party maintains full control of the shared object, as in a producer-consumer queue that can only be closed by the distinguished thread that puts work items into the queue (Beckman et al., 2008).

It turns out that we can encode these permissions with *two* fractions, one that simply counts the number of references (to distinguish unique from other permissions) and one that excludes references through pure permissions (Bierhoff and Aldrich, 2007). The second fraction is defined to be 0 for pure permissions. Additionally, we need a “read-only” bit that distinguishes share from immutable permissions, as share permissions are represented with fractions between 0 and 1 in the second fraction, just like immutable permissions. The rules for splitting and merging permissions can be extended in a straightforward way to maintain consistency between these permissions (Bierhoff and Aldrich, 2007): A unique permission can be split into a full and a pure permission. unique and full permissions can be split into share or immutable permissions, but not both. Using the two fractions, these splits can be reversed as before. Extending our fraction inference algorithm in this way is straightforward.

**Example.** Now that we have the share permission, we can verify the simple producer-consumer example shown in section 1. The rules described in section 2 generate the constraints shown in Figure 8. (For readability we omit the additional fraction and bit mentioned above and instead annotate variables with the kind of permission they carry.) We assume that the queue’s constructor returns a unique permission and the worker thread’s constructor takes and does not return a share permission to the queue, which will allow each worker thread and the producer to modify the queue. Inside the loop and before the constructor call, we have an unknown fraction  $z$  to the queue (which the flow analysis described in the next section generates). After the constructor, we have a fresh variable fraction  $Z$  to the queue and additional constraints relating the previous value of the fraction with the current value and the fraction  $Z'$  passed into the constructor. These variables and constraints come from proving the thread constructor’s pre-condition.

#### 3.1 State Guarantees

The combination of permissions with tpestates gives rise to our notion of a *state guarantee*. When a permission provides a state guarantee it promises not to leave the guaranteed state. This notion is extremely useful in practice: it increases expressiveness and allows more precise reasoning about share and pure permissions. The increased expressiveness allows us to express sharing a “server” object such as a database connection between multiple “clients” such as query results, which require the shared connection to remain *open* while in use. The *open* state becomes the state guarantee for the connection, and each query result can access it with a share permission. Since the *open* state is guaranteed to the query results, they do not have to check whether the connection was closed before accessing it, which improves reasoning precision (Bierhoff et al., 2009).

However, state guarantees do complicate permission inference. State guarantees can be introduced in full or unique permissions but only dropped in unique permissions. This is because pure permissions can be soundly unaware of a later introduced guarantee (because they do not modify the object anyway), but they may rely

```

Blocking_queue createConsumers(int number) {
    Blocking_queue queue = new Blocking_queue();
    // 1 * unique(queue) | true
    for (int i=0; i<number; i++) {
        // z * share(queue) | true
        (new Worker(queue)).start();
        // Z * share(queue) | z = Z + Z' ^ 0 < Z ^ 0 < Z'
    }
    return queue;
}

```

**Figure 8.** The constraints generated by the producer-consumer example shown in section 1.

```

public interface ResultSet {
    @Full(guarantee = "open")
    @TrueIndicates("unread")
    @FalseIndicates("end")
    boolean next();

    @Full(guarantee = "valid", ensures = "read")
    int getInt(int column);

    @Pure(guarantee = "valid", requires = "read")
    boolean wasNull();

    @Full(ensures = "closed")
    void close();
}

```

**Figure 9.** Simplified `ResultSet` specification in Plural (using the tpestates shown in figure 7).

on it, forbidding the guarantee from being dropped unless we are guaranteed that we have the unique permission to the object in question.

To handle this problem, we can modify our atom rules to introduce and drop state guarantees lazily as needed. If a guarantee needs to be introduced or dropped we can generate constraints that force the available permission to be full or unique, respectively. On the other hand, if a permission with the needed state guarantee is available (in other words, we have a “matching” guarantee), we do not need such additional constraints and essentially fall back to the SPLIT rule in Figure 4.

