Proving the Correctness of Fractional Permissions for a Java-like Kernel Language

John Tang Boyland
Department of EE & Computer Science
University of Wisconsin-Milwaukee
Wisconsin, USA
boyland@cs.uwm.edu

Chao (Chris) Sun
Department of EE & Computer Science
University of Wisconsin-Milwaukee
Wisconsin, USA
csun@uwm.edu

ABSTRACT

We announce mechanical proofs of soundness for a type system using fractional permissions with nesting for single-threaded programs in a kernel language, and for a simple non-null system that piggy-backs on the first system. The proofs are all done using Twelf. The (concurrent) kernel language and fractional permission system are pre-existing. The permission type system links them, and serves as a proof foundation for the non-null system. Along the way, we describe our experiences using Twelf.

1. INTRODUCTION

A system of fractional permissions with nesting has been proposed as a semantic foundation for a large variety of alias-based program annotations such as uniqueness, read-only, ownership, mutex protected state and so on [7]. This paper shows incremental progress in two areas (1) proving the soundness of fractional permissions of a Java-like kernel language (albeit without concurrency) and (2) proving the soundness of a simple non-null type system by reduction to fractional permissions.

2. FRACTIONAL PERMISSIONS

We summarize our system of fractional permissions with nesting. We also explain how the system is realized in Twelf.

2.1 Basics

Figure 1 describes the syntax of our permission logic (a) and also gives its realization in Twelf (b).

2.1.1 Permission Syntax

On the left in the figure, we give the (textual) syntax of permissions. A *reference* (ρ) is a pointer to an object with fields. The meta-variable o is used to refer to a literal pointer value (where zero 0 is used to refer to the literal null pointer). The meta-variable r is used to refer to a pointer variable. An example is seen in the "existential permission" below.

Fractions are used to distinguish write access (full = 1) from read access (less than 1). We use q to stand for literal positive fraction values, including $\frac{1}{2}$. The meta-variable z is used to refer to a bound fraction variable.

Permissions give access to (potentially) mutable state in a program. The "interesting" permissions are the unit permissions (described below). At the permission level, we can combine two permissions or scale a permission by a fraction. The empty permission ($\emptyset$) gives no access at all, and is the identity of the combination operator (written + although it is close in semantics to the separation logic [16] $\star$ operator). Syntactic equivalence (not shown) permits commutativity, associativity and distributivity so that, for example, $\Pi \equiv \frac{1}{3}\Pi + \frac{2}{3}\Pi \equiv \frac{1}{3}(\Pi + 2\Pi)$.

Unit permissions include the fundamental field permission

$$o.f \to o'$$

which gives write access to field f of object o whose value is currently o'.[1] Assigning the field will change the permission accordingly. Permissions are linear; you cannot duplicate a permission, since then one might modify the value and the other permissions would become invalid. However, they can be "split" into fractions; $\frac{1}{2}(o.f \to o')$ can be used to read, but not modify the field. Splitting can be carried into indefinitely small fractions. Any fraction (no matter how small, zero is not permitted) gives read access.

An *existential permission* is used when we don't know the literal value of a field, but we know something *about* the value. The existential is restricted to exactly this form to ensure that the result will have a "precise" semantics [13, page 9]; the existentially bound variable is witnessed by the value of the field $o.f$.

A *conditional permission* gives one of two permissions depending on the value of a "formula" (see below). Since formulae have fixed values, conditionals are often used inside existentials. The permission

$$\exists r \cdot (o.f \to r + (r = 0\ ?\ \emptyset : \Pi))$$

gives access to the field $o.f$ together with information that if the field is not null, we have additional permissions Π. (If the field is null, we only have $\emptyset$ which is useless.)

In $\Pi_1 \multimap \Pi_2$, the Π_2 permission is encumbered by Π_1; the former cannot be used until the latter is supplied. Implications allow for the temporary breaking of invariants; the permission Π_2 represents the invariant, and Π_1 represents what situation must be restored before the invariant is

[1] We do not model so-called "primitive" values.

	`vark : type.`	`field = nat.`
	`objectk : vark.`	`predtype : type.`
	`fractionk : vark.`	
	`permissionk : vark.`	`predtype/0 : predtype.`
	`upermk : vark.`	`predtype/+ : vark -> predtype -> predtype.`
	`formulak : vark.`	
		`predicate : predtype -> type.`
	`gterm : vark -> type.`	`predargs : predtype -> type.`
reference:	$\rho ::=$	`object = gterm objectk.`
variable	r	
literal	o	`object/ : nat -> object.`
fraction:	$\xi ::=$	`fraction = gterm fractionk.`
variable	z	
literal	q	`fraction/ : rat -> fraction.`
permission:	$\Pi ::=$	`permission = gterm permissionk.`
variable	v	
empty	$\emptyset$	`empty : permission.`
combined	$\Pi + \Pi$	`combine : permission -> permission -> permission.`
scaled	$\xi\Pi$	`scale : fraction -> permission -> permission.`
unit	π	`unitperm : uperm -> permission.`
unit perm:	$\pi ::=$	`uperm = gterm upermk.`
field	$o.f \to \rho$	`basic : object -> field -> object -> uperm.`
formula	Γ	`nonlinear : formula -> uperm.`
existential	$\exists r \cdot (o.f \to r + \Pi)$	`precise-exists : object -> field -> (object -> permission) -> uperm.`
conditional	$\Gamma \ ? \ \Pi : \Pi$	`conditional : formula -> permission -> permission -> uperm.`
implication	$\Pi \multimap \Pi$	`encumbered : permission -> permission -> uperm.`
formula:	$\Gamma ::=$	`formula = gterm formulak.`
true	$\top$	`t : formula.`
negation	$\neg\Gamma$	`neg : formula -> formula.`
conjunction	$\Gamma \wedge \Gamma$	`conj : formula -> formula -> formula.`
existential	$\exists x \cdot \Gamma$	`exists* : {K} ((gterm K) -> formula) -> formula.`
predicate call	$p(\overline{X})$	`predcall* : {N} predicate N -> predargs N -> formula.`
comparison	$\rho = \rho$	`objequal : object -> object -> formula.`
nesting	$\Pi \prec o.f$	`nested : permission -> object -> field -> formula.`

(a) Permission Syntax

$x ::= r \mid z \mid v$	*any variable*
$X ::= \rho \mid \xi \mid \Pi$	*any term*
$\Delta ::= \{\overline{x}\}$	*set of variables*
$P ::= \left\{\overline{p(\overline{x}) = \Gamma}\right\}$	*predicate def'ns*
$\sigma ::= \left[\overline{x \mapsto X}\right]$	*substitution*

```
predargs/0 : predargs (predtype/0).
predargs/+ : gterm K -> predargs N -> predargs (predtype/+ K N).

predicate/0 : formula -> predicate (predtype/0).
predicate/+ : (gterm K -> predicate N) -> predicate (predtype/+ K N).
predicate/Y : (predicate N -> predicate N) -> predicate N.
```

(b) Twelf Realization

Figure 1: Syntax of permissions and formulae.

recovered. For example, assuming $o.\text{All} \to 0$ represents the object invariant for o, then

$$(\exists r \cdot (o.f \to r + (r = 0 \,?\, \emptyset : r.\text{All}))) \multimap\!\!+ o.\text{All} \to 0$$

means that the invariant can be re-established once we have the existential permission. In particular, implications are used when a "nested" permission is "carved out" of the field it is nested in (see below).

