Semantics of Fractional Permissions with Nesting

John Boyland

December 5, 2007

Abstract

Fractional permissions use fractions to distinguish write access (1) from read access (any smaller fraction). Nesting (an extension of adoption) can be used to model object invariants and ownership. Indeed nesting extends both in a form of "monotonically increasing invariants." Fractional permissions thus provides a foundation for defining the meaning of a large variety of access-based annotations: uniqueness, read-only, immutability, method effects, lock protected state etc. In this paper we give a semantics of fractional permissions with nesting (adoption), in terms of "fractional heaps" indicating the mutable state that can be accessed using each permission. We show that fractions are sound and that the system satisfies separation properties.

College of Engineering & Applied Science

Department of Electrical Engineering and Computer Science

Technical Report CS-07-01

Work supported in part by the National Science Foundation (CCF-0702635). The opinions expressed here are not necessarily those of the National Science Foundation or the US Government.

1 Introduction

Fractional permissions [5] are used to show non-interference between two concurrent computations. Each piece of mutable state is associated with exactly one (whole) permission; this whole fraction is needed when modifying the state. Permissions are linear, that is, may only be transferred, not duplicated. However, a permission may be split into fractions, only a fraction is needed for a read thus permitting read-read concurrency without interference. Fractions can be joined back together thus achieving the ability to write the state again once all the readers are finished, and have surrendered their fractions.

With permission nesting, one permission may be nested in the permission to access a field. Then whoever has a permission for the field (the "nester") implicitly also has access to the nested permission. Nesting once performed lasts "forever." The result is what we call a "monotonically increasing invariant."[1] Unlike permissions in general, the knowledge of nesting may be duplicated arbitrarily (in particular, it can be communicated asynchronously). If one has this knowledge and also the nester permission, the nested permission can be "carved" out of it. This operation leads to a form of linear implication.

Fractional permissions with nesting can be used to connect effects, uniqueness and ownership [6] with the ability to distinguish read and write effects, to declare immutable and read-only state, as well as "unique write" state[2]. The combined system can thus give a firm semantic foundation to a host of potentially confusing state management concepts. Our permission system is intended to be used as a basis for a type system verifying such kinds of design intent.

This paper gives a semantics of fractional permissions with nesting in terms of "fractional heaps." Furthermore, we show that this semantics has certain desirable properties:

- We show that two "half" fractions of a permission can be soundly combined into a whole permission.
- We prove that fractional permissions maintain separation, thus allowing modular reasoning. In particular, nesting actions can be carried out independently.

Fractions and nesting interact in interesting ways. In particular, possession of a fraction of a field permission gives one access to the same fraction of any nested permissions. Thus it must be possible to *scale* an arbitrary nested permission by any legal fraction. The other way around, nesting a fraction, gives "read-only ownership," a useful combination. But as shall be demonstrated, it also leads to the possible of (fractional) cyclic nesting which cannot be syntactically prevented as in the case of non-fractional nesting [6].

Fractions and nesting cannot be added directly to separation logic, because the linear implication and existential from separation logic enable unsound reasoning when combined with fractions and nesting. Rather we define restricted

[1] The reader is asked to overlook the oxymoronic nature of an invariant that varies, even monotonically.

[2] state that is otherwise shared but for which writes are restricted to a certain scope

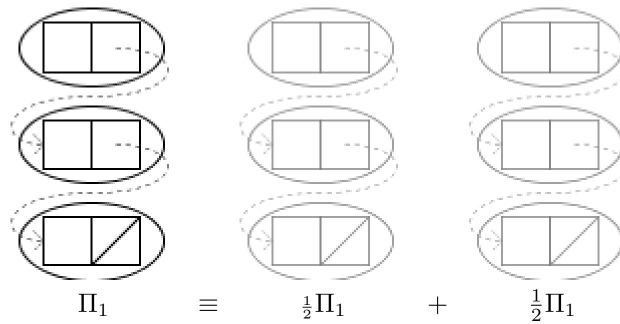
versions of these logical operations that can be safely combined with fractions and nesting.

The paper is structured as follows: Section 2 motivates fractional permissions through a series of related examples involving singly-linked lists; then Section 3 defines the concept of fractional heaps; Section 4 defines the syntax of permissions and shows how the examples are expressed; Section 5 then defines the semantics of fractional permissions, and Section 6 explains and proves fraction combination and separation properties. This paper includes proof sketches; the machine-checked version of all proofs is available (see Appendix).

2 Example

In this section, we informally introduce fractional permissions and fractional heaps with a running example: a linked list and an associated iterator.

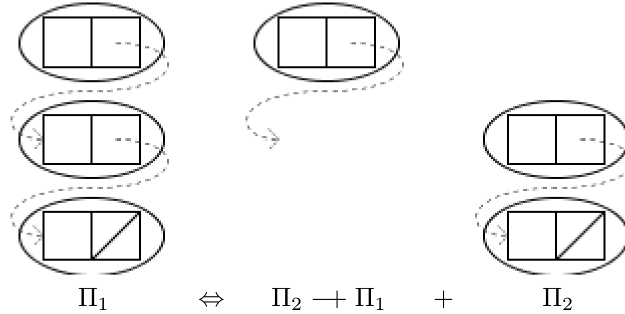
Fractional permissions describe state and are modeled by "fractional heaps." A fractional heap is a heap (partial map of locations to values) that as well as giving the contents of some locations also gives a fraction $q \in (0, 1]$ that this heap possesses of the location. The 1 fraction gives write access, any fraction greater than zero gives read access. A traditional (non-fractional) heap is expressed as a fractional heap with fraction 1 everywhere the heap is defined. Then consider a permission Π_1 that refers to a linked list of three objects. Each object is represented by a permission "All" that nests the two fields (the integer data and the next pointer), drawn as small squares within the "All" oval. We render the heap in black to represent the full permission and in light gray to represent a fractional permission:



$$\Pi_1 \equiv \tfrac{1}{2}\Pi_1 + \tfrac{1}{2}\Pi_1$$

Thus we see that the full (write) permission to access the list can be split into two identical read permissions. The state drawn "gray" will be immutable until the pieces can be put together again. Indeed, one can make arbitrarily small fractions, but it is hard to distinguish these in such a diagram.

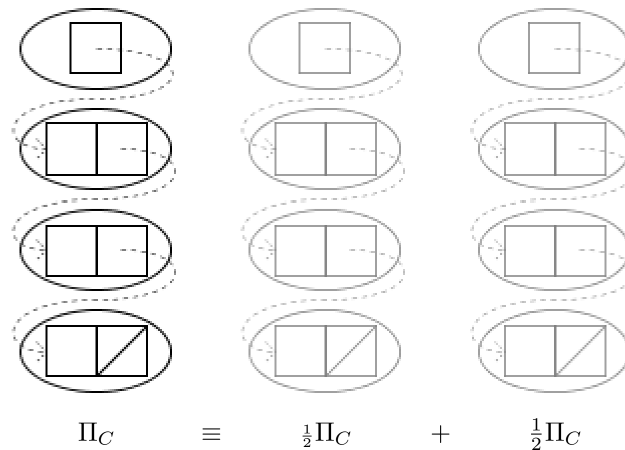
Instead of breaking the list "horizontally" in this way, we can also break it "vertically" using a permission implication:



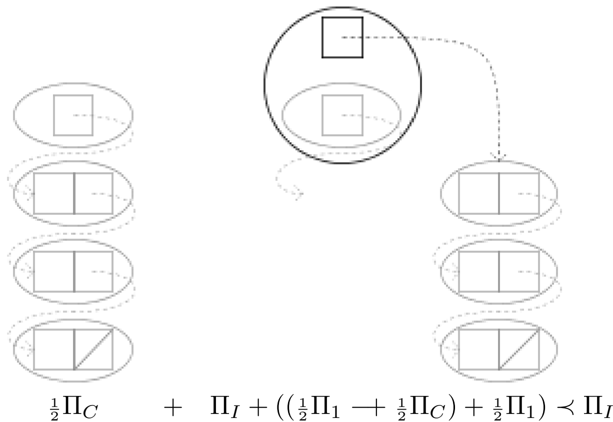
Here the permission $\Pi_2 \mathrel{-\!\!+} \Pi_1$ means "everything permitted by Π_1 except that part permitted by Π_2." Fractions and implications can be combined orthogonally; an example of such a combination appears below.