## 4. Plural: Java Tooling

Our prototype tool, Plural<sup>2</sup>, is a plug-in to the Eclipse IDE that checks tpestate-based protocols using permissions in Java based on the inference algorithm presented in the preceding sections. In this section we explain how Plural tracks permissions and how API implementations are verified. We also discuss tool extensions we found useful in practice.<sup>3</sup>

### 4.1 Flow Analysis for Local Permission Inference

Like any dataflow analysis, Plural tracks its analysis information in a lattice. Intuitively, the information Plural tracks are the permissions available for each object. In section 2 we discussed how permissions can be inferred automatically for a program using constraints, and how composite contexts allow reasoning about linear logic  $\oplus$  and  $\&$  connectives. Plural tracks constraints as part of its flow analysis information and includes support for composite con-

<sup>2</sup><http://code.google.com/p/pluralism/>

<sup>3</sup>Some of the details presented in this section appeared in Bierhoff et al. (2009) as background for the case studies described there, but without being a contribution of the paper.

```

public static Integer getFirstInt(
    @Full(guarantee = "open") ResultSet rs) {
    Integer result = null;
    if(rs.next()) {
        result = rs.getInt(1);
        if(rs.wasNull())
            result = null;
        return result;
    }
    else {
        // ERROR: rs in "end" instead of "valid"
        return rs.getInt(1);
    }
}

```

**Figure 10.** Simple `ResultSet` client with error in `else` branch that is detected by Plural.

texts, which will be discussed below. Plural’s transfer functions essentially implement the permission inference rules described in section 2 and will therefore not be discussed in detail here.

#### 4.1.1 Annotations Make Analysis Modular

Our goal is to avoid annotations inside method bodies completely: based on developer-declared method pre- and post-conditions (in the form of Java annotations such as `@Full` that describe permissions flowing in and out of methods, figures 9 and 10), Plural infers how permissions flow through method bodies. Since Plural is based on a dataflow analysis, it automatically infers loop invariants as well.

Annotations make the analysis modular: Plural checks each method separately, temporarily trusting annotations on called methods and checking their bodies separately. When checking a method or constructor, Plural assumes the permissions required by the method’s annotations. At each call site, Plural makes sure that permissions required for the call are available, splits them off, and merges permissions ensured by the called method or constructor back into the current context. Most methods “borrow” permissions (cf. section 4.2.1), which means that they are both required and ensured. At method exit points, Plural checks that permissions ensured by the method’s annotations are available, i.e., it checks the declared post-condition.

Thus, permissions are handled by Plural akin to conventional Java typing information: Permissions are provided with annotations on method parameters and then tracked automatically through the method body, like conventional types for method parameters. Unlike with Java types, local variables do not need to be annotated with permissions; instead, their permissions are inferred by Plural. Note that Plural annotations do not affect the run-time behavior of a program.

#### 4.1.2 Tuple Lattice

At the heart of Plural’s analysis is a tuple lattice for tracking permissions and constraints for individual objects. A tuple lattice in general tracks separate analysis information for each object and is compared and joined pointwise. The information tracked for each object is a pair: the permission currently available for the object together with the constraints collected for these permissions.

Such a tuple conceptually corresponds to an atomic context as discussed in section 2. However, it is a simplification because atomic contexts can in general contain linear logic formulae, whereas Plural tuples contain atomic permissions only. Formulae, e.g., from a method post-condition, have to be broken down into their constituent permissions before merging them into a given tuple. This has no consequences on precision in the case of multiplicative conjunctions  $P_1 \otimes P_2$ , where permissions  $P_1$  and  $P_2$  can

be separately merged into the tuple. There is a loss of precision when inserting  $\oplus$  and  $\&$  formulae. However, we *never* used such formulae in our case studies, so this issue has no consequences in practice.

#### 4.1.3 Comparing and Joining Permission Tuples

When comparing permissions for the same object (as mentioned, tuples are compared and joined pointwise), Plural compares these permissions using the following strategy:

1. When comparing permissions with the same state guarantee it compares fractions pairwise and treats a.) pairwise equal fractions as equal, b.) logic variable ( $\mathbf{Z}$ ) or literal fractions (such as 0 and 1) in place of unknown fractions ( $z$ ) from existential quantifiers in the other permission as more precise and c.) all other fractions as incomparable.
2. if one permission has a state guarantee but the other one does not, Plural attempts to align these permissions by introducing or dropping the guarantee and then uses the above rules to compare the new permissions; if guarantees can neither be introduced nor dropped then the permissions are considered incomparable.

The same rules guide the process of joining permissions: incomparable pairs of permissions with the same state guarantee are approximated (described below), and permissions with misaligned state guarantees are either aligned (by introducing or dropping the guarantee) or dropped (if alignment is not possible).