Formulae (Γ) represent "facts," that remain true (or false). Nonlinear reasoning can be used on facts. Formulae can be negated, conjoined and bound by existentials using standard logical notation. Formula abstractions, (named) *predicates*, take arbitrary arguments and return a truth value.

The two primitive formulae are object equality (which compares two pointer values for identity) and "nesting." A permission Π can be (irrevocably) granted to a field $o.f$. Once that action has taken place, the formula $\Pi \prec o.f$ is true. The nested permission can be accessed when one has access to the field it was nested in.

Nesting is used for abstraction. By convention, the states of all (other) fields are nested in a special "All" field and so the permission $o.\text{All} \to 0$ gives one access to the entire state of the object. The fields are nested with information about their values, thus

$$(\exists r \cdot (o.f \to r + (r = 0 \,?\, \emptyset : r.\text{All}))) \prec o.\text{All}$$

says that the f field is nested along with the permission to access the object it points to (unless null). If one has the permission $o.\text{All} \to 0$, one can *carve out* the nested (existential) permission that says that f is accompanied by the permission to the object it points to and have

$$(\exists r \ldots) + ((\exists r \ldots) \multimap\!\!+ o.\text{All} \to 0)$$

which gives use the ability to update $o.f$ and do other actions, but the original permission $o.\text{All} \to 0$ can only be restored when one surrenders the required existential permission.

Using fractions, $\frac{1}{2}(o.\text{All} \to 0)$ gives one *read* access to all this state, without needing to know precisely what fields the object has. Using another conventional field, "Owned," nesting can also model ownership, if $(o.\text{All} \to 0) \prec o'.\text{Owned}$, then we say that o' *owns* o.

In our textual formalization (albeit, not in Twelf), we use the notion of an explicit substitution, σ. Substitutions are typed: $\sigma : \Delta_1 \to \Delta_2$ where Δ_1 is the set of variables that are mapped, and Δ_2 is the set of (free) variables used in the range of the map. A substitution $\sigma : \Delta \to \emptyset$ thus maps all its variables to closed terms.

The figure defines a meta-variable v for permission variables, but no meta-variables for unit permission variables or formula variables. We have found no use for them, but the Twelf formalization permits variables for any of the five syntactic kinds.

2.1.2 Twelf Realization

All our proofs have been mechanized in Twelf [15]. In order that we can discuss some of the challenges we faced in this task, we give the Twelf realization of the permission syntax on the right side of Figure 1. We do not assume familiarity with Twelf.

Twelf includes declaration of new *types* and *constructors*, and also definitions of named abbreviations. At the top of the page,[2] there is a declaration of a new type `vark` for the five syntactic kinds (hence "`k`") used to define permissions. Then five constructors are defined for that type. In Twelf, every new type or constructor is, by definition, different from any other type or constructor. Thus the `vark` type has exactly five values, the five constructors given. These five are used to mark the five different syntactic kinds (references, fractions, etc).

Types can be dependent (or "indexed") on values, and thus the (dependent) type `gterm` can be used in five different ways. The abbreviation `object = gterm objectk` gives a shorthand to refer to the first such kind of "generalized term."

On the top right, we see first a simple abbreviation (we use natural numbers for field identifiers for simplicity). Next there is a definition of a more complex type `predtype` for indexing formal and actual parameters. In Twelf, most characters (such as `/` and `+`) can be used in identifiers. Thus `predtype/+` is just an identifier. There is a convention that constructors for a type T are often given names such as T/x for some string x. A predicate type can be empty (`predtype/0`) or be constructed from a variable kind and a predicate type (using the curried constructor `predtype/+`). Thus there is one type of predicate with no parameters, five types for predicates of one parameter, 25 if two etc.

The predicate type is used to index the types `predicate` and `predargs` where the latter is the syntactic type of actual parameters. Below, (in the definition of `predcall`) we see that the type of the predicate must match the arguments.

In the body of the right-hand side of the figure, we give the Twelf constructors corresponding to each kind of syntax. We use HOAS (explained below) and thus have no need for an explicit syntax of variables, but we do need a syntax for literal references and use the constructor `object/` to construct an object from a natural number. For fractions, we have a similar convention, where `rat` is the type of positive rational numbers.

For each syntactic kind, we use an abbreviation (`object`, `fraction`, `permission`, etc.) for clarity and conciseness. The constructors `empty`, `combine`, `scale` and `unitperm` indicate the four ways permissions can be constructed.

The constructor `precise-exists` exhibits *higher-order abstract syntax* (HOAS) in which binding in the host language (Twelf) is used to represent binding in the syntax being described. The last argument of the curried constructor is of type `object -> permission`, a function that given the value of the object returns the permission (which may use the parameter object). HOAS means we don't need to worry about modeling identifiers, substitution or alpha-conversion; all these are handled by Twelf.

Further down the page, the constructor `exists*` features both HOAS and indexing: the parameter can be from any of the five syntactic kinds. Next, we have `predcall*` which uses indexing to ensure that the actual parameters match the syntactic kinds of the formal parameters.

The bottom of the page shows how actuals (`predargs`) are constructed and the different kinds of predicates. The simplest kind of predicate (`predicate/0`) has no arguments, just the body which is a formula. The next kind `predicate/+` is essentially a lambda expression, and we use HOAS to express the binding. Recursive predicates can be constructed

[2] In the figure, for clarity and layout reasons, we sometimes violate definition-before-use.

with `predicate/Y`. This allows us to have a powerful predicate system without needing named predicates (which our more standard syntax, seen on the left bottom, uses).

2.1.3 Transformation

Permission *transformation* is an implication of the form:

$$\Delta; \Pi \rightsquigarrow \Delta'; \Pi'$$

which (roughly) says that if we have permission Π (with free variables from Δ) it is safe to replace these with Π' with perhaps new free variables (skolemization) or fewer old variables. The ability to remove unused variables was not present in our earlier work [6], but is necessary in order to have loop invariants, as described later. We call the $\Delta; \Pi$ pairs "environments." We have proved [5] that many special cases are valid transformations such as linear *modus ponens*, skolemization of existentials and materialization of an invariant ("carving"). Most importantly, we have proved that any permission may be surrendered and nested in any field of any object:

$$\Delta; \Pi \rightsquigarrow \Delta; (\Pi \prec o.f)$$

3. KERNEL LANGUAGE

3.1 Syntax

In earlier work, we describe a kernel language for Java-style concurrency [3]; Fig. 2 shows the single-threaded subset of this language. As with the syntax of permissions, the Twelf realization is on the right. HOAS is used in the constructors for functions and let-bindings. The `MAP` type constructor (which uses `nat` as the domain of the map and the argument type as the range) is not actually part of Twelf; we use a preprocessor to define Twelf types for finite maps.