The $\mathrel{-\!\!+}$ operator (pronounced "scepter") is closely related to separation logic's $-\!\!\star$ operator (and linear logic's $\multimap$ operator). The main distinction between $\mathrel{-\!\!+}$ and $-\!\!\star$ is that $\mathrel{-\!\!+}$ has a restricted semantics that ensures separation in the presence of fractions and nesting.

The invariant in our running example is the invariant of a non-mutating iterator over a linked list container. Following our earlier work [7], an iterator encumbers the permission of the container in order that the container will not be mutated while the iterator is in use. Since we need only read-only access to the container, we first split the container permission into two:



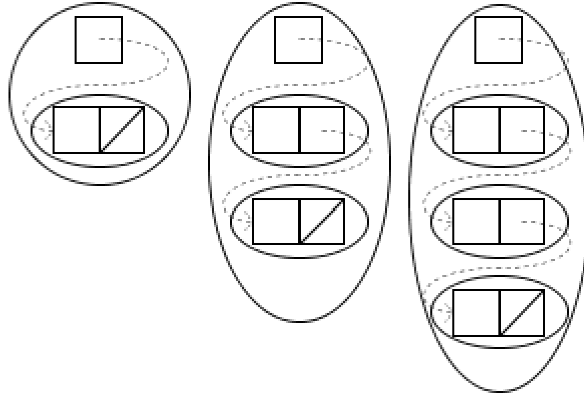
One of the pieces is given to the iterator object which has a pointer to the first element of the list and includes the remaining permission for the container in the form of an implication. The various pieces are now:



On the left, we have the remaining half permission for the container. This permission lets us query the container (including creating another iterator), but does not allow us to mutate it. On the right we have the full permission for the iterator (large oval) which nests both an implication $\frac{1}{2}\Pi_1 \multimap \frac{1}{2}\Pi_C$ and also a field with read permission $\frac{1}{2}\Pi_1$. When we are done with the iterator, the two parts can be combined using linear *modus ponens* to retrieve the half permission to the container. The nesting fact is written as $\Psi \prec \Pi_I$ where Ψ is this complex invariant.[3] The nesting fact doesn't give any permission itself; it only indicates that Π_I includes these other permissions and thus if one has Π_I, one has these other permissions indirectly as well. Even as the iterator moves along, the invariant stays true: the permissions to the individual nodes are transfered from the list part of the invariant to the implication part. Permission nesting thus performs information hiding in the same way as the module invariants of separation logic [16].

Unlike these invariants however, nesting facts can also be added dynamically, thus enabling *monotonically increasing invariants*. The container class described above used "uniqueness" [6]: each node includes the permission to access the next node. An alternate model is "ownership" in which the container "owns" all the nodes and each node only indicates that its successor is also owned by the container. Using ownership, the list could thus have a cycle in it. Ownership is forever; when new nodes are allocated, the container "stretches" to nest another node:

[3]Technically the permission Ψ is nested in the *field*, not the field *permission*.



In no case, however, can a node be transfered out of the container. This is what we mean by a "monotonically increasing invariant."

3 Fractional Heaps

The power of separation logic comes from the fact that separate heap contexts are non-overlapping. Thus reasoning using one subheap is guaranteed not to be invalidated by changes in another subheap. The resulting frame property is very powerful: if some section of code only accesses a subheap, then (almost by definition), any properties on the remainder of the heap are unaffected. The same ability is available using permissions:

$$\frac{\{P\}\,C\,\{Q\}}{\{P \star R\}\,C\,\{Q \star R\}} \qquad\qquad \frac{\Delta;\Pi \vdash s \dashv \Delta';\Pi'}{\Delta;\Pi+\Pi'' \vdash s \dashv \Delta';\Pi'+\Pi''}$$

Separation Logic — Permission Types

Separation logic is given semantics using heaps [13]; fractional permissions with "fractional heaps." To handle fractions, the heap is not simply either defined or undefined for each location, but rather if defined, is defined to a fractional extent between 0 exclusive and 1 inclusive. The heaps of separation logic can then be seen as a special case where every fraction is one.

We now explain why we use an algebraic structure as powerful as the positive rational numbers. Brookes [8] explains why one needs an addition operation with no identity. Now a field permission Ψ_1 may nest a fraction of another field permission $q_2\Psi_2$; if one possesses a fraction $q_1\Psi_1$ of the first field permission, the result is that one indirectly posseses $q_1 q_2$ of the second field permission. Thus nesting implies the need for a multiplication operation, and thus scaling. We permit fractions above 1 in intermediate fractional heaps to make scaling more uniform. Comparing heaps requires the existence of a $\leq$ operation on fractions, and every (useful) fraction must be comparable with 1, so that we know if the

final is legal in a heap or not. This operation must support the usual laws w.r.t. addition and multiplication. The requirement that we support fractions such as $\frac{1}{3}$ comes from the fact that circular fractional nesting is unavoidable as explained later. Putting all these requirements together, one sees that a structure about as rich as the positive rational numbers is called for.

Therefore, a (fractional) heap h is a finite (partial) map of locations to pairs of a positive fraction and a value (for us, an object reference from the countably infinite set O):

$$h : L \rightharpoonup (\mathbf{Q}^+ \times O)$$

Here L is the (countably infinite) set of locations in the heap. We use $\hat{\emptyset}$ to refer to the empty heap that is defined nowhere. A fractional heap where every fraction is 1 is called a *memory* (written μ).

We apply our permission types to object-oriented languages and thus for us locations are fields of objects ($L = O \times F$, where F is a finite set of field names); every object's field has its own individual permission since fields of (mutable) objects can be updated independently. The fraction represents the permission to access the heap at that location (field); it does *not* grant access to the object whose reference is stored in the heap at that location (field). We say that one heap is included in another $h_1 \leq h_2$ if for every fraction in the first, it matches the second with at most that fraction:

$$h_1 \leq h_2 = \forall_{(q_1,o)=h_1(l)}\, q_1 \leq q_2 \text{ where } (q_2, o) = h_2(l)$$

Heaps can be combined by adding together the corresponding fractions, but *only if the values match*: If there is a location l such that $h_i(l) = (q_i, o_i)$ and $o_1 \neq o_2$ then $h_1 \hat{+} h_2$ is undefined. Otherwise it is defined as follows:

$$(h_1 \hat{+} h_2)(l) = \begin{cases} h_1(l) & \text{if } h_2(l) \text{ is undefined} \\ h_2(l) & \text{if } h_1(l) \text{ is undefined} \\ (q_1 + q_2, o) & \text{where } (q_i, o) = h_i(l) \end{cases}$$

It is clearly the case that $h \hat{+} \hat{\emptyset} = \hat{\emptyset} \hat{+} h = h$ for all h.