A pair of incomparable permissions is approximated by generating a new permission with fresh *unknown* fractions (in contrast to the fresh variable fractions introduced by the permission inference). Because these fresh unknown fractions have an unknown value they are deemed less precise than variable and literal fractions, as mentioned in the description of Plural’s comparison rules.

Notice that we do *not* attempt to determine whether a fraction is smaller than another, although this would allow comparison of permissions incomparable in the above rules (larger fractions are more precise). We have not found this to be a problem in practice, but it appears that such a refinement of our lattice comparison algorithm would be possible.

Our strategy of introducing unknown fractions to join incomparable permissions allows permissions to be consumed in loops, which was not previously possible (see (Terauchi, 2008)).

**Tuple lattice has finite height.** As with any dataflow analysis, the question arises whether ours is guaranteed to terminate, which, because of the iterative nature of the algorithm, reduces to showing that the lattice in use has finite height. We informally argue that this is the case.

- Tuple lattices have finite height (for a finite set of objects) if the information tracked for an individual object forms a lattice of finite height. In our case, this information is the permission tracked for an object.
- When joining individual permissions we either (a) preserve the fractions present in these permissions (if they are equal) or (b) approximate them with an unknown variable. Already approximated fractions will not be approximated again because the unknowns that were introduced are less precise than any other fraction (including other unknowns), terminating the process of approximation of individual permissions after one iteration.

Typically, our analysis terminates after three or less iterations of analyzing a loop body.

#### 4.1.4 Composing Tuples

The tuple lattice discussed above induces a lattice of composite tuples that correspond to the composite *choice* and *all* contexts from section 2. Formally, we define the elements  $A$  of this lattice as follows:

$A, B, C ::=$	$T$	<i>atomic tuple</i>
	$A \& B$	<i>choice element</i>
	$A \oplus B$	<i>all element</i>
	$\top$	<i>top</i>
	$\perp$	<i>bottom</i>

Comparing the elements of this lattice is straightforward, given the comparison  $T_1 \sqsubseteq T_2$  for individual tuples we described in section 4.1.3.

$$\begin{array}{c}
 \frac{A \sqsubseteq C}{A \& B \sqsubseteq C} \quad \frac{B \sqsubseteq C}{A \& B \sqsubseteq C} \quad \frac{A \sqsubseteq B \quad A \sqsubseteq C}{A \sqsubseteq B \& C} \\
 \\
 \frac{A \sqsubseteq C \quad B \sqsubseteq C}{A \oplus B \sqsubseteq C} \quad \frac{A \sqsubseteq B}{A \sqsubseteq B \oplus C} \quad \frac{A \sqsubseteq C}{A \sqsubseteq B \oplus C} \\
 \\
 \overline{A \sqsubseteq \top} \quad \overline{\perp \sqsubseteq A}
 \end{array}$$

This lattice allows reasoning about linear logic predicates that include internal ( $\&$ ) and external choice ( $\oplus$ ) operators. Unfortunately, however, the lattice has infinite height. We can finitize it by limiting the nesting depth of tuples. For example, we can restrict ourselves to only allow atomic tuples in  $\&$ -elements and only  $\&$ -elements in  $\oplus$ -elements, similar to a disjunctive normal form. Such a restriction is not complete with respect to the infinite lattice because of our limitation to a fragment of linear logic. In particular, the problem is that in the constructive fragment we are using (which does not include negation), certain De Morgan rules only work in one direction but not the other. For example  $(A \& B) \otimes C \vdash (A \otimes C) \& (B \otimes C)$ , but  $(A \otimes C) \& (B \otimes C) \not\vdash (A \& B) \otimes C$ .

The implementation of Plural is currently restricted to only use  $\&$ -elements. This simplification was possible because we only used conjunctions in our case studies, never internal or external choices. Furthermore, we suspect that the added benefit of  $\oplus$ -elements would be small because conventional joining of lattice elements achieves something very similar to  $\oplus$ -elements. Joining approximates two elements into one that contains as much information from the two original elements as possible. The resulting lattice element must then be able to satisfy the subsequent predicates. With  $\oplus$ -elements, Plural still has to make sure that *both* alternatives satisfy all subsequent predicates.  $\&$ -elements, on the other hand, allow delaying choices, which was in particular useful in the context of API implementation checking (see below).