The kernel language has object literals, variables, allocation, field reads and writes, let bindings, procedure calls, conditionals and loops. This language has separate syntactic kinds for expressions and for conditions, although since equality tests include expressions, side-effects are possible in conditionals as well as in expressions. It does not include primitive arithmetic or dynamic dispatch, because neither of these affects aliasing or threading. We have a type system for concurrent programs [18], but haven't completed the mechanized proof of type soundness, and thus concurrency is omitted in the present paper.

3.2 Evaluate

Evaluation $((e; \mu) \rightarrow_g (e'; \mu'))$ takes an expression in the context of a memory to another expression and a (perhaps) changed memory. Here g is the "program," a set of procedure definitions.

Ignoring concurrency, evaluation for a thread can get stuck only in one of the following situations:

1. Calling a non-existent procedure;
2. Calling a procedure with the wrong number of parameters;
3. Reading or writing a field on an object not (yet) allocated;
4. Reading or writing a field on an object that does not have it;

In the Twelf realization, we use HOAS for let-bound variables and formal parameters. Procedures are given (not necessarily consecutive) natural number indices, and a program maps procedures to parameterized expressions.

As with the permission system, object identifiers are natural numbers. The memory maps each object identifier to an "object state," a map from field numbers to object identifiers.

4. PERMISSION TYPES

Procedures in programs are assigned procedure types (see Fig. 3); these are used to check if a program is well-typed. A procedure type has universally quantified variables for its input variables (including a distinguished series of object variables for the parameters) in the input permission Π and existentially quantified variables for the output variables (including one distinguished variable for the result value) that may additionally appear in the output permission Π'.

In the Twelf realization, `output exprk` is used as the type of an expression (as explained below); it may include existentially quantified variables. A procedure type can simply be the type of the body (`proctype/base`) if there are no parameters but uses HOAS to refer to the formal parameters (`proctype/arg`). Finally, for those extra, non-argument, variables, we can introduce arbitrary permission (`gterm`) variables in our procedure types (`proctype/forall`). A program type is simply a (finite) map from natural numbers to the procedure type for the procedure with that number.

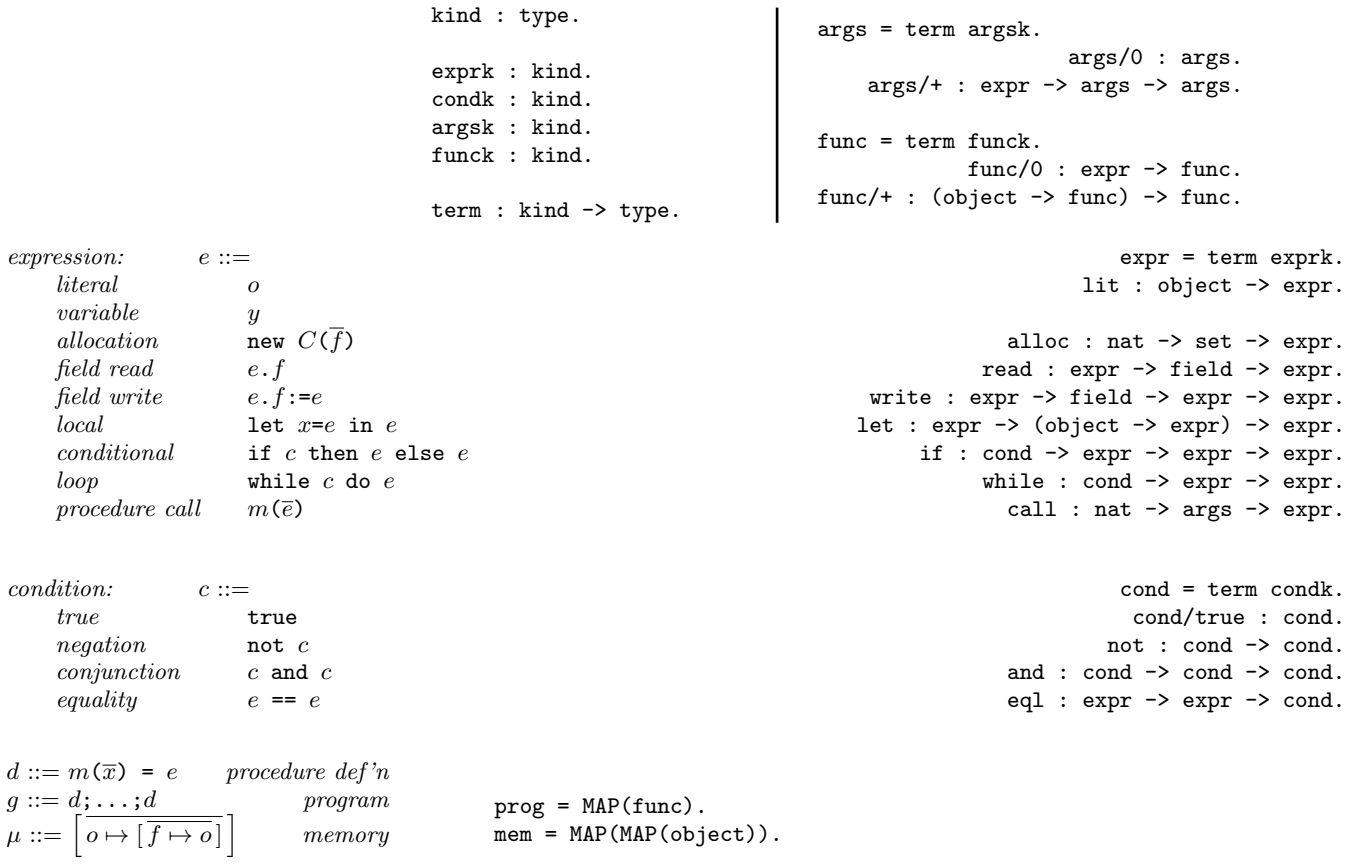
Figure 4 shows a permission type system for the kernel language. The relation $\Pi \vdash_\omega e \Downarrow \rho \dashv \Delta'; \Pi'$ says that assuming program type ω in the environment $E = (\emptyset; \Pi)$, the expression e will (if it terminates) evaluate to an object reference or variable ρ in the new environment $E' = (\Delta'; \Pi')$. We assume the input environment has no variables in order to be closer to the Twelf realization. (Space precludes us from giving the Twelf form of all type rules.)

