Fraction–Polymorphic Permission Inference

Kevin Bierhoff, Nels E. Beckman, and Jonathan Aldrich

Carnegie Mellon University, Pittsburgh, PA, USA
{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, this inference approach supports polymorphism over fractions. The paper shows how permission inference can be implemented in a dataflow analysis that is able to infer loop invariants even when permissions are consumed in loops.

1 Introduction

Fractional permissions [8] 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 [22] as well as for verifying properties of multi-threaded [2, 14] and single-threaded programs [7, 4], where fractional permissions are typically embedded into a substructural logic. In particular, the authors have proposed using permissions for sound modular reasoning about typestate-based protocols in the presence of aliasing [4, 2]. However, fractional permissions are challenging to reason about automatically. This is because in practice we would like users to not have to explicitly annotate program expressions with concrete fractions, which in turn makes fraction inference necessary.

This paper contributes an algorithm for proving predicates over fraction-polymorphic permissions in a decidable fragment of linear logic [10] (Sect. 2), which we believe carries over to other substructural logics. Unlike previous work, this algorithm supports *polymorphism* over fractions: it is able to instantiate quantifiers when proving programmer-declared predicates such as method preconditions. (Quantifiers can also be inferred in loops, see below.)

The paper further shows that an intra-procedural dataflow analysis can automatically track permissions through imperative programs based on this algorithm (Sect. 3) to ensure that valid fractional permissions are available when aliased objects are accessed. In particular, our prototype implementation, Plural, only requires annotations on object fields and method parameters to track permissions in Java programs. Plural uses permissions for sound verification of correct usage and implementation of typestate [20] protocols. We have previously described our experience with using Plural [6]; this paper is the first description of its underlying permission inference algorithm and dataflow analysis. 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 [8].

Fractional permissions intuitively associate each program reference (x) with a rational *fraction* (k) in the range $(0, 1]$, written $k \cdot x$. unique permissions by definition carry a fraction of 1. immutable permissions are represented by fractions less than 1 and indicate that more (immutable) 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. Since permissions are resources that cannot be duplicated, they are typically reasoned about in the context of a substructural logic such as linear or separation logic.

Fractional permissions complicate the automatic derivation of proofs about program behavior because permissions are divisible resources that can theoretically be *split* an arbitrary number of times to satisfy different assertions. (Linear and separation logic traditionally treat resources as indivisible.) They can also be *merged* back together, which may be necessary to obtain 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.)

There are at least two important goals in automatically deriving proofs about realistic programs: comprehensiveness and modularity. *Comprehensiveness* calls for an automated prover to work with most useful code; *modularity* means checking parts of a larger program separately, which allows complex provers to scale and to check code that employs or implements reusable libraries [4]. By supporting universally and existentially quantified fractions in permissions—i.e., *fraction-polymorphic* permissions—with our inference system we elegantly achieve both of these goals.

Previously published fraction inference systems [22, 14] attempt to assign concrete fractions, such as 3/4, to all references in the program, an approach that, while intuitive, is neither modular nor comprehensive:

Modularity. Any module system requires type annotations at module boundaries [e.g., 16]. In previous approaches programmers write explicit fractions, like $3/4 \cdot x$, at module boundaries. This seems to resolve the modularity issue, but it reveals too much information: any change to a module that results in a different aliasing pattern is likely to affect fixed fractions in the interface, and changing the interface breaks compatibility with code that previously linked to the module. On the other hand, polymorphic specifications are naturally robust to many kinds of code changes, allowing methods to require *some* fraction and produce *any* fraction in return. In fact, in all our case studies [4, 2, 5, 6] we have needed concrete fractions in only two examples [4, 5], suggesting that fraction-polymorphic permissions are a much better "fit" for most programs.

Comprehensiveness. Loops sometimes reduce the available fraction for a program reference in every iteration, and as a result no concrete fraction can be

assigned to such a reference. Terauchi [21, section 5] observes that his approach cannot handle loops like the following, in which we assume that every new `Worker` consumes a fractional permission to the `queue` object passed into its constructor:

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

In contrast, our algorithm continues to work because the remaining permission for `queue` in the example above always carries *some* fraction. The dataflow analysis presented in this paper handles loops based on this idea without any developer intervention.