#### 4.1.5 Local Alias Analysis

Internal to a method body, Plural uses a simple local alias analysis. Any objects that escape from the local context will be “tracked” through the permission system. Therefore the purpose of Plural’s local alias analysis is *not* to soundly approximate runtime aliasing. It merely prevents us from having to split permissions when a local is assigned to another local with a *copy* instruction such as  $x = y$ , avoiding developer-provided annotations in method bodies.

#### 4.1.6 Dynamic State Tests

APIs often include methods whose return value indicates the current state of an object, which we call *dynamic state tests*. For example, `next` in figure 9 is specified to return `true` if the cursor was advanced to a *valid* row and `false` otherwise.

In order to take such tests into account, Plural performs a *branch-sensitive* flow analysis: if the code tests the state of an object, for instance with an `if` statement, then the analysis updates the state of the object being tested *according to the test's result*. For example, Plural updates the result set's state to *unread* at the beginning of the outer *if* branch in figure 10. Likewise, Plural updates the result set's state to *end* in the *else* branch and, consequently, signals an error on the call to `getInt`. Recent whole-program protocol analyses (Naeem and Lhoták, 2008; Bodden et al., 2008) would miss this error because they merely ensure that dynamic state tests are *called*.

Notice that this approach does not make Plural path-sensitive: analysis information is still joined at control-flow merge points. Thus, at the end of figure 10, Plural no longer remembers that there was a path through the method on which the result set was *valid*.

When checking the implementation of a state test method, Plural checks at every method exit that, assuming `true` (or `false`) is returned, the receiver is in the state indicated by `true` (resp. `false`). This approach can be extended to other return types, although reasoning about predicates such as integer ranges may require using a theorem prover.

#### 4.1.7 API Implementation Checking

Our approach not only allows checking whether a client of an API follows the protocol required by that API, it can also check that code implementing an API is safe when used with its declared protocol. The key abstraction for this are *state invariants*, which we adapted from Fugue (DeLine and Fähndrich, 2004). A state invariant associates a typestate of a class with a predicate over the fields of that class. In our approach, this predicate usually consists of access permissions for fields.

Whenever a receiver field is used, Plural *unpacks* a permission to the surrounding object to gain access to its state invariants (Bierhoff and Aldrich, 2007). Essentially, unpacking means replacing the receiver permission with permissions for the receiver's fields as specified in state invariants. Before method calls, and before the analyzed method returns, Plural *packs* the receiver, possibly to a different state, by splitting off that state's invariant from the available field permissions (Bierhoff and Aldrich, 2007).

#### 4.1.8 Error Reporting

Plural reports possible protocol violations at the point in the program where errors are detected. In particular, if a method pre-condition cannot be satisfied at a call site then Plural will issue a warning at that call site, explaining the nature of the violation. Plural finds violations by querying the results of the dataflow analysis and checks at each method call site whether the pre-condition is satisfiable. The actual error may precede the point in the method where a protocol violation occurs, for instance because an erroneous method call may set up a situation where a later pre-condition becomes unsatisfiable. But the error will always be in the method being identified by Plural, including a wrong declared pre-condition.

## 4.2 Extensions

This section describes features implemented in Plural that are *not* captured in the theory underlying the tool but which we found useful in practice (see section 5).

### 4.2.1 Borrowing

Over and over again we encountered methods that *borrow* permissions, meaning they return the same permission that was passed in

when they were invoked.<sup>4</sup> Technically, our inference system cannot express borrowing because it only quantifies over fractions inside a method pre- or post-condition, but not both. Borrowed permissions are easily supported in Plural because they are essentially permissions whose fractions are universally quantified across both the method pre- and post-condition.

Support for borrowing increases both precision and performance. To see how precision is increased, consider the following:

When a method call requires introducing a state guarantee for a permission that is borrowed, then it is *always* safe to drop the state guarantee after the call. Without support for borrowing, there would be an error in figure 10. The reason is that the given full permission is strong enough to introduce the *valid* state guarantee for `getInt`, but not to drop it afterwards, which is necessary to return the permission with the *open* state guarantee as declared. With borrowing, we know that we can restore the previous state guarantee.