In a slight abuse of notation, we write $\forall_\Delta D$ as a shorthand for $\forall_{\sigma:\Delta\rightarrow\emptyset}\sigma D$: it means that D should be true no matter what (closed) bindings the variables in Δ have. In Twelf, this is accomplished using a *hypothetical* binding, as explained below.

Rule WRITE, for example, shows that after typing the target object e_1 and new value e_2, we require that the permissions include write access to the field in question and then change the permissions to reflect the write. The output environment has all of the variables picked up while checking the two parts. If the environment always increased its variables, it would be impossible to have a loop invariant. The LOOP rule thus uses transformation to establish an invariant $\Delta; \Pi$ and also to re-establish it after the body has executed each iteration. Conditionals are typed into formulae Γ. Then as can be seen in COND, the value of conditional can be assumed true in the true branch and assumed false in the false branch, and similarly in LOOP.

Figure 5 defines a well-typed program: the body of each procedure can be typed using its procedure type.

For space reasons, we only show part of the Twelf realization of the permission types (see Figure 6). Relations are represented in Twelf as curried type constructors. The `typing` relation combines all the output parts of the textual relation $\ldots \Downarrow \rho \dashv \Delta; \Pi$ into a single type `output exprk`. This is needed since the output object and permissions may use new variables from the output. The `typing` relation is



```
kind : type.

exprk : kind.
condk : kind.
argsk : kind.
funck : kind.

term : kind -> type.
```

```
args = term argsk.
                    args/0 : args.
    args/+ : expr -> args -> args.

func = term funck.
           func/0 : expr -> func.
func/+ : (object -> func) -> func.
```

expression:	$e ::=$	`expr = term exprk.`
literal	o	`lit : object -> expr.`
variable	y	
allocation	`new` $C(\overline{f})$	`alloc : nat -> set -> expr.`
field read	$e.f$	`read : expr -> field -> expr.`
field write	$e.f$`:=`e	`write : expr -> field -> expr -> expr.`
local	`let` x`=`e `in` e	`let : expr -> (object -> expr) -> expr.`
conditional	`if` c `then` e `else` e	`if : cond -> expr -> expr -> expr.`
loop	`while` c `do` e	`while : cond -> expr -> expr.`
procedure call	$m(\overline{e})$	`call : nat -> args -> expr.`

condition:	$c ::=$	`cond = term condk.`
true	`true`	`cond/true : cond.`
negation	`not` c	`not : cond -> cond.`
conjunction	c `and` c	`and : cond -> cond -> cond.`
equality	e `==` e	`eql : expr -> expr -> cond.`

$d ::= m(\overline{x})$ `=` e	*procedure def'n*	
$g ::= d;\ldots;d$	*program*	`prog = MAP(func).`
$\mu ::= \left[\overline{o \mapsto [\overline{f \mapsto o}]}\right]$	*memory*	`mem = MAP(MAP(object)).`

Figure 2: Syntax of Kernel Language (omitting concurrency).

$\alpha ::= \forall_{\overline{r}} \forall_{\Delta}\, \Pi \to \exists_{r_0} \exists_{\Delta'}\, \Pi'$ *procedure type*

$\omega ::= \{\overline{m \mapsto \alpha}\}$ *program type*

```
output : kind -> type.
output/expr : object -> permission -> output exprk.
output/exists : ((gterm V) -> output K) -> output K.
...

proctype : type.
proctype/base : permission -> output exprk -> proctype.
      proctype/arg : (object -> proctype) -> proctype.
 proctype/forall : ((gterm V) -> proctype) -> proctype.

progtype = MAP(proctype).
```

Figure 3: Procedure and program types

LITERAL
$$\Pi \vdash_\omega o \Downarrow o \dashv \emptyset; \Pi$$

ALLOC
$$\frac{}{\Pi \vdash_\omega \texttt{new}\ C(\overline{f}) \Downarrow r \dashv r; \overline{r.f \to 0} + \Pi}$$

LET
$$\frac{\Pi_1 \vdash_\omega e_1 \Downarrow \rho_1 \dashv \Delta_2; \Pi_2 \qquad \forall_{\Delta_2}\ \Pi_2 \vdash_\omega [x \mapsto \rho_1] e_2 \Downarrow \rho' \dashv \Delta'; \Pi'}{\Pi_1 \vdash_\omega \texttt{let}\ x\texttt{=}e_1\ \texttt{in}\ e_2 \Downarrow \rho' \dashv \Delta_2, \Delta'; \Pi'}$$

READ
$$\frac{\Pi \vdash_\omega e \Downarrow \rho \dashv \Delta'; \xi\rho.f \to \rho' + \Pi'}{\Pi \vdash_\omega e\texttt{.}f \Downarrow \rho' \dashv \Delta'; \xi\rho.f \to \rho' + \Pi'}$$

WRITE
$$\frac{\Pi_1 \vdash_\omega e_1 \Downarrow \rho_1 \dashv \Delta_2; \Pi_2 \qquad \forall_{\Delta_2}\ \Pi_2 \vdash_\omega e_2 \Downarrow \rho_2 \dashv \Delta'; \rho_1.f \to \rho' + \Pi'}{\Pi_1 \vdash_\omega e_1\texttt{.}f\texttt{=}e_2 \Downarrow \rho_2 \dashv \Delta_2, \Delta'; \rho_1.f \to \rho_2 + \Pi'}$$

COND
$$\frac{\Pi \vdash_\omega c \Downarrow \Gamma \dashv \Delta_0; \Pi_0 \qquad \forall_{\Delta_0}\ \Gamma + \Pi \vdash_\omega e_1 \Downarrow \rho' \dashv \Delta'; \Pi' \qquad \forall_{\Delta_0}\ \neg\Gamma + \Pi \vdash_\omega e_2 \Downarrow \rho' \dashv \Delta'; \Pi'}{\Pi \vdash_\omega \texttt{if}\ c\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \Downarrow \rho' \dashv \Delta_0, \Delta'; \Pi'}$$

LOOP
$$\frac{\emptyset; \Pi_1 \rightsquigarrow \Delta; \Pi \qquad \forall_\Delta\ \Pi \vdash_\omega c \Downarrow \Gamma \dashv \Delta'; \Pi' \qquad \forall_{\Delta,\Delta'}\ \Gamma + \Pi' \vdash_\omega e \Downarrow \rho \dashv \Delta''; \Pi'' \qquad \Delta, \Delta', \Delta''; \Pi'' \rightsquigarrow \Delta; \Pi}{\Pi_1 \vdash_\omega \texttt{while}\ c\ \texttt{do}\ e \Downarrow 0 \dashv \Delta, \Delta'; \neg\Gamma + \Pi'}$$