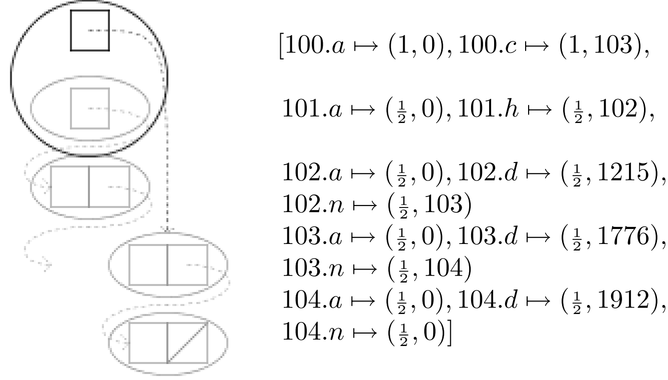
Two fractional heaps that can be added are called *compatible*. Because we permit fractions to exceed 1, a fractional heap is always compatible with itself.

Heaps can be *scaled* by any fraction $q > 0$:

$$(qh)l = (qq', o) \text{ where } hl = (q', o)$$

The ability to scale heaps (and also permissions) is one distinction between this work and the permission semantics of Brookes [8], and is required because of nesting, as explained above.

For example, consider the iterator state after one iteration. Assume the objects are allocated at 100,101, ...; and assume the existence of fields a (all), c (current), h (head), d (data), n (next). Assume further that the data values are 1215, 1776 and 1912 respectively.



Fractional heaps are used to give semantics to permissions (written Π). If we have $h \models_N \Pi$ then the permission grants access as defined by the (fractional) heap h, assuming the current nesting situation is N.

Ambiguity in the semantics comes from having multiple heaps that model the same permission. Unrestricted ambiguity can make the fundamental fraction intuition $\frac{1}{2}\Pi + \frac{1}{2}\Pi \equiv \Pi$ unsound since the two occurrences of Π could refer to different heaps.

Most desirable would be if a permission were "precise." We adopt the concept of "precision" from separation logic: A permission is *precise* if any two compatible fractional heaps h_1 and h_2 that model Π must be equal. Unfortunately however, there are some permissions, such as the first example of an implication $\Pi_2 \multimap \Pi_1$ that are imprecise, as will be shown in Section 5.

O'Hearn and others describe a weaker property "supported" [16] that still permits sound reasoning. In this paper, in contrast, we limit the logical operators to ensure that the ambiguity is harmless. In particular, we prove that if a permission can be modeled by two different but compatible heaps, then it can be modeled by any linear combination of them as well. Fuller explanation must wait until we have defined the modeling relation in Section 5.

4 Syntax

This section describes the syntax of fractional permissions, shows how to express the example permissions and defines a syntactic equivalence operation.

4.1 Basics

Figure 1 gives the syntax for permissions. The basic building block for permission is the field permission $o.f \to o'$ which states that the field f of the object o points to object o'.

The identity of the permission combination operator "+" is $\emptyset$. (We have used the "comma" for this operator in the past.) Here $\Pi + \Pi'$ should be read

$\Pi, \Psi ::=$	*permission:*	$\Gamma ::=$	*formula:*
$o.f \rightarrow o$	*field*	$\top$	*true*
$\emptyset$	*empty*	$\neg\Gamma$	*negation*
$\Pi + \Pi$	*combined*	$\Gamma \wedge \Gamma$	*conjunction*
$q\Pi$	*scaled*	$o = o$	*comparison*
Γ	*formula*	$\Psi \prec o.f$	*nesting*
$\exists r \cdot o.f \rightarrow r + \Pi$	*exist'l*	$p(\bar{o})$	*predicate call*
$\Gamma \mathbin{?} \Pi : \Pi$	*conditional*	$\exists x \cdot \Gamma$	*existential*
$\Psi \multimap \Pi$	*implication*	$\ldots$	$\ldots$

Figure 1: Syntax of permissions and formulae.

as "both Π and Π'."[4] We have already seen scaling and implication in the examples. Existentials are syntactically restricted to be precise. A conditional permission uses a boolean formula to choose between permissions.

A permission can include a (non-linear) formula Γ. Formulae grant no permission, but may include useful information. Thus non-linear logic is embedded in the (linear) permission logic in layered fashion rather than orthogonally as in BI-logic [13]. Figure 1 also gives a (partial) syntax for formulae. Because of the layered nature of the embedding, the semantics of fractional permission depends only in a few well-defined ways on the semantics of formulae. Thus, it is easy to add new formulae to represent such (immutable) things as object type tags.

Aside from the nesting predicate, all the other forms of formulae are standard. The nesting relation $\Psi \prec o.f$ is true if the permission given is nested in the field $o.f$. Being a formula, it doesn't grant any permission, but it does make the nested permission available should one have permission to access the field. The nesting predicate serves as (partial) knowledge of the invariant associated with the object. An object can have separate invariants by using different fields to nest the permissions in. This corresponds to ownership domains [1]. In our examples, we only use one invariant which is nested in the "All" (model) field. In our case, the field "All" is always null (0) and would not need storage in (say) a Java implementation.

4.2 Examples

Named predicates are used to express recursive invariants. For example for a list node stored at an arbitrary location r, the "data" and "next" fields are both nested in the "All" field. The former has a value that is not relevant to permissions (since integers do not denote storage), but the "next" field points to a "unique" node. In other words, it includes permission to access all of the state of the node, and it knows that this node has the same invariant:

[4] Combination (+) corresponds rather confusingly to the "multiplicative conjunctions" of linear logic ($\otimes$) and separation logic ($\star$).

$$
\begin{aligned}
\text{Node}(r) = {} & (\exists i \cdot r.\text{data} \to i) \prec r.\text{All} \wedge \\
& (\exists n \cdot \; r.\text{next} \to n + \\
& \qquad n = 0 \;?\; \emptyset \\
& \qquad\qquad :\; n.\text{All} \to 0 + \\
& \qquad\qquad\;\; \text{Node}\,(n)) \prec r.\text{All}
\end{aligned}
$$

Here the nested conditional permission explicitly tests for nullness. This test is required for soundness because otherwise someone could get (multiple!) permission to access fields of the null pointer. In this example, we used a conjunction of two separate nesting facts Another possibility is to nest a combined permission.

Continuing to the container class: it has one field "head" which points to a node (or is null)

$$
\begin{aligned}
\text{List}(r) = {} & (\exists h \cdot \; r.\text{head} \to h + \\
& \qquad h = 0 \;?\; \emptyset \\
& \qquad\qquad :\; h.\text{All} \to 0+ \\
& \qquad\qquad\;\; \text{Node}\,(h)) \prec r.\text{All}
\end{aligned}
$$

The iterator class has a "cur" field including a permission implication (here we use z to refer to an arbitrary positive fraction):

$$
\begin{aligned}
\text{It}(r,l,z) = {} & (\exists c \cdot \; r.\text{cur} \to c + \\
& \qquad z\,(c = 0 \,?\, \emptyset : c.\text{All} \to 0 + \text{Node}(c)) + \\
& \qquad z\,(c = 0 \,?\, \emptyset : c.\text{All} \to 0 + \text{Node}(c)) \multimap zl.\text{All} \to 0) \\
& \prec r.\text{All}
\end{aligned}
$$

The two parts of this nested permission cannot be (usefully) nested separately because a permission can only use the value of a field (here c) if it includes at least a fraction of the permission for the field, and splitting the permission for "cur" into two pieces, while sound, renders it immutable.