Support for borrowing increases performance because we do not have to keep constraints introduced for borrowed permissions beyond the current call site and can instead revert to the permission that was previously in the lattice. In practice, this cut the time it took our regression suite to run in half.

### 4.2.2 Concrete Predicates

Concrete predicates track information about *values* such as Booleans as well as the null-ness of references. Tracking the former is necessary for properly handling dynamic state tests, while the latter is useful in implementation predicates as well as for avoiding null dereference errors.

Every *test* in a program, for example in an *if* statement or loop header, ascertains the truth or falsehood of some (temporary) variable at run time. This information often *implies* or *indicates* some other fact. For example, truth of the `ResultSet`'s `next` method's return value indicates that the method call's receiver is in the *valid* state (see figure 9).

Plural maintains a list of known implications from Boolean to other facts and eagerly eliminates them when knowledge about the value of a Boolean variable becomes available. Plural handles the fact that a reference is "null" or "non-null" similar to a Boolean fact about a variable. If null-ness is tested at run time, Plural produces an implication from a Boolean to a null-ness fact as described above.

### 4.2.3 Dealing with Java

This section details how Plural handles Java features such as constructors, static fields, and arrays. All of the features discussed in this section are common in practical programming languages; therefore, the discussion applies to languages other than Java as well.

**Constructors.** We leverage API implementation checking to reason about constructors by injecting an unpacked unique permission for the receiver into the initial lattice element. The constructor post-condition is developer-declared and checked as usual.

**Static Fields (Globals).** Sometimes we need to associate permissions with static fields. (Static fields are similar to global variables in procedural languages.) Plural currently allows doing so with permission annotations directly on the field. To simplify matters, however, static fields can only be associated with permissions that can be duplicated (more precisely, split into equally powerful permissions, i.e. *share*, *immutable*, and *pure*). This avoids problems when permissions from a static field access are still in use when the same static field is accessed a second time.

<sup>4</sup> The other common case is that permissions are *consumed*, i.e. not returned to the caller of a method.

**Arrays.** Plural associates arrays with a permission of their own and makes sure that a modifying permission (unique, full, or share) is available upon stores into the array. Plural does *not* currently allow tracking permissions for the array elements, which is subject of future work. This issue is related to putting permissions into containers such as lists and sets.

## 5. Experience

We have used Plural, the tpestate verification tool described in Section 4, to specify and verify a number of real programs. In this section, we show that the performance of Plural and the fraction inference algorithm it implements is good enough to be usable in practice.

Figure 11 shows the results of running Plural on several programs. Because our analysis is a modular one, we have highlighted the number of methods in each program and the average time Plural spent on each method. As long as Plural shows reasonable performance on larger and more interesting methods, we expect our analysis will scale to programs with a very large total size.

In order to gauge the performance of Plural, we ran Plural on several programs, case studies and benchmarks and then recorded the time it took Plural to verify each program as well as some basic metrics about each program.<sup>5</sup> Figure 11 shows for each program (from left to right), the mean runtime of four runs in seconds, the total number of methods, the runtime per method in milliseconds, the number of lines of source code inside method bodies (excluding e.g., field initializers), the lines of source per method and the size in lines of source of the largest method. (For technical reasons, we could not determine the analysis time for the largest method separately.) We benchmarked five Java programs:

**PMD.** PMD<sup>6</sup> is a static analysis framework for Java that we specified and verified as part of an earlier case study (Bierhoff et al., 2009). This program is interesting because it uses the Java Iterator interface extensively, and it is relatively large. Plural successfully verified that the iterator protocol was obeyed. While the overall runtime was larger for PMD due to its size, the time spent verifying each method was rather low due to the relatively small number of permissions that were tracked per method.

**Regression Suite.** We ran Plural over the entire Plural regression suite, which consists of 132 classes and is meant both to test basic functionality as well as some more interesting cases that previously caused our analysis to exhibit bugs.

**4InALine.** 4InALine<sup>7</sup> is a Swing video game based on the board game Connect Four. As part of another case study we specified just the aliasing behavior of 23 of the classes in this program and used Plural to verify the consistency of those annotations.

**Beehive.** Apache Beehive<sup>8</sup> is a application framework designed to simplify the development of J2EE-based applications. As part of an earlier case study (Bierhoff et al., 2009) we specified protocols in the Java collections framework, the JDBC framework, the Java regular expressions classes and the Throwable class (which defines a very simple protocol for the setting of its “cause”). We verified 11 classes in this program.