In summary, contributions of this paper include the following:

- A novel inference algorithm for proving pre- and post-conditions over polymorphic fractional permissions in a large, decidable fragment of linear logic (Section 2).
- A dataflow analysis based on the algorithm which tracks permissions as they flow through code (Section 3).

Section 4 discusses related work before section 5 concludes.

2 Inference System

This section provides a deterministic algorithm for proving linear logic predicates over fractional permissions in a core object-oriented language with explicit predicates only required in method signatures. In a nutshell, the algorithm collects linear arithmetic constraints over fractions needed to satisfy predicates encountered during typechecking. These constraints can then be checked for satisfiability. The appearance of certain linear logic connectives gives rise to more interesting proof contexts outlined below.

Devising such an algorithm is non-trivial because proving linear logic predicates based on fractional permissions is highly non-deterministic; in other words, finding a proof otherwise involves a lot of "guesswork:"

- Permissions can be split an arbitrary number of times and merged back together. Exactly how a given permission has to be split up and merged in order to satisfy different predicates has to be "guessed." This non-determinism is not an issue in conventional linear logic proof search where resources are indivisible.
- 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.

– The order in which proof rules are applied matters, requiring "guessing" whether a premise or the conclusion has to be broken down next. The standard technique of *focusing* [1] efficiently and deterministically tries all possible orders of applying proof rules. Hence we do not further discuss this problem here.

Linear logic proof rules are typically written $\Delta \vdash P$, treating the available resources in Δ as an input to prove a predicate P [10]. In linear logic, as with other substructural logics, we can think of the resources in Δ being *consumed* to prove new resources P. Similarly, linear logic-based type systems typically use a judgment like $\Gamma \mid \Delta \vdash M : x.P$ to prove that term M produces resources P by consuming Δ under variable context Γ [4, 2]. P can mention variables in Γ as well as x, which refers to the value that M evaluates to. In our context, P will associate the permissions produced by M with x.

Such a typing judgment treats Δ as an input and produces 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:

$$\frac{\Gamma \mid \Delta_1 \vdash M_1 : x_1.P \qquad \Gamma, x_1 \mid \Delta_2, P \vdash M_2 : x_2.P_2 \qquad \Delta = \Delta_1, \Delta_2}{\Gamma \mid \Delta \vdash \texttt{let}\ x = M_1\ \texttt{in}\ M_2 : x_2.P_2}$$

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 such as the one shown for `let` deterministic, we take a page from linear logic proof search [9] and add an additional output, Δ', 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 ($P_1 \oplus P_2$) or an internal choice (P_1 & P_2). Typically a backtracking approach is used, where a prover attempts to prove one predicate and, if that fails, tries to prove the other predicate instead. That would be onerous when successively proving pre-conditions for program instructions, where the last instruction may cause backtracking to the first one. Instead of backtracking, we will therefore carry the 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 [13], we extend our proof search technique by associating constraint implications $C_0 \rightarrow C$ (abbreviated I) with our linear context Δ. 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 will be made when eliminating quantified fractions. We will refer to $\Delta|I$ as an *atomic context*.

The remainder of this section contains a detailed description of our permission inference system.

2.1 Syntax

The syntax for the permission inference system is summarized in Figure 1. It is based on Featherweight Java [11], but for simplicity we are not keeping track of object types and instead assume that conventional typechecking has already ensured type safety. Programs Λ are simply lists of methods (suitably renamed to avoid name clashes) and class constructors with declared pre- and post-condition predicates. Expressions M 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 [15, 4].

Expression types E bind a variable representing the value computed by the typed expression in the scope of 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. Permissions can have zero (0) fractions if they are never used for satisfying any predicates; "real" permissions always need a fraction that is strictly greater than zero.

Proof contexts Ψ are built up from atomic contexts $\Delta \mid I$. 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 are available.

2.2 Typechecking

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

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

This judgment reads, "in context Ψ, expression M (with free variables defined in Γ) will produce a value x with predicate P as well as a new context Ψ' for checking the remainder of the program". For example, the typing rule for `let` expressions presented at the beginning of section 2 now appears as T-Let in figure 2: Ψ 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 Ψ'. Ψ' is in turn used to prove the second subexpression, producing Ψ'' which is returned to the next part of the program.

The remaining typing rules prove method and constructor pre-conditions with the linear logic proof judgement presented in the next section. Field access requires a permission for the accessed object, which poses no additional problems for fraction inference and is hence omitted in this paper [details in 3].