CALL
$$\frac{\begin{array}{c}\Pi_0 \vdash_\omega e_1 \Downarrow \rho_1 \dashv \Delta_1; \Pi_1 \\ \forall_{\Delta_1}\ \Pi_1 \vdash_\omega e_2 \Downarrow \rho_2 \dashv \Delta_2; \Pi_2 \\ \vdots \\ \forall_{\Delta_1,\ldots,\Delta_{n-1}}\ \Pi_{n-1} \vdash_\omega e_n \Downarrow \rho_n \dashv \Delta_n; \Pi_n \\ \omega\, m = \forall_{r_1,\ldots,r_n} \forall_\Delta \Pi \to \exists_{r_0} \exists_{\Delta'} \Pi' \qquad \sigma : \{r_1, \ldots, r_n\} \cup \Delta \to \Delta_1, \ldots, \Delta_n \qquad \forall_i\ \sigma r_i = \rho_i \\ \Pi_n = \sigma\Pi + \Pi'' \qquad (\Delta_1, \ldots, \Delta_n) \cap (r_0, \Delta') = \emptyset\end{array}}{\Pi_0 \vdash_\omega m\texttt{(}e_1\texttt{,}\ldots\texttt{,}e_n\texttt{)} \Downarrow r_0 \dashv \Delta_1, \ldots, \Delta_n, r_0, \Delta'; \sigma\Pi' + \Pi''}$$

TRANSFORM
$$\frac{\begin{array}{c}\emptyset; \Pi_1 \rightsquigarrow \Delta_2; \Pi_2 \\ \forall_{\Delta_2}\ \Pi_2 \vdash_\omega e \Downarrow \rho \dashv \Delta_3; \Pi_3 \\ \forall_o\ \Delta_2, \Delta_3; \rho = o + \Pi_3 \rightsquigarrow \Delta_4; \rho' = o + \Pi_4\end{array}}{\Pi_1 \vdash_\omega e \Downarrow \rho' \dashv \Delta_4; \Pi_4}$$

TRUE
$$\Pi \vdash_\omega \texttt{true} \Downarrow \top \dashv \emptyset; \Pi$$

NOT
$$\frac{\Pi \vdash_\omega c \Downarrow \Gamma \dashv \Delta'; \Pi'}{\Pi \vdash_\omega \texttt{not}\ c \Downarrow \neg\Gamma \dashv \Delta'; \Pi'}$$

AND
$$\frac{\Pi_0 \vdash_\omega c_1 \Downarrow \Gamma_1 \dashv \Delta_1; \Pi_1 \qquad \forall_{\Delta_1}\ \Pi_1 \vdash_\omega c_2 \Downarrow \Gamma_2 \dashv \Delta_2; \Pi_2}{\Pi_0 \vdash_\omega c_1\ \texttt{and}\ c_2 \Downarrow \Gamma_1 \wedge \Gamma_2 \dashv \Delta_1, \Delta_2; \Pi_2}$$

EQ
$$\frac{\Pi_0 \vdash_\omega e_1 \Downarrow \rho_1 \dashv \Delta_1; \Pi_1 \qquad \forall_{\Delta_1}\ \Pi_1 \vdash_\omega e_2 \Downarrow \rho_2 \dashv \Delta_2; \Pi_2}{\Pi_0 \vdash_\omega e_1\ \texttt{==}\ e_2 \Downarrow \rho_1 = \rho_2 \dashv \Delta_1, \Delta_2; \Pi_2}$$

Figure 4: Permission Typing Relations $\Pi \vdash_\omega e \Downarrow \rho \dashv \Delta'; \Pi'$ and $\Pi \vdash_\omega c \Downarrow \Gamma \dashv \Delta'; \Pi'$.

$$\frac{\{\overline{r}\}, \Delta, \{r_0\}, \Delta' \text{ pairwise disjoint} \qquad \forall_{\Delta,\overline{r}}\ \left(\Pi \vdash_\omega [\overline{x \mapsto r}]e \Downarrow r_0 \dashv r_0, \Delta'; \Pi'\right)}{\vdash_\omega \texttt{m(}\overline{x}\texttt{)}\ \texttt{=}\ e : \forall_{\overline{r}} \forall_\Delta\, \Pi \to \exists_{r_0} \exists_{\Delta'}\, \Pi'} \qquad \frac{\vdash_\omega \overline{\texttt{m(}\overline{x}\texttt{)}\ \texttt{=}\ e : \omega\, m}}{\vdash_\omega \overline{\texttt{m(}\overline{x}\texttt{)}\ \texttt{=}\ e}\ \text{OK}}$$

Figure 5: Well-Typed Procedures and Programs

```
typing : progtype -> permission -> term K -> output K -> type.

tRead : typing W (combine (scale Q (unitperm (basic O F O'))) Pi) (read (lit O) F)
        (output/expr O' (combine (scale Q (unitperm (basic O F O'))) Pi)).

tRead1 : typing W Pi E Out1 ->
         letTyping W Out1 ([o] (read (lit o) F)) Out2 ->
    typing W Pi (read E F) Out2.


letTyping : progtype -> output exprk -> (object -> term K) -> output K -> type.

letTyping/base : {O} typing W Pi (F O) Out ->
    letTyping W (output/expr O Pi) F Out.

letTyping/exists : ({v} letTyping W (In v) F (Out v)) ->
    letTyping W (output/exists In) F (output/exists Out).
```

Figure 6: Twelf Realization (Partial)

also used to type conditions, arguments and functions, hence the kind (`K`) polymorphism.

The output includes any number of existentially bound permission variables of any kind (`output/exists` in Figure 3) but in the end includes the object resulting from the typing plus any resulting permissions (`output/expr`).

A type inference rule is represented in Twelf as a constructor for the relation:

$$\textsc{Rule}\ \frac{\textit{premise 1} \quad \textit{premise 2}}{\textit{conclusion}} \qquad \texttt{rule} : \textit{premise 1} \to \textit{premise 2} \to \textit{conclusion}.$$

The `tRead` rule shows how to type the reading of the field `F` of a literal object `O`. The resulting output includes the value of the field `O'` as well as the input permissions unchanged. If the expression object is not a literal (see `tRead1`), we first type it to get an output `Out1` and then use the `letTyping` relation to crack open the `output` type. We pass in a function (`[`o`]` ... is used to designate a lambda expression in Twelf) which will be applied to the resulting object (ρ in the textual system).

There are two rules for `letTyping`: if the output has no (more) variables, we type the term resulting from applying the lambda to the output reference (see `letTyping/base`). (The explicit quantification `{O}` seen here is needed for obscure technical reasons.)

The second rule for `letTyping/exists` shows the *hypothetical* binding: the premise of the rule is that for an unknown `v`, we can form a let-typing of the body of the existential. In other words, the let-typing of `In v` must not depend on knowing anything about the value of `v`.