The other structure for linked lists used owned nodes. Here l is the list header, the owner.

$$
\begin{aligned}
\text{ONode}(r,l) = {} & (r.\text{All} \to 0) \prec l.\text{All} \wedge \\
& (\exists i \cdot r.\text{data} \to i) \prec r.\text{All} \wedge \\
& (\exists n \cdot \; r.\text{next} \to n + \\
& \qquad n = 0 \,?\, \emptyset : \text{ONode}\,(n, l)) \prec r.\text{All}
\end{aligned}
$$

The differences between this and the one using uniqueness are that the permission to access r.All is nested in the owner's All permission and the next node doesn't come with permission to access it. The owning container has analogous changes:

$$
\begin{aligned}
\text{OList}(r) = {} & (\exists h \cdot r.\text{head} \to h + \\
& \qquad h = 0 \,?\, \emptyset : \text{ONode}\,(h, r)) \prec r.\text{All}
\end{aligned}
$$

The iterator can be simpler because we don't need to transfer the nodes (no implication):

$$\frac{}{\Pi + \emptyset \Rrightarrow \Pi}\ \text{Q-Identity} \qquad \frac{}{\Pi + \Pi' \Rrightarrow \Pi' + \Pi}\ \text{Q-Commute} \qquad \frac{}{\Pi + (\Pi' + \Pi'') \Rrightarrow (\Pi + \Pi') + \Pi''}\ \text{Q-Associate}$$

$$\frac{\Pi_1 \Rrightarrow \Pi_1' \qquad \Pi_2 \Rrightarrow \Pi_2'}{\Pi_1 + \Pi_2 \Rrightarrow \Pi_1' + \Pi_2'}\ \text{Q-Combine} \qquad \frac{}{q\emptyset \Rrightarrow \emptyset}\ \text{Q-Zero} \qquad \frac{}{1\Pi \Rrightarrow \Pi}\ \text{Q-One} \qquad \frac{}{q(\Pi + \Pi') \Rrightarrow q\Pi + q\Pi'}\ \text{Q-Distribute}$$

$$\frac{q, q' > 0 \qquad qq' = q''}{q(q'\Pi) \Rrightarrow q''\Pi}\ \text{Q-Multiply} \qquad \frac{q, q' > 0 \qquad q + q' = q''}{q\Pi + q'\Pi \Rrightarrow q''\Pi}\ \text{Q-Add}$$

Figure 2: Permission Equivalence (helper relation).

$$\begin{aligned} \mathrm{OIt}(r,l,z) = (\exists c \cdot\ & r.\mathrm{cur} \to c + \\ & c = 0\ ?\ \emptyset : \mathrm{ONode}(c, l) + \\ & zl.\mathrm{All} \to 0) \\ \prec\ & r.\mathrm{All} \end{aligned}$$

In either case however, the iterator permission $i.\mathrm{All} \to 0$ includes all the permission needed to use the iterator. This shows how nesting can be used to perform information hiding.

4.3 Equivalence

Figure 2 defines a relation $\Rrightarrow$ on permissions used to define equivalence:

Definition 4.1 The $\equiv$ relation is the transitive, symmetric and reflexive closure of the $\Rrightarrow$ relation. We write $\Pi \geq \Pi'$ if and only if there exists Π'' where $\Pi \equiv \Pi' + \Pi''$.

The equivalence relation induces a partition on permissions. We choose a particular representative as "canonical."

Definition 4.2 Given an arbitrary total ordering $<$ on permissions, a permission Π is in *canonical* form, if it is in the form

$$(q_1\pi_1 + (q_2\pi_2 + \ldots (q_n\pi_n + \emptyset) \ldots))$$

for $n \geq 0$, where for every $0 < i < j \leq n$, we have $\pi_i < \pi_j$ using this arbitrary order, and each π (called a "unit" permission) has the form

$$\pi \quad ::= \quad \Gamma \mid o.f \to o \mid \Gamma\ ?\ \Pi : \Pi \mid \exists r \cdot \Pi \mid \Pi \multimap \Pi$$

Lemma 4.3 *The following rule is "admissible"*

$$\frac{\Pi \Rrightarrow \Pi'}{q\Pi \Rrightarrow q\Pi'}\ \text{Q-Scale}$$

In other words, adding it would not change the definition of equivalence.

PROOF By induction over the proof of $\Pi \Rrightarrow \Pi'$. Intuitively, we can use Q-DISTRIBUTE, Q-MULTIPLY or Q-ZERO to merge the outer fraction with the shape of permission Π. □

Theorem 4.4 *For every Π, there exists exactly one canonical Ψ such that $\Pi \equiv \Psi$.*

PROOF We prove the result by giving an algorithm for creating the canonical permission using two helper functions:

$$
\begin{aligned}
&\text{Scale}(q,\ (\emptyset)) = (\emptyset)\\
&\quad (q,\ (q'\pi' + \Pi)) = (qq'\pi' + \text{Scale}(q,\ \Pi))
\end{aligned}
$$

$$
\begin{aligned}
&\text{Combine}((),\ \Pi) = \Pi\\
&\quad (\Pi,\ ()) = \Pi\\
\pi < \pi' &\quad ((q\pi + \Pi),\ (q'\pi' + \Pi')) = (q\pi + \text{Combine}(\Pi,(q'\pi' + \Pi')))\\
\pi > \pi' &\quad ((q\pi + \Pi),\ (q'\pi' + \Pi')) = (q'\pi' + \text{Combine}((q\pi + \Pi),\Pi'))\\
\pi = \pi' &\quad ((q\pi + \Pi),\ (q'\pi' + \Pi')) = ((q + q')\pi + \text{Combine}(\Pi,\Pi'))
\end{aligned}
$$

$$
\begin{aligned}
&\text{Canon}(\pi) = (1\pi + \emptyset)\\
&\quad (\emptyset) = (\emptyset)\\
&\quad (q\Pi) = \text{Scale}(q,\text{Canon}(\Pi))\\
&\quad (\Pi + \Pi') = \text{Combine}(\text{Canon}(\Pi),\text{Canon}(\Pi'))
\end{aligned}
$$

The “Scale” helper function scales a canonical permission; the “Combine” helper function merges two canonical permissions.

The full proof (see appendix for information on the machine-checked proofs of all statements in this paper) proceeds by showing that the “Canon” function always terminates, always produces a canonical permission, always produces an equivalent permission and produces the same result for any two equivalent permissions. □

5 Semantics

This section defines the semantics of fractional permissions in terms of fractional heaps. The semantics depends on the current nesting situation N, a complete map on locations to the permissions nested in that location:

$$N : L \rightarrow \{\Pi\}$$

We require that this map yields the empty permission $\emptyset$ except for a finite number of locations.