Programs	$\Lambda ::= \bullet$	
	$\mid \Lambda, m(x_1, \ldots, x_n) : P \multimap E = M$	*method*
	$\mid \Lambda, c(x_1, \ldots, x_n) : P \multimap x.1 \cdot x$	*constructor*
Expressions	$M ::= x_0.m(x_1, \ldots x_n)$	*call*
	$\mid$ `new` $c(x_1, \ldots, x_n)$	*construction*
	$\mid$ `let` $x = M_1$ `in` M_2	*sequence*
Expr. types	$E ::= x.P$	
Predicates	$P ::= k \cdot x$	*atom*
	$\mid P_1 \otimes P_2$	*separate conjunction*
	$\mid P_1 \;\&\; P_2$	*internal choice*
	$\mid P_1 \oplus P_2$	*external choice*
	$\mid \forall z.P \mid \exists z.P$	*quantification*
Fractions	$k ::= z$	*unknown fraction*
	$\mid \mathbf{Z}$	*fraction variable*
	$\mid 1 \mid 0 \mid \ldots$	*constants*
Var. contexts	$\Gamma ::= \bullet \mid \Gamma, x$	
Proof contexts	$\Psi ::= \Delta \mid I$	*atom*
	$\mid \Psi_1 \;\&\; \Psi_2$	*choice*
	$\mid \Psi_1 \oplus \Psi_2$	*all*
Implication	$I ::= C_0 \rightarrow C$	
Permission set	$\Delta ::= \bullet \mid \Delta, P$	
Constraints	$C ::= \top \mid \bot$	*true, false*
	$\mid F \mid C_1 \wedge C_2$	*formula, conjunction*
Formulae	$F ::= k \doteq k_1 + k_2 \mid 0 < k$	*see Sect. 2.3*

Program vars.	$x, y ::=$ `this` $\mid \ldots$	
Fraction vars.	z	
Class names	$c ::=$ `Object` $\mid \ldots$	
Node names	$n ::=$ *alive* $\mid \ldots$	
Method names	m	

Fig. 1. Syntax for permission inference system

$$\text{T-CALL}\quad \frac{\mathsf{mtype}(m) = \forall x_0, x_1, \ldots, x_n.P \multimap E \qquad \Psi \vdash P \Rightarrow \Psi'}{\Gamma \mid \Psi \vdash x_0.m(x_1, \ldots, x_n) : E \Rightarrow \Psi'}$$

$$\text{T-NEW}\quad \frac{\mathsf{init}(c) = \forall x_1, \ldots, x_n.P \multimap E \qquad \Psi \vdash P \Rightarrow \Psi'}{\Gamma \mid \Psi \vdash \texttt{new}\ c(x_1, \ldots, x_n) : E \Rightarrow \Psi'}$$

$$\text{T-LET}\quad \frac{\Gamma \mid \Psi \vdash M_1 : x.P \Rightarrow \Psi' \qquad \Gamma, x \mid \Psi', P \vdash M_2 : E \Rightarrow \Psi''}{\Gamma \mid \Psi \vdash \texttt{let}\ x = M_1\ \texttt{in}\ M_2 : E \Rightarrow \Psi''}$$

$$\frac{m(x_1, \ldots, x_n) : P \multimap E = M \in \Lambda}{\mathsf{mtype}(m) = \forall \texttt{this}, x_1, \ldots, x_n.P \multimap E} \qquad \frac{c(x_1, \ldots, x_n) : P \multimap E \in \Lambda}{\mathsf{init}(c) = \forall x_1, \ldots, x_n.P \multimap E}$$

Fig. 2. Typechecking rules and helper judgments for permission inference

2.3 Proof Rules

The judgment for proving a predicate P from a given context Ψ is the following:

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

This judgment will be presented in several steps. We first discuss the rules for deriving constraints from splitting and merging individual permissions. Then, rules for proving predicates from atomic contexts $\Delta \mid I$ are presented. Finally, we show how complex contexts Ψ are broken down into such atomic contexts.

Atom Rules. We need three rules for dealing with atomic permissions, one for splitting, one for merging, and one for trivial failure (Figure 3). The basic idea behind them is to delay splits until they are needed for proving a permission, while eagly merging permissions to the same variable.