**Blocking\_queue.** Blocking\_queue is a case study used to evaluate the NIMBY (Beckman et al., 2008) analysis program, an extension to Plural which also checks correct usage of concurrency.

<sup>5</sup> All program were run on a Dell PC running Windows XP Service Pack 2, with a 3.2GHz Intel Pentium 4 and 2GB of RAM.

<sup>6</sup> <http://pmd.sourceforge.net/>

<sup>7</sup> <http://code.google.com/p/fourinaline/>

<sup>8</sup> <http://beehive.apache.org/>

This case study includes the blocking queue class, which defines an open/closed protocol, and several clients which use the queue in interesting ways.

**Discussion.** The performance of Plural was generally good. As we expected, there seems to be correlation between the size of the methods in a program and the amount of time it takes to analyze each method. The exception here is the Blocking\_queue benchmark which takes longer to analyze than its average method size would suggest. We believe this is due to the relative complexity of the invariants and the large number of permissions that are being used throughout this program as well as the additional checks that must be performed in order to guarantee thread safety.

## 6. Related Work

Fractional permissions were proposed by Boyland for avoiding data races in concurrent programs (Boyland, 2003) and have since been used in a variety of contexts.

We previously proposed type systems for sound and modular checking of tpestate-based protocols that extend Boyland’s permissions to allow a great deal of flexibility in how objects can be aliased (Bierhoff and Aldrich, 2007; Beckman et al., 2008). In particular, we introduced new kinds of permissions, the notion of state guarantees, and used linear logic to compose predicates from permissions.

This paper provides a permission inference system for a fragment of linear logic predicates that is polymorphic with respect to fractions and describes its implementation in a protocol checking tool for Java that can fully check individual methods in less than 60ms on average.

Terauchi and Aiken (2008) previously proposed a whole-program fraction inference system for checking data races in concurrent programs whose implementation is based on a linear programming engine (Terauchi, 2008). In their work, control flow merges introduced new fraction variables that were forced to be smaller than the incoming fractions. Function signatures were also inferred with fraction variables. Constraints could then be collected by scanning the entire program once, even in the presence of conditionals and loops, allowing constraint collection to be asymptotically linear in the size of the program.

Conversely, we collect constraints with an intra-procedural flow analysis, which is polynomial in the worst case (Nielson et al., 1999, chapter 6) and requires annotations for permissions passed into and out of methods. An advantage of our modular approach is that permissions can be consumed in loops, which was not previously possible (see Terauchi, 2008, section 5). Figure 8 shows such an example. We are also able to use universal quantification in function signatures (currently provided with explicit annotations), which may be more flexible than inferring concrete fractions for signatures. These differences are due to our support for polymorphic fractions, which are not available in previous work.

Boyland’s ongoing work on avoiding data races unfortunately elides the details of their fraction inference implementation (Zhao, 2007).

Bornat et al. (2005) combined Boyland’s fractional permissions with separation logic, but their approach is intended for handwritten proofs about program correctness. VeriFast is an automated proof checker for separation logic that to our knowledge supports Boyland’s fractional permissions (Jacobs and Piessens, 2008), but the details of the implementation are again unknown. VeriFast seems to require significant developer input.

Other ongoing work encodes Boyland’s fractions using integers representing percentages between 0 and 100 as well as “infinitesimal” fractions in first-order logic (Leino and Müller, 2009). Loop invariants have to be provided by hand, while our implementation

Program	Runtime (s)	Methods	Runtime / Method (ms)	LOC	LOC / Method	LOC in Largest Method
PMD	259.978	4198	61.93	30594	7.288	517
Regression Suite	10.151	491	20.67	1867	3.802	102
4InALine	6.575	206	31.92	1405	6.82	71
Beehive	9.758	52	187.65	623	11.98	114
Blocking_queue	3.278	25	131.10	142	5.68	43
<b>Total</b>	<b>289.739</b>	<b>4972</b>	<b>58.27</b>	<b>34631</b>	<b>6.961</b>	<b>517</b>

**Figure 11.** Results of running the Plural tpestate checker on various programs.

infers loop invariants. Unlike with polymorphic fractions, it appears that the (duplicable) infinitesimal fractions cannot be used to borrow or otherwise temporarily alias objects, such as database connections. Consequently, this unusual encoding will require developers to consistently use concrete percentage values in most of their method pre-conditions and loop invariants.