$$\textsc{B-True}\quad A; N \vdash \top \Downarrow \text{true}$$

$$\textsc{B-Neg}\quad \frac{A; N \vdash \Gamma \Downarrow b}{A; N \vdash \neg\Gamma \Downarrow \neg b}$$

$$\textsc{B-AndFalse1}\quad \frac{A; N \vdash \Gamma_1 \Downarrow \text{false}}{A; N \vdash \Gamma_1 \wedge \Gamma_2 \Downarrow \text{false}}$$

$$\textsc{B-AndFalse2}\quad \frac{A; N \vdash \Gamma_2 \Downarrow \text{false}}{A; N \vdash \Gamma_1 \wedge \Gamma_2 \Downarrow \text{false}}$$

$$\textsc{B-AndTrue}\quad \frac{A; N \vdash \Gamma_1 \Downarrow \text{true} \qquad A; N \vdash \Gamma_2 \Downarrow \text{true}}{A; N \vdash \Gamma_1 \wedge \Gamma_2 \Downarrow \text{true}}$$

$$\textsc{B-Equal}\quad A; N \vdash o{=}o' \Downarrow (o = o')$$

$$\textsc{B-Nest}\quad \frac{N(l) \geq \Psi}{A; N \vdash \Psi \prec l \Downarrow \text{true}}$$

$$\textsc{B-Exist}\quad \frac{A; N \vdash [\delta \mapsto X]\Gamma \Downarrow \text{true}}{A; N \vdash \exists \delta . \Gamma \Downarrow \text{true}}$$

$$\textsc{B-Axiom}\quad \frac{\Gamma \in A}{A; N \vdash \Gamma \Downarrow \text{true}}$$

$$\textsc{B-Pred}\quad \frac{A \cup \{p(\overline{o})\}\,; N \vdash [\overline{r \rightarrow o}]P(p) \Downarrow \text{true}}{A; N \vdash p(\overline{o}) \Downarrow \text{true}}$$

Figure 3: Evaluation rules for boolean formulae: $A; N \vdash \Gamma \Downarrow b$

Permissions are used to analyze stateful programs. At the beginning, we start with the empty nesting situation: $N_0(l) = \emptyset$. While the program is running, new permissions may be nested in locations; thus N may grow (but never shrink). We define the $\leq$ operation on nesting situations $N_1 \leq N_2$:

$$\forall l \cdot N_1(l) \leq N_2(l)$$

where $\leq$ on permissions is defined using equivalence (see Defn. 4.1).

5.1 Formula Evaluation

For the semantics of permissions, we need an evaluation of formula to boolean values written $A; N \vdash \Gamma \Downarrow b$ where $b \in \{\text{true}, \text{false}\}$. The A set is a set of assumptions used for simulated coinduction. It starts off empty. Formula evaluation need not be defined for all formulae, but it must be **deterministic**: a formula cannot evalate to both true and false in the same context, and it must be **stable**:

$$N_1 \leq N_2 \wedge (A; N_1 \vdash \Gamma \Downarrow b) \Rightarrow (A; N_2 \vdash \Gamma \Downarrow b)$$

In other words, a formula cannot change its value during execution. This permits non-linear reasoning even in the face of state changes: once true, always true; once false, always false.

Figure 3 defines evaluation rules for the formulae forms defined in Figure 1. Of some interest is the fact that conjunction can avoid evaluating a sub-formula, which may be necessary because a formula may be undefined in the current nesting situation. For instance, B-Nest shows that a nesting fact can only be true, never false. Since nestings increase monotonically, we can never depend on a nesting *not* being true.

$$\frac{h; \Psi + \Psi' \models^C_N \Pi}{h; \Psi \models^C_N \Psi' \multimap \Pi}\ \text{S-IMPLICATION}$$

$$\hat{\emptyset}; \Psi \models^C_N \Psi \ \text{S-OBLIGATION}$$

$$\frac{\emptyset; N \vdash \Gamma \Downarrow \text{true}}{\hat{\emptyset}; \emptyset \models^C_N \Gamma}\ \text{S-FORMULA}$$

$$\frac{\emptyset; N \vdash \Gamma \Downarrow b \qquad h; \Psi \models^C_N \Pi_b}{h; \Psi \models^C_N \Gamma\ ?\ \Pi_{\text{true}} : \Pi_{\text{false}}}\ \text{S-COND}$$

$$\frac{h; \Psi \models^C_N [\delta \mapsto X]\Pi}{h; \Psi \models^C_N \exists \delta.\Pi}\ \text{S-EXIST}$$

$$\frac{h; \Psi \models^C_N \Pi}{qh; q\Psi \models^C_N q\Pi}\ \text{S-FRACTION}$$

$$\frac{h_1; \Psi_1 \models^C_N \Pi_1 \qquad h_2; \Psi_2 \models^C_N \Pi_2}{h_1 \hat{+} h_2; \Psi_1 + \Psi_2 \models^C_N \Pi_1 + \Pi_2}\ \text{S-COMBINE}$$

$$\frac{\Psi \equiv \Psi' \qquad \Pi \equiv \Pi' \qquad h; \Psi' \models^C_N \Pi'}{h; \Psi \models^C_N \Pi}\ \text{S-EQUIV}$$

$$\frac{h; \Psi \models^{C \cup \{(h;\Psi) \prec l\}}_N N(l) \qquad u = [l \mapsto (1, o)]}{h \hat{+} u; \Psi \models^C_N l \to o}\ \text{S-FIELD}$$

$$\frac{(h; \Psi) \prec l \in C \qquad u = [l \mapsto (1, o)]}{h \hat{+} u; \Psi \models^C_N l \to o}\ \text{S-FIELD-CO}$$

Figure 4: Semantics of Fractional Permissions with Nesting

A similar situation applies to existential formulae and recursive predicates. An existential or predicate call can only be evaluated as true. We assume that $P(p)$ gives the definition of predicate p and that we never call a predicate with the wrong number of arguments. Reflecting standard practice, we use an overline to represent multiple formals $\overline{r}$ and actuals $\overline{o}$. The form $[\overline{r \to o}]P(p)$ means to substitute the actuals for the formals in the body of the predicate. Rules B-PRED and B-AXIOM together implement coinduction: while evaluating the body of the predicate, we assume the result will be true. If we permitted both true and false assumptions, a recursive formula could evaluate to both true and false, violating determinism.

Theorem 5.1 *Boolean evaluation is deterministic.*

PROOF Straightforward structural induction. □

Theorem 5.2 *Boolean evaluation is stable.*

PROOF Straightforward structural induction. □

5.2 Permissions

Figure 4 shows the relation that define when a heap models a permission $h; \Psi \models^C_N \Pi$. We have already discussed the N qualification. We now discuss the

presence of Ψ on the left-side of the relation. The C qualification is explained below.

The Ψ on the left of the modeling relation is an "obligation." The obligation starts empty. Recall that $\Pi_1 \multimap \Pi_2$ means "everything permitted by Π_2 except that permitted by Π_1." This intuition is expressed in making Π_1 an obligation that must be discharged *symbolically* while expanding Π_2. This can be seen in the rules S-IMPLICATION and S-OBLIGATION in Fig. 4. Thus in S-OBLIGATION, if we are looking for heap to model a permission that exactly matches the obligation, the obligation is dicharged, and thus the empty heap fits. New obligations are added in a sub-goal in S-IMPLICATION.

This semantics for implication differs markedly from $-\!\!\star$, for which the permission-equivalent rule would be as follows:

$$\text{S-MAGIC}\quad \frac{\forall_h\ h \models^{\emptyset}_{N} \Psi \wedge (h \mathbin{\hat{+}} h') \text{ exists } \Rightarrow (h \mathbin{\hat{+}} h') \models^{C}_{N} \Pi}{h' \models^{C}_{N} \Psi -\!\!\star \Pi}$$

This definition is highly imprecise since h' could be any heap that conflicts with all heaps h modeling Ψ as well as something more reasonable. We restrict the semantics of $\multimap$ to reduce ambiguity, but it still retains the linear *modus ponens* property.