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" wherever they are used, as explained above. Let us see how these constraints are generated by the atom rules in Figure 3.

SPLIT proves a permission for some variable x if there is a permission for that variable in the context. 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".

$$\textsc{Split}\quad \frac{C' = k \doteq k' + \mathbf{Z}'' \wedge 0 < k \wedge 0 < k' \qquad \mathbf{Z}''\ \mathit{fresh}}{\Delta, k \cdot x \mid C_0 \to C \vdash k' \cdot x \Rightarrow \Delta, \mathbf{Z}'' \cdot x \mid C_0 \to (C \wedge C')}$$

$$\textsc{Merge}\quad \frac{\Delta, \mathbf{Z}'' \cdot x \mid C_0 \to (C \wedge \mathbf{Z}'' \doteq k + k') \vdash P \Rightarrow \Psi \qquad \mathbf{Z}''\ \mathit{fresh}}{\Delta, k \cdot x, k' \cdot x \mid C_0 \to C \vdash P \Rightarrow \Psi}$$

$$\textsc{Fail}\quad \frac{\mathit{no\ permission\ for}\ x\ \mathit{in}\ \Delta}{\Delta \mid C_0 \to C \vdash k \cdot x \Rightarrow \Delta \mid C_0 \to (C \wedge \bot)}$$

$$\otimes\ \mathrm{L}\quad \frac{\Delta, P_1, P_2 \mid I \vdash P \Rightarrow \Psi}{\Delta, P_1 \otimes P_2 \mid I \vdash P \Rightarrow \Psi} \qquad \otimes\ \mathrm{R}\quad \frac{\Delta \mid I \vdash P_1 \Rightarrow \Psi_1 \qquad \Psi_1 \vdash P_2 \Rightarrow \Psi_2}{\Delta \mid I \vdash P_1 \otimes P_2 \Rightarrow \Psi_2}$$

$$\oplus\ \mathrm{L}\quad \frac{\Delta, P_1 \mid I \vdash P \Rightarrow \Psi_1 \qquad \Delta, P_2 \mid I \vdash P \Rightarrow \Psi_2}{\Delta, P_1 \oplus P_2 \mid I \vdash P \Rightarrow \Psi_1 \oplus \Psi_2} \qquad \oplus\ \mathrm{R}\quad \frac{\Delta \mid I \vdash P_1 \Rightarrow \Psi_1 \qquad \Delta \mid I \vdash P_2 \Rightarrow \Psi_2}{\Delta \mid I \vdash P_1 \oplus P_2 \Rightarrow \Psi_1 \,\&\, \Psi_2}$$

$$\&\ \mathrm{L}\quad \frac{\Delta, P_1 \mid I \vdash P \Rightarrow \Psi_1 \qquad \Delta, P_1 \mid I \vdash P \Rightarrow \Psi_2}{\Delta, P_1 \,\&\, P_2 \mid I \vdash P \Rightarrow \Psi_1 \,\&\, \Psi_2} \qquad \&\ \mathrm{R}\quad \frac{\Delta \mid I \vdash P_1 \Rightarrow \Psi_1 \qquad \Delta \mid I \vdash P_2 \Rightarrow \Psi_2}{\Delta \mid I \vdash P_1 \,\&\, P_2 \Rightarrow \Psi_1 \oplus \Psi_2}$$

$$\forall\ \mathrm{L}\quad \frac{\mathbf{Z}\ \mathit{fresh} \qquad \Delta, [\mathbf{Z}/z]P' \mid C_0 \to (C \wedge \mathbf{Z} > 0) \vdash P \Rightarrow \Psi'}{\Delta, \forall z.P' \mid C_0 \to C \vdash P \Rightarrow \Psi'} \qquad \forall\ \mathrm{R}\quad \frac{\Delta \mid (C_0 \wedge z > 0) \to C \vdash P \Rightarrow \Psi'}{\Delta \mid C_0 \to C \vdash \forall z.P \Rightarrow \Psi'}$$

$$\exists\ \mathrm{L}\quad \frac{\Delta, P' \mid (C_0 \wedge z > 0) \to C \vdash P \Rightarrow \Psi'}{\Delta, \exists z.P' \mid C_0 \to C \vdash P \Rightarrow \Psi'} \qquad \exists\ \mathrm{R}\quad \frac{\mathbf{Z}\ \mathit{fresh} \qquad \Delta \mid C_0 \to (C \wedge \mathbf{Z} > 0) \vdash [\mathbf{Z}/z]P \Rightarrow \Psi'}{\Delta \mid C_0 \to C \vdash \exists z.P \Rightarrow \Psi'}$$