Type soundness says that programs that permission check execute without errors, in particular the four errors listed in Section 3.2. Execution depends on memory, but permission checking depends on permissions. The connection between the two is called "consistency" and depends on a *fractional heap* (a map from (o, f) pairs to pairs of a fraction and an object) and the current nesting situation (written N):

$$\frac{\forall_{(o,f)\mapsto(q,o')\in h}\ \mu\, o\, f = o' \qquad \Pi' = \sum \{\Pi_{o,f} \mid (o,f) \mapsto \Pi_{o,f} \in N, \mu\, o\, f \text{ undefined}\} \qquad h \models_N \Pi + \Pi'}{\Pi \text{ consistent with } \mu \text{ using } N}$$

The first condition states that the fractional heap agrees with the memory where it is defined. The second condition collects all permissions nested into fields not (yet) present in the memory. The third condition uses the fractional heap semantic model of permissions [5, page 22:20].

We didn't realize the need for the second condition until producing the machine-checked proof. The second condition addresses the possibility that someone may nest a permission in a field of an object that has not yet been allocated. This strange situation is a possibility because nesting happens implicitly (unlike in our earlier work [6]). Once allocation actually happens, the pre-existing nesting would cause consistency to fail if we had not kept track of it before. Over all, defining consistency appropriately was the biggest task in proving soundness of the single-threaded system and is the biggest task currently while proving the concurrent system type-safe. Consistency is yet more complex when we need to handle concurrency.

Proving the soundness of the type system (without concurrency) in Twelf took one person (John Boyland) a year, probably about 500 hours of work. The biggest hurdles that were encountered were consistency (as just explained) and the typing of "while." In the latter case, our original type rule was vacuous since it required a permission invariant while adding new variables to the environment. It was impossible to satisfy. In order to prevent such vacuous definitions, we used Twelf to verify the typing of a simple program.

The final realization of the type rule for "while" is

```
tWhile :
    transform (output/expr (object/ z) Pi) Inv ->
    letTyping W Inv ([o] C) Cout ->
    whileTyping W Cout E Out1 Out2 ->
    discard-value Out1 Out' ->
    transform Out' Inv ->
  typing W Pi (while C E) Out2.
```

In this rule, one sees that in the Twelf realization, `transform` applies to entire "outputs," not just a $(\Delta; E)$ pair. Thus we need to "seed" the output with a null pointer at the start, and partway through, we need to explicitly replace the body's result value with null (`discard-value`) so that any variables used only in the (now discarded) value can be removed.

While the proof takes much longer to write than natural language proofs, there is a much greater confidence in correctness. Furthermore, during the proof construction, it often turns out that one has to strengthen an invariant, or add new conditions to a relation (e.g., consistency) requiring a large portion of the proof to be redone. A mechanized system makes it much easier to find the places where changes must be made.

Twelf proofs in the Twelf meta-logic are very different from proofs in most other proof systems which are tactic-based. Instead, in Twelf, for each theorem/lemma, one gives a series of cases that cover all the inputs. Each case then applies theorems in sequence to produce the required conclusion. The proof is human readable and editable with normal text editing. Proving a theorem is thus very much like programming in moded Prolog.

5. PIGGY-BACKING NON-NULL ON PERMISSION TYPES

In this section we show how soundness of a simple non-null type system can be proved via reducing it to fractional permissions. The proofs are done in Twelf and can be checked under version 1.5R3 or later.

Our target language uses the same syntax as the kernel language defined in Sec. 3. However, since the latter is not object-oriented, we designate certain procedures to serve as constructors, in which the first parameter is the object being initialized. Constructors are required to establish the invariant, in particular that non-null fields are indeed not null.

Since non-null type systems are well known [10], we only show some of the typing rules here (Fig. 7). As can be seen, each reference type in the system is augmented with additional information of whether the reference is *not null* or *possibly null*. Following Fähndrich and Leino [10], we denote the former with c^- and latter with c^+, while c is the class identifier. Without extra explanation, we also use c to denote a type that could be either not null or possibly null, instead of representing the class identifier itself. In addition, we use a special `Null` type for the reference whose value is exactly `null` (which we will use 0 to represent).

There are several environments used in the typing rules. A class map C is a map from class identifiers to their field maps, a procedure map M is a map from procedure identifiers to their types, a F is a map from field identifiers to their types, and a E is a map from local variables to their types.

$$\begin{aligned} C &::= \epsilon \mid C, c : F \ (c \text{ is the class identifier}) \\ M &::= \epsilon \mid M, m : (c_1, \ldots, c_n) \to c \\ F &::= \epsilon \mid F, f : c \\ E &::= \epsilon \mid E, x : c \end{aligned}$$

We also use $E(x) = c$ to denote the same meaning as $x : c \in E$, where we always get the first binding. This notation applies *mutatis mutandis* to C, M, and F. Also, in a slight abuse of notation, we use $C(c)(f) = c_f$ to mean $C(c) = F$ and $F(f) = c_f$.

Before going further, there are some details need mentioning. First, in fractional permissions, invariants are established through nesting. In our non-null system, we simply assume every field is `shared` (that is, in terms of ownership, belongs to the world). In permission syntax, suppose a field f of an object o is pointing to another object r, the fact that the permission to access is shared can be written as follow:

$$(\exists r \cdot o.f \to r + p(r)) \prec 0\,.\mathrm{Owned}$$

while p is the class predicate for type of the field f, and 0 is used to encode the "world" object. As mentioned before, the Owned region is used to model object ownership. Note that, here for simplicity, we nest each field permission inside the Owned region directly, instead of in the All region first. This would have no effect on the proof since everything is shared in the system.

Second, for every procedure, the input permission is always $0\,.\mathrm{Owned} \to 0$, and the output permission is the same plus the permission for the return value. Using terms of effects annotations [7], the input permission can be expressed as "`writes shared`", which grants the annotated method privilage to write fields that are shared.

Since the system does not have inheritance, we do not need to consider the full range of the "rawness" problem [10]. But to avoid leaking a partially initialized object, constructors in the system are syntactically restricted: the body of the constructor must be a sequence of assignment to the fields of this class, followed by returning the special `this` (the first parameter) reference. The special variable `this` can only appear on the left-hand-side of each assignment, thus the object will not be used until its invariant is fully established. We also restrict through typing rules that each non-null field has to be assigned inside constructor body.

The most interesting clause in the type system is NotNull; the typing rule gives the added assumption that x is *not null* in the `then` part. In this case, the syntactic construct offers a *narrowing* operation on a type.

In our Twelf realization for the non-null system, we use the `map` signature defined in previous work [4] for all the environments. Each class identifier is represented by an unique natural number, as are methods and fields. In addition, inside M, we identify a constructor as a procedure with the same identifier as the class. We also need to ensure each map is consistent with the others (for instance, each class identifier used in some type should also exist as an entry in C). These restrictions are enforced by a series of consistency relations.