Theorem 5.3 *If $h_1; \emptyset \models^{\emptyset}_{N} \Pi_1$ and $h_2; \emptyset \models^{\emptyset}_{N} \Pi_1 \multimap \Pi_2$ and h_1 is compatible with h_2, then $h_1 \mathbin{\hat{+}} h_2; \emptyset \models^{\emptyset}_{N} \Pi_2$.*

PROOF (Sketch) We break the provided permission into its unit permissions and then rework the proof tree, substituting the provided unit permission when it is discharged in an obligation. We also combine two fractions of the modeling relation to ensure that we either substitute all of the unit permission or none. This makes the changes at coinduction anticipatable. □

Rule S-FORMULA uses formula evaluation to check that the formula is true, in which case the formula is modeled by the empty heap; this is a consequence of the fact that formulae grant no permissions. This situation contrasts with separation logic where **true** is modeled by any heap.

Rule S-COND uses formula evaluation to choose which permission to expand. The obligation is passed on. Rule S-EXIST ignores the syntactic restriction on existentials for simplicity.

Rules S-FRACTION and S-COMBINE handle scaling and combination respectively, each requires the obligation to be in the form matching the permission being expanded. Rule S-EQUIV may be used to achieve the needed form.

The last rules S-FIELD and S-FIELD-CO deal with field permissions. In the first rule, the nesting situation N is used to determine the current nesting for the location. The resulting heap is added to a unit permission for the field. The C set is used to get a co-inductive effect of the first rule by permitting it to help establish itself. The elements of the C set are triples written $(h; \Psi) \prec l$.

The rule S-FIELD-CO uses the C set to avoid recursively looking at the nested permissions.

This manual implementation of co-induction is needed because a field may directly or indirectly nest itself. For an example, suppose that a field permission $o.f \to 0$ nests one quarter of itself: $N(o.f) = \frac{1}{4}o.f \to 0$. Then consider the following inference tree:

$$\dfrac{\dfrac{\dfrac{\dfrac{\left(([o.f \mapsto (\frac{1}{3},0)],\emptyset) \prec o.f\right) \in R}{[o.f \mapsto (\frac{4}{3},0)];\emptyset \models_N^C o.f \to 0}\text{S-FIELD-CO}}{[o.f \mapsto (\frac{1}{3},0)];\emptyset \models_N^C \frac{1}{4}o.f \to 0}\text{S-FRACTION}}{[o.f \mapsto (\frac{4}{3},0)];\emptyset \models_N^\emptyset o.f \to 0}\text{S-FIELD}}{[o.f \mapsto (\frac{2}{3},0)];\emptyset \models_N^\emptyset \frac{1}{2}o.f \to 0}\text{S-FRACTION}$$

Here $C = \left\{([o.f \mapsto (\frac{1}{3},0)],\emptyset) \prec o.f\right\}$ which is determined "out of thin air" (co-inductively).

The result may look suspect: where did that denominator of "3" come from? The answer is that the result can be seen as fulfilling the recurrence relation:

$$h = \frac{1}{4}h \mathbin{\hat{+}} [o.f \mapsto (1,0)]$$

As it happens, the inference tree just shown gives the only semantics for the permission; the solution is unique. This example shows why we do not restrict ourselves to fractions with power-of-two denominators.

For a second example, suppose we had a linked list structure in which, unlike in the unique example, each node had *half* of the permission of the next node:

$$\begin{aligned}\text{HNode}(r) = {} & (\exists i \cdot r.\text{data} \to i) \prec r.\text{All} \land \\ & (\exists n \cdot \; r.\text{next} \to n + \\ & \quad\quad n = 0 \;?\; \emptyset \\ & \quad\quad\quad\quad : \tfrac{1}{2}n.\text{All} \to 0 + \\ & \quad\quad\quad\quad\;\; \text{HNode}\,(n)) \prec r.\text{All}\end{aligned}$$

Consider the permission $\frac{1}{2}o.\text{All} \to 0$ This permission could be modeled by the heap

$$[o.\text{data} \mapsto (1,42), o.\text{next} \mapsto (1,o), o.\text{All} \mapsto (1,0)]$$

in which the node points to itself (a cyclic list). In other words, a cyclic list can express the concepts of an infinite list with a convergent series of fractions.

Both of these examples *could* occur in a running program. Thus if we were to omit S-FIELD-CO, a program could create an unrepresentable heap. Preventing this occurrence with a type system would be very restrictive. In particular, detecting a cycle requires global knowledge. Rather, we wish to support a type system where nesting decisions may be made independently. Thus we are

$$
\dfrac{
  \dfrac{\dfrac{\dots}{\emptyset; N \vdash o = 0 \Downarrow \text{false}}=
  \quad
  \dfrac{
    \dfrac{
      \dfrac{
        \dfrac{\dfrac{\hat{\emptyset};\emptyset \models_N \emptyset}{h_1;\emptyset \models_N o.\text{data} \to 666}\to}{h_1;\emptyset \models_N \exists i \cdot o.\text{data} \to i}\exists
        \quad
        \dfrac{\dfrac{\dfrac{\hat{\emptyset};\emptyset \models_N \emptyset}{h_2;\emptyset \models_N o.\text{next} \to o}\to \quad \dfrac{}{\hat{\emptyset};\Psi(o) \models_N \Psi(o)}!}{h_2;\Psi(o) \models_N o.\text{next} \to o + \Psi(o)}+}{h_2;\Psi(o) \models_N \exists r \cdot o.\text{next} \to r + \Psi(r)}\exists
      }{h_1 \hat{+} h_2;\Psi(o) \models_N \exists i \cdot o.\text{data} \to i + \exists r \cdot o.\text{next} \to r, \Psi(r)}+
    }{h_3;\Psi(o) \models_N o.\text{All} \to 0}\to
    \quad
    \dfrac{\vdots}{\hat{\emptyset};\emptyset \models_N \text{Node}(o)}\Gamma
  }{h_3;\Psi(o) \models_N o.\text{All} \to 0 + \text{Node}(o)}+
  }{h_3;\Psi(o) \models_N \Psi(o)}?
}{h_3;\emptyset \models_N \Psi(o) \mathrel{-\!\!+} \Psi(o)}\mathrel{-\!\!+}
$$

Figure 5: Example of a non-empty heap modeling a trivial implication.

compelled to support the possibility of circular nesting. Incidentally, we don't use a completely co-inductive system because co-induction would require that S-Equiv (obviously) and S-Obligation (more problematically) would need to be folded into the syntax-directed inference rules.

5.3 Imprecision

The rules in Figure 4 do not ensure a "precise" semantics. For example, for an arbitrary non-null object reference o, consider the permission $\Psi(o) \mathrel{-\!\!+} \Psi(o)$ where $\Psi(r)$ is a shorthand defined as follows:

$$r = 0 \mathbin{?} \emptyset : r.\text{All} \to 0, \text{Node}(r)$$

Let N be the nesting situation such that o is a legal node (both the "o.data" and "o.next" fields are nested in "o.All"). Now we have the trivial modeling:

$$\dfrac{\dfrac{}{\hat{\emptyset};\Psi(o) \models_N \Psi(o)}\text{S-Obligation}}{\hat{\emptyset};\emptyset \models_N \Psi(o) \mathrel{-\!\!+} \Psi(o)}\text{S-Implication}$$

Figure 5 shows a much more complex modeling. Let $h_1 = [o.\text{data} \mapsto (1, 666)]$ be a heap that defines the data field and $h_2 = [o.\text{next} \mapsto (1, o)]$ be a heap that defines the next field to point back to the node. Let $h_3 = h_1 \hat{+} h_2 \hat{+} [o.\text{All} \mapsto (1, 0)]$ be a heap that defines both fields and their nesting.