Fig. 3. Proof rules for linear logic formulae

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 x explicit: the rule inserts "false" ($\bot$) into the constraints if no permission for a variable x is available whatsoever. Another way of thinking about this rule is that any variable not explicitly mentioned in Δ implicitly has a zero ($0 \cdot x$) fraction associated with it. Proving a permission for such a variable with SPLIT would then result in the unsatisfiable constraint $0 < 0$.

Proving Predicates. Figure 3 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 suitably alpha-converted to not capture existing parameters. We ensure that quantifiers contain "real" fractions by adding suitable constraints where quantifiers are introduced. When quantifiers are used ($\forall L$ and $\exists R$) we introduce logic variables to find a suitable instantiation.

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.

Discussion. We have not included rules for solving linear implication ($\multimap$). In practice we have found that full support for linear implication was made unnecessary by specialized support for dynamic state tests [6]. Full support for implication would require the standard use of focusing [1].

Notice that the & and $\oplus$ rules as well as the $\forall$ and $\exists$ rules are dual to each other, as they should be.

$$\textsc{Ctx-}\& \quad \frac{\Psi_1 \vdash P \Rightarrow \Psi_1' \qquad \Psi_2 \vdash P \Rightarrow \Psi_2'}{\Psi_1 \mathbin{\&} \Psi_2 \vdash P \Rightarrow \Psi_1' \mathbin{\&} \Psi_2'} \qquad \textsc{Ctx-}\oplus \quad \frac{\Psi_1 \vdash P \Rightarrow \Psi_1' \qquad \Psi_2 \vdash P \Rightarrow \Psi_2'}{\Psi_1 \oplus \Psi_2 \vdash P \Rightarrow \Psi_1' \oplus \Psi_2'}$$

Fig. 4. Context proof rules

Context Rules. The rules for breaking down "choice" and "all" contexts in order to apply the rules in Figure 3 are shown in Figure 4. The rules stipulate that predicates are proven separately for compound contexts and put back together into a new compound context afterwards.

2.4 Solving Constraints

The collection of constraints during typechecking allows us to effectively accumulate information about all the permissions needed in a piece of code. But a program should only typecheck if constraints are satisfiable.

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 outermost expressions, which in our case are method bodies. The relevant rule for ensuring that a method body is well-defined is as shown in Figure 5. The rules 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 $\bot$) eagerly.

Constraint Satisfiability. Constraints contain "unknown" variables, written z, and *logic variables* $\mathbf{Z}$. The quantifier proof rules ($\forall$, $\exists$) of Figure 3 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 an unspecified but particular 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

$$\textsc{T-Meth}\ \frac{\texttt{this}, x_1, \ldots, x_n \mid (P \mid \top \rightarrow \top) \vdash M : E \Rightarrow \Psi \qquad \models \Psi}{m(x_1, \ldots, x_n) : P \multimap E = M \in \Lambda}$$

$$\textsc{Sat-Atom}\ \frac{C_0 \rightarrow C}{\models \Delta \mid C_0 \rightarrow C} \qquad \textsc{Sat-\&-L}\ \frac{\models \Psi_1}{\models \Psi_1\ \&\ \Psi_2} \qquad \textsc{Sat-\&-R}\ \frac{\models \Psi_2}{\models \Psi_1\ \&\ \Psi_2} \qquad \textsc{Sat-}\oplus\ \frac{\models \Psi_1 \qquad \models \Psi_2}{\models \Psi_1 \oplus \Psi_2}$$

Fig. 5. Well-defined methods

notational convenience. If we make these quantifications explicit then determining constraint satisfiability $C_0 \rightarrow C$ is equivalent to deciding the following logical formula:

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

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 treated the same way in (1), 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.[1] In a nutshell, Fourier-Motzkin can be used to eliminate a given variable from a conjunction of linear constraints by re-writing the input formula into an equivalent formula that does not include the eliminated variable [19]. 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. This final formula is decidable because equality and inequality between rational constants (such as $0 < 1$ or $1 = 1$) 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 [18] discusses other algorithms for eliminating linear quantifiers which are applicable to our constraint formulae as well. We leave the evaluation of these algorithms to future work.[2]

Example. As an example, we can verify the producer-consumer from the introduction as shown in Figure 6. We assume that the queue's constructor returns a unique permission ($1 \cdot queue$) and the worker thread's constructor takes and does not return *some* fractional ($\exists z.z \cdot queue$) permission to the queue, which will allow each worker thread and the producer to access 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.