## 7. Conclusions

Fractional permissions are a very promising mechanism for reasoning about programs with aliasing. This paper contributes a permission inference algorithm for polymorphic fractions and its implementation in our prototype tool Plural for automated sound and modular reasoning about tpestate-based protocols using permissions. The tool analyzes methods of several benchmarks in less than 60ms on average, indicating sufficient performance for being used by developers similar to a conventional typechecker.

## Acknowledgments

The authors thank Frank Pfenning and Rob Simmons for their help with linear logic proof search.

## References

J.-M. Andreoli. Logic programming with focusing proofs in linear logic. *Journal of Logic and Computation*, 2(3):197–347, 1992.

N. E. Beckman, K. Bierhoff, and J. Aldrich. Verifying correct usage of Atomic blocks and tpestate. In *ACM Conference on Object-Oriented Programming, Systems, Languages & Applications*, pages 227–244, Oct. 2008.

K. Bierhoff and J. Aldrich. Modular tpestate checking of aliased objects. In *ACM Conference on Object-Oriented Programming, Systems, Languages & Applications*, pages 301–320, Oct. 2007.

K. Bierhoff, N. E. Beckman, and J. Aldrich. Practical API protocol checking with access permissions. In *European Conference on Object-Oriented Programming*, pages 195–219, July 2009.

E. Bodden, P. Lam, and L. Hendren. Finding programming errors earlier by evaluating runtime monitors ahead-of-time. In *ACM Symposium on the Foundations of Software Engineering*, pages 36–47, Nov. 2008.

R. Bornat, C. Calcagno, P. O’Hearn, and M. Parkinson. Permission accounting in separation logic. In *ACM Symposium on Principles of Programming Languages*, pages 259–270, Jan. 2005. doi: <http://doi.acm.org/10.1145/1047659.1040327>.

J. Boyland. Checking interference with fractional permissions. In *International Symposium on Static Analysis*, pages 55–72. Springer, 2003.

I. Cervesato, J. S. Hodas, and F. Pfenning. Efficient resource management for linear logic proof search. *Theoretical Computer Science*, 232:133–163, Feb. 2000.

R. DeLine and M. Fähndrich. Tpestates for objects. In *European Conference on Object-Oriented Programming*, pages 465–490. Springer, 2004.

J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In *ACM Conference on Object-Oriented Programming, Systems, Languages & Applications*, pages 132–146, 1999. URL [citeseer.ist.psu.edu/igarashi99featherweight.html](http://citeseer.ist.psu.edu/igarashi99featherweight.html).

B. Jacobs and F. Piessens. The VeriFast program verifier. Technical Report CW-520, Department of Computer Science, Katholieke Universiteit Leuven, Aug. 2008.

J. Jaffar and J.-L. Lassez. Constraint logic programming. In *ACM Symposium on Principles of Programming Languages*, pages 111–119, Jan. 1987.

K. R. M. Leino and P. Müller. A basis for verifying multi-threaded programs. In *European Symposium on Programming*, pages 378–393. Springer, Mar. 2009.

N. Naeem and O. Lhoták. Tpestate-like analysis of multiple interacting objects. In *ACM Conference on Object-Oriented Programming, Systems, Languages & Applications*, pages 347–366, Oct. 2008.

F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer, 1999.

T. Nipkow. Linear quantifier elimination. In *International Joint Conference on Automated Reasoning*, pages 18–33. Springer, 2008.

J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *IEEE Symposium on Logic in Computer Science*, pages 55–74, 2002.

A. Schrijver. *Theory of Linear and Integer Programming*. Wiley, July 1998.

R. E. Strom and S. Yemini. Tpestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering*, 12:157–171, 1986.

T. Terauchi. Checking race freedom via linear programming. In *ACM Conference on Programming Language Design and Implementation*, pages 1–10, June 2008.

T. Terauchi and A. Aiken. A capability calculus for concurrency and determinism. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 30(5):1–30, Aug. 2008. ISSN 0164-0925. doi: <http://doi.acm.org/10.1145/1387673.1387676>.

Y. Zhao. *Concurrency Analysis Based on Fractional Permission System*. Ph.D. Dissertation, University of Wisconsin-Milwaukee, Aug. 2007.