The diagram in Fig. 5 omits the C qualification which is unused; it also elides formula evaluation with dots; finally it uses abbrevations for the names of rules: S-Obligation is !, S-Field is $\to$, S-Combine is +, S-Exists is $\exists$, S-Conditional is ?, S-Implication is $\mathrel{-\!\!+}$).

Thus, we see that the permission $\Psi(o) \mathrel{-\!\!+} \Psi(o)$ has at least two different compatible heaps: $\hat{\emptyset}$ and h_3. Analyzing this result, one sees it does not depend on circular nesting or even (non-whole) fractions. Indeed nesting plays no major role either; it is used only to express a recursive data structure. Separation logic

uses recursive (linear) predicates for a similar purpose and the same result would apply, even if it used $-\!\!+$ rather than $-\!\!\star$. Intuitively, in a memory where the node currently points to itself, the implication permits the loop to be traversed once, or not at all.

The imprecision we see here, however, is not fatal. In the following section, we prove the semantics still preserves important properties.

6 Properties

In this section, we explain and prove the properties of our semantics of fractional permissions with nesting. We also show where less restricted operations (such as the implication and existential operators in separation logic) would make the properties not true.

6.1 Compatability

The fundamental fraction property is $\frac{1}{2}\Pi + \frac{1}{2}\Pi \equiv \Pi$. While S-EQUIV ensures that this property is carried over to heap modeling, one needs a stronger property. In particular, one wishes that a syntactic existential can be treated as such:

Theorem 6.1 *If $h; \emptyset \models^{\emptyset}_{N} \exists r \cdot \Pi$, then there exists o such that $h; \emptyset \models^{\emptyset}_{N} [r \mapsto o]\Pi$.*

If we did *not* restrict the form of the existential, we would have the following counter-example:

$$
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{\hat{\emptyset}; \emptyset \models_{\emptyset} \emptyset}{u_1; \emptyset \models^{\emptyset}_{\emptyset} 1.f \to 0}\,\to
}{u_1; \emptyset \models^{\emptyset}_{\emptyset} \exists r \cdot r.f \to 0}\,\exists
}{\frac{1}{2}u_1; \frac{1}{2}\emptyset \models^{\emptyset}_{\emptyset} \frac{1}{2}\exists r \cdot r.f \to 0}\,\tfrac{1}{2}
\qquad
\dfrac{
\dfrac{
\dfrac{\hat{\emptyset}; \emptyset \models_{\emptyset} \emptyset}{u_2; \emptyset \models^{\emptyset}_{\emptyset} 2.f \to 0}\,\to
}{u_2; \emptyset \models^{\emptyset}_{\emptyset} \exists r \cdot r.f \to 0}\,\exists
}{\frac{1}{2}u_2; \frac{1}{2}\emptyset \models^{\emptyset}_{\emptyset} \frac{1}{2}\exists r \cdot r.f \to 0}\,\tfrac{1}{2}
}{\frac{1}{2}u_1 \hat{+} \frac{1}{2}u_2; \frac{1}{2}\emptyset + \frac{1}{2}\emptyset \models^{\emptyset}_{\emptyset} \frac{1}{2}\exists r \cdot r.f \to 0 + \frac{1}{2}\exists r \cdot r.f \to 0}\,+
}{\frac{1}{2}u_1 \hat{+} \frac{1}{2}u_2; \emptyset \models^{\emptyset}_{\emptyset} \exists r \cdot r.f \to 0}\,\equiv
$$

Here $u_i = [i.f \mapsto (1,0)]$ and there is no way to bind r to get a single modeling.

Theorem 6.1 is proved using the following more general lemma about the inversion of the modeling of "unit" permissions (see page 10):

Lemma 6.2 (Unit inversion) *If $h; \emptyset \models^{\emptyset}_{\emptyset} \pi$, then there exists a proof of this relation where the bottom (root) rule application is the appropriate for the unit permission: one of* S-IMPLICATION*,* S-FORMULA*,* S-COND*,* S-EXIST*,* S-FIELD*.*

PROOF Instances of S-FRACTION, S-COMBINE, S-EQUIV at the root are pushed up into the unit rules. In the case of S-EXIST, we make use of the syntactic restriction on existentials to ensure that two separate models bind the variable to the same location, thus ensuring compatability. □

6.2 Separation

The semantics we have defined satisfy the all-important "separation property":

Theorem 6.3 (Separation) *Given a memory μ, permission Π, nesting situation N and $h \leq \mu$ such that $h; \Psi \models^{\emptyset}_{\emptyset} \Pi$ then for any memory μ' where $h \leq \mu'$, we still have $h; \Psi \models^{\emptyset}_{\emptyset} \Pi$.*

This weak form of separation is trivially true since changing μ to μ' has no effect on the relation $h; \Psi \models^{\emptyset}_{\emptyset} \Pi$. Somewhat more interesting is the ability to discard unneeded permissions:

Theorem 6.4 (Discard Separation) *Given a memory μ, permissions $\Pi_1 + \Pi_2$, N and $h \leq \mu$ such that $h; \emptyset \models^{\emptyset}_{\emptyset} \Pi_1 + \Pi_2$, then there exists a $h' \leq h$ such that $h'; \emptyset \models^{\emptyset}_{\emptyset} \Pi_1$.*

Proof Simple induction. □

The main separation result for this paper is "nesting separation" which means that nesting can be carried out locally without restriction. In other words, if one has the permission to some state, it can be nested in any location whatsoever and the new nesting situation can be used to define semantics for the remainder of the permissions with a heap that is no larger than the sum of what it was before plus the heap for the permission that was nested. This would *not* be the case if the semantics didn't handle cyclic nesting (S-Field-Co).

Theorem 6.5 (Nesting Separation) *Given a memory μ, a nesting situation N, and permissions Ψ and Π where $h; \emptyset \models^{\emptyset}_{\emptyset} \Psi$, $h'; \emptyset \models^{\emptyset}_{\emptyset} \Pi$ and $h \hat{+} h' \leq \mu$ then, for an arbitrary l in which we nest Ψ, there exists $h'' \leq h \hat{+} h' \leq \mu$ such that $h''; \emptyset \models^{\emptyset}_{N'} \Pi$ where $N' = N[l \mapsto N(l) + \Psi]$.*

Proof (Sketch)
We prove a more general lemma that computes

$$h'' \leq h' \hat{+} \frac{r'}{1-r} h$$

where $(r, o) = h(l), (r', o') = h(l')$. (Here, in an abuse of notation, we use r to signify that the fraction could be zero.) If $r = 1$, then $r' = 0$, and we instead prove that $h'' = h'$.

The property is demonstrated by rewriting the proof tree for Π and whenever we find a field permission for the nester, we graft in the proof tree for the nested permission. This process is a little more complex because the newly grafted tree may cause a cycle, and must be examined for instances of the nester field permission. The potential infinite recursion is stopped by remembering which field permissions we are currently expanding and substituting with an instance of S-Field-Co for those cases. □

As a consequence, nesting can be carried out locally, although it has global consequences,

Nesting separation is *not* true in the presence of the less restricted "magic wand" operator of separation logic. If we had S-MAGIC, then one can form a counter-example using $\Pi = (\top \prec 0.f \mathrel{-\!\!\star} \neg\top)$. Then using S-MAGIC, we have $\hat{\emptyset}; \emptyset \models^{\emptyset}_{\emptyset} \Pi$ because with an empty nesting, there can be no heap that models $\top \prec 0.f$. Indeed Π can be modeled by any heap. But if we then nest $\top$ in $o.f$, then Π cannot be represented by *any* heap. The problem is that the permission on the left-hand-side of S-MAGIC is treated in a "negative" manner which gets around the the attempt by B-NEST to ensure that nesting cannot be negated.