[1] Linear programming [19] cannot decide alternatingly quantified formulae.

[2] In practice, our tool's performance ranges around 60ms per method, which includes conventional typechecking, constraint collection, and constraint resolution[3].

Our prototype implementation uses Java annotations to declare method signatures. For instance, the shown `@ResultShare` annotation promises to return a share permission (with *some* fraction). share permissions are tracked exactly like immutable ones but permit modifications of the referenced object [4], and we assume that each `Worker` also captures such a permission. Therefore, the constraints generated for the return value are similar to those shown in the loop body.

3 Flow Analysis for Local Permission Inference

Our prototype tool, Plural[3], is a typestate checker for Java programs. It depends on permission information and implements the inference algorithm presented in the preceding section. Plural is a modular, intra-procedural analysis that uses annotations to declare permissions consumed and returned by methods (see Figure 6).

Plural tracks the permissions available for each object at each program point. Like any dataflow analysis, this information is structured as a lattice. In section 2 we discussed how permissions can be inferred automatically 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 contexts, which will be discussed below. Plural's transfer functions essentially implement the permission inference rules described in section 2.

3.1 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 anal-

[3] `http://code.google.com/p/pluralism/`

```
@ResultShare public static Blocking_queue createConsumers(int number) {
  Blocking_queue queue = new Blocking_queue();
  // 1 * queue | 0 < z --> true
  for (int i = 0; i < number; i++) {
    // z * queue | 0 < z --> true
    (new Worker(queue)).start();
    // Z * queue | 0 < z --> z = Z + Z' ^ 0 < Z ^ 0 < Z'
  }
  // z * queue | 0 < z --> true
  return queue;
  // Z1 * queue | 0 < z --> z = Z1 + Z1' ^ 0 < Z1 ^ 0 < Z1'
}
```

Fig. 6. Constraints generated for a producer-consumer example

ysis 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. 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, as described below. However, we *never* used such formulae in our case studies, so this issue has had no consequences in practice to date.

3.2 Comparing and Joining Permission Tuples

Tuples are compared and joined pointwise. Permissions for the same object are compared using their fractions as follows:

1. pairwise equal fractions are equal.
2. logic variable (**Z**) or literal fractions (such as 1) are more precise than unknown fractions (z) from existential quantifiers. That is, knowing a particular fractional value, or knowing that we can set the value to whatever we want, is a stronger assumption than assuming that a value is unknown.
3. all other fractions are incomparable.

A pair of incomparable permissions is approximated by generating a new permission with fresh *unknown* fraction, z' (and in contrast to the fresh variable fractions, **Z'** introduced by the permission inference). These unknown fractions conservatively approximate variable and literal fractions, as mentioned above.

As shown in Figure 6, introducing unknown fractions to join incomparable permissions allows permissions to be consumed in loops, which was not previously possible [see 21].

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.

3.3 Composing Tuples

Plural's tuple lattice supports composite tuples corresponding to the composite *choice* and *all* contexts from section 2. Formally, we define the elements A of this lattice as follows:

$$\begin{array}{rll} A, B, C ::= & T & \textit{atomic tuple} \\ | & A \;\&\; B & \textit{choice element} \\ | & A \oplus B & \textit{all element} \\ | & \top & \textit{top} \\ | & \bot & \textit{bottom} \end{array}$$

Comparing the elements of this lattice is straightforward, given the comparison $T_1 \sqsubseteq T_2$ for individual tuples we described in section 3.2.