Having the class structures defined, we need to convert them to fractional permissions. The most important part of this process is to construct *predicate* for each class in C. A predicate for object o has the form:

$$\begin{aligned} p(o) \stackrel{\text{def}}{=} (&\exists r_1 \cdot (o.f_1 \to r_1 + \Pi_1) + \\ &\ldots + \exists r_n \cdot (o.f_n \to r_n + \Pi_n)) \prec 0.\mathrm{Owned} \end{aligned}$$

where Π_i for reference ρ_i has the form:

$$\Pi_i = \begin{cases} \neg(r_i = 0) + p_i(r_i) & r_i \text{ is not null} \\ \neg(r_i = 0)\ ?\ p_i(r_i) : \emptyset & r_i \text{ is possibly null} \end{cases}$$

In Twelf, each class predicate has a type which takes exactly one parameter (the object itself):

$$\textsc{Var}\ \frac{E(x) = c}{C; M; E \vdash x : c} \qquad \textsc{Null}\ \frac{}{C; M; E \vdash 0 : \texttt{Null}}$$

$$\textsc{Read}\ \frac{C; M; E \vdash e : c^- \qquad C(c)(f) = c_f}{C; M; E \vdash e\,.\,f : c_f} \qquad \textsc{Write}\ \frac{C; M; E \vdash e_1 : c_1^- \qquad C; M; E \vdash e_2 : c_2 \qquad C(c_1)(f) = c_2}{C; M; E \vdash e_1\,.\,f\texttt{=}e_2 : c_2}$$

$$\textsc{Let}\ \frac{C; M; E \vdash e_1 : c_1 \qquad C; M; E, x : c_1 \vdash e_2 : c}{C; M; E \vdash \texttt{let}\ x\texttt{=}e_1\ \texttt{in}\ e_2 : c} \qquad \textsc{Cond}\ \frac{C; M; E \vdash b \qquad C; M; E \vdash e_1 : c \qquad C; M; E \vdash e_2 : c}{C; M; E \vdash \texttt{if}\ b\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : c}$$

$$\textsc{Loop}\ \frac{C; M; E \vdash b \qquad C; M; E \vdash e : c}{C; M; E \vdash \texttt{while}\ b\ \texttt{do}\ e : \texttt{Null}} \qquad \textsc{NotNull}\ \frac{E(x) = c^+ \qquad C; M; E, x : c^- \vdash e_1 : c' \qquad C; M; E \vdash e_2 : c'}{C; M; E \vdash \texttt{if not}\ x\texttt{==}0\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : c'}$$

$$\textsc{Sub}\ \frac{C; M; E \vdash e : c \qquad c <: c'}{C; M; E \vdash e : c'} \qquad \textsc{Call}\ \frac{M(m) = (c_1, \ldots, c_n) \to c \qquad C; M; E \vdash e_i : c_i}{C; M; E \vdash m\texttt{(}e_1\texttt{,...,}e_n\texttt{)} : c}$$

Figure 7: Non-null Typing Rules (Partial)

```
predicate (predtype/+ objectk predtype/0)
```

Also, since the definition is recursive, we use the constructor `predicate/Y` for all classes.

While realizing this in Twelf, we first tried to construct predicate for each class independently. However, as we found out later, this approach didn't work, since each class predicate is not equivalent to its unfolding in permission logic (using `predicate/Y`). In our second approach, we construct the class predicates all at once, and store them inside a special *predicate map*, which is defined as follow:

$$P ::= \epsilon \mid P, c : p(x) \text{ (c is the class identifier)}$$

where p is the predicate for class c. Specifically, when constructing the predicate for a particular class, we first generate a variable to use for the class predicate and then construct the predicates needs for the fields. WHen we need the class predicate, we use the variable stored in the map. The algorithm is similar to a depth-first-traversal on class structure. After all the classes of the enclosing fields are seen, it replaces the predicate variable with the recursive predicate. Then the algorithm moves on to the next unseen class in C.

Once the predicate map is constructed, the conversion for type environment E is straightforward. The conversion of procedure map M to program type ω is a simple iteration; for each procedure type $(c_1, \ldots, c_n) \to c$, we convert it to a procedure type α, while α is:

$$\forall x_1, \ldots, x_n\ (\Pi_1 + \ldots + \Pi_n + 0.\text{Owned} \to 0) \to \exists x_t\ (\Pi_t + 0.\text{Owned} \to 0)$$

where Π_i is the converted permission for type c_i.

The soundness of the non-null type system depends on the following result: for each expression e in the kernel language, if e is well-typed and has type t in non-null type system, with consistent environments C, M, and E, then after converting these environments to P, ω, and Π, respectively, e is also well-typed under permission type system, with ω and input permission Π. In addition, the output permission is $\Pi + \Pi'$, while Π' is the permission for type t, converted from t by using P. This is proved by case analysis on all the possibly forms of expression e. For a conditional b, a similar property is proved via mutual induction.

Take the Write rule as an example, to show $e_1.f\texttt{=}e_2$ is well-typed under permission type, we first need to use induction to get the assumption that e_1 and e_2 are well-typed. For e_1, since it is not null, the output permission Π_2 is

$$\neg(\rho_1 = 0) + p_1(\rho_1) + \Pi_1$$

while p_1 is the predicate for class c_1. For e_2, the above permission is the input of permission typing rule, therefore we need to use the frame rule first. Depending on e_2's type, the permission varies. Here, assume e_2 is not null, then the output permission is:

$$\neg(\rho_1 = 0) + p_1(\rho_1) + \neg(\rho_2 = 0) + p_2(\rho_2) + \Pi_1$$

while p_2 is the predicate for class c_2.

To convert this output to something like $\rho_1.f \to \rho' + \Pi'$, we need to have the field permission *carved out* from $0.\text{Owned} \to 0$ in Π_1. Then, the permission will be transformed to:

$$\exists\rho' \cdot (\rho_1.f \to \rho' + \Pi_f) + \exists\rho' \cdot (\rho_1.f \to \rho' + \Pi_f) \multimap 0.\text{Owned} + \neg(\rho_1 = 0) + p_1(\rho_1) + \neg(\rho_2 = 0) + p_2(\rho_2) + \Pi_1'$$

while Π_f is the permission for field f, and Π_1' is Π_1 without $0.\text{Owned} \to 0$. We can then get this form by unpacking the existential.

The final step is to transform the output permission of consequence to the right form. Since we have $\rho_1.f \to \rho_2$ and $p_2(\rho_2)$, we can pack them again to existential form, and use linear modus ponens to get the permission $0.\text{Owned} \to 0$ back. Also, extra formulae, like $\neg(\rho_1 = 0)$ and $p_1(\rho_1)$, are discarded. For the case that e_2 is possibly null, the proof is similar.

Given the above result, the well-typedness of methods is straightforward. However, constructors need extra attention; for each constructor, the input field permissions are raw, that is are initialized to null. We need to pack them according on their non-null type. Also, since non-null fields are guaranteed to be assigned in constructor body, we need to filter them out and construct the permission accordingly. This task is done by classifying fields into disjoint sets, with separate relations describing properties about them. As "output"

for the theorem, all fields are packed and the class invariant is established.

With the above works done, the soundness theorem is as follow:

soundness For every program g in the kernel language, if g is well-typed under consistent environments C and M, then with the converted program type ω, g can also be type checked under the system of fractional permission.