7 Related Work

Our work is inspired by the "separation logic" of O'Hearn, Reynolds and others [18, 16, 13]. In particular we wish to add to the work of Bornat, Parkinson, Brookes and others [4, 17, 8] who have also put fractions and separation logic together. The main advances of this work beyond Brookes' semantics [8] are nesting and fractional scaling of permissions. Brooke's generalization of fractions captures fraction addition but not fraction multiplication.

Separation logic includes "resource invariants" [16] which function as a form of statically fixed nesting: when one accesses the named resource using a particular syntactic form, one gets permission to access the invariant term in the dynamic extent of the enclosed block. Nesting extends this concept by enabling any field to be a "resource" (unbounded number of resources) and by allowing the invariant to be dynamically increased. Furthermore, since implications are used to track the state of a resouce whose invariant is being used, there is no need to use synchronization to access all resources.

Aside from fractions, nesting and notational differences ($+$ and $-\!\!+$ in place of $\star$ and $-\!\!\star$), our permission logic differs from separation logic in several other ways:

- We syntactically restrict the use of existentials to ensure that the witness can be determined from memory state. A similar restriction could be used in separation logic for "precise" terms.
- The non-linear portion of the logic (Γ) can only describe state-free (immutable) properties. Thus it has "second-class" status within the logic rather than being fully intermixed as in the logic of bunched implications.
- The linear implication has a restricted semantics that enables nesting separation; permitting monotonically increasing invariants.

Bierhoff and Aldrich [2] use linear logic and fractions to check typestate in Java-like programs. Unconventionally, they use a fraction value of 0 to represent a read permission, and so use other techniques to keep count of the number of read permissions that have been issued. Type states can be seen as permissions

nested in model fields. Instead of full scaling (fractions are not visible at the linear logic level), they use a rule that ensures that read access to a typestate is transitive applied to nested permissions. They do not use general nesting between objects (ownership), but do include a concept of a "shared" read/write pointer. They include a semantics (consistency) in a companion technical report [3]. Since only one object can be "open" at a time there is no need to define fractional heaps.

Nesting is an extension of Vault's adoption operation [11]. Adoption is a relation between whole objects. Nesting extends adoption to permit any field to nest state, and to permit the state to be an arbitrary permission, rather than only whole objects. Furthermore with adoption, one could access only one adoptee at once, but with nesting, any number of nested permissions can be accessed simultaneously. Adoption was defined only with a type system, but here we provide a semantics.

Permission nesting unifies and generalizes object "ownership" [9] and data groups [14, 12]. Nesting can only model relations that are immutable once effected during the lifetime of the participants. Thus it is insufficient by itself to model ownership transfer [10, 15]. Rather one must use a unique proxy object.

8 Conclusions

Fractional permissions with nesting are assigned a semantics based on fractional heaps. We define a new kind of linear implication. Fractional permissions satisfy the requirements of separation, and thus can be used to analyze concurrent programs.

Acknowledgments

I thank Bill Retert for numerous discussions and comments. Yang Zhao helped write natural language proofs in an earlier formalization of the system. The FLUID and PLAID groups at CMU read several drafts and provided helpful comments. I thank James Noble for incisive comments.

SDG

References

[1] Jonathan Aldrich and Craig Chambers. Ownership domains: Separating aliasing policy from mechanism. In Martin Odersky, editor, *ECOOP'04 — Object-Oriented Programming, 18th European Conference*, volume 3086 of *Lecture Notes in Computer Science*, pages 1–25, Berlin, Heidelberg, New York, 2004. Springer.

[2] Kevin Bierhoff and Jonathan Aldrich. Modular typestate checking of aliased objects. In *OOPSLA '07*, New York, 2007. ACM Press. To appear.

[3] Kevin Bierhoff and Jonathan Aldrich. Modular typestate verification of aliased objects. Technical Report CMU-ISRI-07-105, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA, March 2007.

[4] Richard Bornat, Cristiano Calcagno, Peter O'Hearn, and Matthew Parkinson. Permission accounting in separation logic. In *POPL '05*, 2005.

[5] John Boyland. Checking interference with fractional permissions. In *SAS '03*, 2003.

[6] John Boyland and William Retert. Connecting effects and uniqueness with adoption. In *POPL '05*, 2005.

[7] John Boyland, William Retert, and Yang Zhao. Iterators can be independent "from" their collections. In *IWACO '07: International Workshop on Aliasing Confinement and Ownership*, July 2007.

[8] Stephen Brookes. Variables as resource for shared-memory programs: Semantics and soundness. In *MFPS XXII*, 2006.

[9] David Clarke. *Object Ownership and Containment*. PhD thesis, University of New South Wales, Sydney, Australia, 2001.

[10] David Clarke and Tobias Wrigstad. External uniqueness. In Benjamin C. Pierce, editor, *Informal Proceedings of International Workshop on Foundations of Object-Oriented Languages 2003 (FOOL 10)*. January 2003.

[11] Manuel Fähndrich and Robert DeLine. Adoption and focus: Practial linear types for imperative programming. In *PLDI '02*, 2002.

[12] Aaron Greenhouse and John Boyland. An object-oriented effects system. In Rachid Guerraoui, editor, *ECOOP'99 — Object-Oriented Programming, 13th European Conference*, volume 1628 of *Lecture Notes in Computer Science*, pages 205–229, Berlin, Heidelberg, New York, 1999. Springer.

[13] Samin S. Ishtiaq and Peter W. O'Hearn. BI as an assertion language for mutable data structures. In *POPL '01*, 2001.

[14] K. Rustan M. Leino, Arnd Poetzsch-Heffter, and Yunhong Zhou. Using data groups to specify and check side effects. In *Proceedings of the ACM SIGPLAN '02 Conference on Programming Language Design and Implementation*, volume 37, pages 246–257, New York, May 2002. ACM Press.

[15] Peter Müller and Arsenii Rudich. Formalization of ownership transfer in universe types. Technical Report 556, ETH Zurich, 2007.

[16] P. W. O'Hearn, H. Yang, and J. C. Reynolds. Separation and information hiding. In *POPL '04*, 2004.

[17] Matthew Parkinson, Richard Bornat, and Cristiano Calcagno. Variables as resource in Hoare logics. In *LICS '06*, 2006.

[18] John Reynolds. Separation logic: a logic for shared mutable data structures. In *Logic in Computer Science*, 2002.

A Supplementary Material

We have mechanized all the theorems and lemmas in over 50K lines of Twelf including over two thousand theorems, two of which are not proved. The latter two deal with an (unimplemented) bijection between permissions and the natural numbers. The existence of such a bijection is obvious, but the proof has proved too difficult for now to formalize.

The current release of the Twelf proofs (which require Twelf 1.5R3) is available at `http://www.cs.uwm.edu/~boyland/papers/frac-nesting.tgz` including proofs of the following lemmas and theorems, giving the name of the corresponding Twelf metatheorem(s):

Lemma 4.3 `equiv/frac-admissible`

Theorem 4.4 `canon-deterministic`, `canon-total`, `canon-eq-implies-equiv`, `equiv-preserves-canon`

Theorem 5.1 `booleval-deterministic`

Theorem 5.2 `booleval-stable`

Theorem 5.3 `scepter-complements-combine*`

Theorem 6.1 `precise-exists-can-be-opened`

Lemma 6.2 `models-unit-inversion`

Theorem 6.4 `discard-separation`

Theorem 6.5 `nesting-separation`