$$\frac{A \sqsubseteq C}{A \;\&\; B \sqsubseteq C} \qquad \frac{B \sqsubseteq C}{A \;\&\; B \sqsubseteq C} \qquad \frac{A \sqsubseteq B \qquad A \sqsubseteq C}{A \sqsubseteq B \;\&\; C} \qquad \frac{A \sqsubseteq C \qquad B \sqsubseteq C}{A \oplus B \sqsubseteq C}$$

$$\frac{A \sqsubseteq B}{A \sqsubseteq B \oplus C} \qquad \frac{A \sqsubseteq C}{A \sqsubseteq B \oplus C} \qquad \frac{}{A \sqsubseteq \top} \qquad \frac{}{\bot \sqsubseteq A}$$

This lattice allows reasoning about linear logic predicates that include internal (&) and external choice ($\oplus$) operators, but it has infinite height. It can for instance be finitized by joining external choices where they appear [3]. The resulting lattice over only &-elements is finite because lattice elements can only contain a finite number of incomparable tuples. As mentioned, we never used internal or external choices in our case studies [6], so the precision impact of this approximation is unknown.

4 Related Work

Fractional permissions were proposed by Boyland for avoiding data races in concurrent programs [8] and have since been used in a variety of contexts.

We previously proposed type systems for sound and modular checking of typestate-based protocols that extend Boyland's permissions to allow a great deal of flexibility in how objects can be aliased [4, 2]. In particular, we introduced new kinds of permissions, the notion of state guarantees, and used linear logic to build predicates from individual permissions.

This paper contributes a permission inference system for a fragment of linear logic predicates that is polymorphic with respect to fractions and describes the dataflow analysis used in our protocol checking tool, Plural. We previously mentioned Plural [2, 6]; this paper gives the first detailed account of its dataflow analysis and underlying inference theory.

Terauchi and Aiken [22] previously proposed a whole-program fraction inference system for checking data races in concurrent programs whose implementation is based on a linear programming engine [21]. In their work, control flow

merges introduce new fraction variables that are forced to be at most as large as the incoming fractions. Function signatures are also inferred with fraction variables. Constraints are hence 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 dataflow analysis, which is polynomial in the method size in the worst case [17, chapter 6] and requires annotations for permissions passed into and out of methods. An advantage of this approach is that permissions can be consumed in loops, which is not possible in Terauchi [21, see section 5]. Figure 6 shows such an example. We are also able to use universal quantification in function signatures, which is more modular than inferring concrete fractions for signatures [14]. These differences are due to our support for polymorphic fractions, which are not available in previous work.

Leino and Müller [14] encode Boyland's fractions using integers representing percentages between 0 and 100 as well as "infinitesimal" fractions in first-order logic. Loop invariants have to be provided by hand, while our implementation infers loop invariants. Unlike with polymorphic fractions, it appears that the (duplicable) infinitesimal fractions cannot be used to borrow or otherwise temporarily alias objects. Consequently, this unusual encoding will require developers to consistently use concrete percentage values in most of their method pre-conditions and loop invariants.

Bornat et al. [7] combined Boyland's fractional permissions with separation logic, but their approach is intended for hand-written proofs about program correctness. Boyland's ongoing work on avoiding data races unfortunately elides the details of their fraction inference implementation [23]. VeriFast is an automated proof checker for separation logic that to our knowledge supports Boyland's fractional permissions [12], but the details of the implementation are again unknown. VeriFast seems to require significant developer input.

5 Conclusions

Fractional permissions are a very promising mechanism for reasoning about programs with aliasing. This paper contributes a permission inference algorithm for fraction-polymorphic permissions that forms the basis of a comprehensive and modular dataflow analysis for tracking permissions in imperative programs. Our prototype implementation, Plural, uses this analysis for sound automated reasoning about typestate-based protocols, but the analysis is generic and we therefore hope that it will allow others to automate their applications of fractional permissions as well.

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

Bibliography

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

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

[3] K. Bierhoff. API protocol compliance in object-oriented software. Technical Report CMU-ISR-09-108, Carnegie Mellon University, Apr. 2009.

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

[5] K. Bierhoff and J. Aldrich. Permissions to specify the composite design pattern. In *7th International Workshop on Specification and Verification of Component-Based Systems*, Nov. 2008.

[6] 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. Springer, July 2009.

[7] 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.

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

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

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

[11] 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.

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

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

[14] 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.

[15] Y. Mandelbaum, D. Walker, and R. Harper. An effective theory of type refinements. In *ACM International Conference on Functional Programming*, pages 213–225, 2003.

[16] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.

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

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

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

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

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

[22] 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.

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