The proof for this theorem consists basically of iterating over all procedures in the program, and use the above proof of well-typedness for either methods or constructors. This theorem shows not only that non-null types can be expressed using fractional permissions, but also that an entire type system can be reduced to fractional permissions.

6. RELATED WORK

In this work, we prove soundness of a permission system for single-threaded programs. The original fractional permissions paper [2] handled fractions and concurrency, but did not have support for linked structures or nesting. Our later proof [6] did not handle fractions or concurrency. Neither of these earlier proofs were machine-checked. We are currently working to extend our work to cover concurrency using the (unproved) type system outlined by Zhao [18].

Of course, innumerable type systems have been proved sound, so in this section we concentrate on those which incorporate partially linear systems, such as separation logic. Brookes [8] has proved that extending concurrent separation logic to (fractional) permissions is sound in a natural language proof. The underlying system supports a static set of variables and concurrently accessible "resources" but not dynamic allocation. Dynamic allocation [9] has been proved sound in the absence of (fractional) permissions, but it seems reasonable that the combination should be sound, but dynamic allocation of resources and permitting more powerful "imprecise" invariants are not handled.

Parkinson [14] proved the soundness of separation logic applied to a single-threaded Java subset. Vafeiadis [17] recently produced machine-checked proofs (including one handling fractions) that concurrent separation logic is sound for a generic imperative language and a static set of resources including imprecise invariants.

A rather different system proved sound is Bierhoff and Aldrich's modular type state system [1]. This system unusually uses fractions to track shared write permissions. It doesn't address concurrency and the proof is not machine-checked. It does handle dynamically allocated objects. Consistency is maintained by ensuring that only *one* object is "open" at a time and in a syntactically limited context.

Hurlin, Haack and Huisman [12, 11] have a proof of the correctness of a concurrent fractional permission system for a Java-like language. Apparently it handles even imprecise resource invariants, but we haven't been able to check the proof fully yet and the proof itself is not machine-checked.

7. CONCLUSIONS

As expected, fractional permissions with nesting is sound as a type system for Java-like programs, at least in a single-threaded context. Producing a machine-checked proof in Twelf forced us to address strange cases we might not otherwise have noticed in a natural language proof. We also benefited from machine-checked examples to make sure a type system isn't sound merely because it rejects all programs with a particular construct (such as loops).

Fractional permissions can be used as the basis on which to prove the soundness of other type systems, which avoids the need to handle evaluation and redefine consistency for the new systems.

8. REFERENCES

[1] K. Bierhoff and J. Aldrich. Modular typestate verification of aliased objects. Technical Report CMUISRI-07-105, School of Computer Science, Carnegie Mellon University, 2007.

[2] J. Boyland. Checking interference with fractional permissions. In R. Cousot, editor, *Static Analysis: 10th International Symposium*, volume 2694 of *Lecture Notes in Computer Science*, pages 55–72, Berlin, Heidelberg, New York, 2003. Springer.

[3] J. Boyland. An operational semantics including volatile for safe concurrency. *Journal of Object Technology*, 8(4):33–53, June 2009.

[4] J. Boyland. Generating bijections between HOAS and the natural numbers. In *LFMTP: Logical Frameworks and Meta-languages: Theory and Practice*, July 2010.

[5] J. Boyland. Semantics of fractional permissions with nesting. *ACM Transactions on Programming Languages and Systems*, 32(6):Article 22, Aug. 2010.

[6] J. Boyland and W. Retert. Connecting effects and uniqueness with adoption. In *Conference Record of POPL 2005: the 32nd ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 283–295, New York, 2005. ACM Press.

[7] J. Boyland, W. Retert, and Y. Zhao. Comprehending annotations on object-oriented programs using fractional permissions. In M. Parkinson, editor, *International Workshop on Aliasing, Confinement and Ownership in object-oriented programming (IWACO)*, New York, 2009. ACM Press. To appear.

[8] S. Brookes. Variables as resource for shared-memory programs: Semantics and soundness. In *Twenty-second Conference on the Mathematical Foundations of Programming Semantics*, pages 123–150, North-Holland, 2006. Elsevier.

[9] S. Brookes. A semantics for concurrent separation logic. *Theoretical Computer Science*, 375(1–3):227–270, 2007.

[10] M. Fähndrich and K. R. M. Leino. Declaring and checking non-null types in an object-oriented language. In *OOPSLA'03 Conference Proceedings—Object-Oriented Programming Systems, Languages and Applications*, pages 302–312, New York, Nov. 2003. ACM Press.

[11] C. Haack, M. Huisman, and C. Hurlin. Permission-based separation logic for multithreaded Java programs. Submitted for publication, July 2010.

[12] C. Hurlin. *Specification and Verification of Multithreaded Object-Oriented Programs with Separation Logic*. PhD thesis, Université Nice—Sophia Antipolis, Sept. 2009.

[13] P. W. O'Hearn, H. Yang, and J. C. Reynolds. Separation and information hiding. In *Conference Record of POPL 2004: the 31st ACM*

SIGACT-SIGPLAN Symposium on Principles of Programming Languages, pages 268–280, New York, 2004. ACM Press.

[14] M. J. Parkinson. *Local reasoning for Java*. PhD thesis, University of Cambridge, Nov. 2005.

[15] F. Pfenning and C. Schürmann. System description: Twelf—a meta-logical framework for deductive systems. In H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, volume 1632 of *Lecture Notes in Artificial Intelligence*, pages 202–206, Berlin, Heidelberg, New York, July 1999. Springer.

[16] J. Reynolds. Separation logic: a logic for shared mutable data structures. In *Logic in Computer Science*, pages 55–74, Los Alamitos, California, July 22–25 2002. IEEE Computer Society.

[17] V. Vafeiadis. Concurrent separation logic and operational semantics. In *Twenty-seventh Conference on the Mathematical Foundations of Programming Semantics*, 2011.

[18] Y. Zhao. *Concurrency Analysis Based on Fractional Permissions*. PhD thesis, University of Wisconsin–Milwaukee, Department of EE & CS, 2007.

APPENDIX

A. THE PROOFS

The proofs are available online from the webpage `http://www.cs.uwm.edu/~boyland/papers/proving.html`. They consist of 49 Twelf files in the following categories:

Topic	Files	KBytes
Library Signatures	6	828
Kernel Language	13	696
Fractional Permissions	10	1450
Permission Type System	6	300
Non-Null Type System	15	773
Total	49	4047

The library signatures mostly consists of signatures for natural numbers, rational numbers and sets. The kernel language files include all the definitions of the kernel language and generic lemmas, but not the specific proofs about proper synchronization. The permission type system (and soundness proofs) takes up only just under 300K of which 100K are generated automatically from a library map "functor." The non-null system seems large but over 500K of that figure is automatically generated, and it includes the types that it uses instead of using fractional permissions as does the permission type system.