Superficially Substructural Types

Neelakantan R. Krishnaswami
MPI-SWS
neelk@mpi-sws.org

Aaron Turon
Northeastern University
turon@ccs.neu.edu

Derek Dreyer
MPI-SWS
dreyer@mpi-sws.org

Deepak Garg
MPI-SWS
dg@mpi-sws.org

Abstract

Many substructural type systems have been proposed for controlling access to shared state in higher-order languages. Central to these systems is the notion of a *resource*, which may be split into disjoint pieces that different parts of a program can manipulate independently without worrying about interfering with one another. Some systems support a *logical* notion of resource (such as permissions), under which two resources may be considered disjoint even if they govern the *same* piece of state. However, in nearly all existing systems, the notions of resource and disjointness are fixed at the outset, baked into the model of the language, and fairly coarse-grained in the kinds of sharing they enable.

In this paper, inspired by recent work on "fictional disjointness" in separation logic, we propose a simple and flexible way of enabling any module in a program to create its own custom type of splittable resource (represented as a commutative monoid), thus providing fine-grained control over how the module's private state is shared with its clients. This functionality can be incorporated into an otherwise standard substructural type system by means of a new typing rule we call *the sharing rule*, whose soundness we prove semantically via a novel resource-oriented Kripke logical relation.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory; D.3.3 [*Programming Languages*]: Language Constructs and Features—Abstract data types; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs; F.3.3 [*Logics and Meanings of Programs*]: Studies of Program Constructs

General Terms Languages, Design, Theory, Verification

Keywords Substructural type systems, separation logic, sharing rule, commutative monoids, fictional disjointness, ADTs, hidden state, dependent types, capabilities, Kripke logical relations

1. Introduction

Over the past decade, many *substructural* type systems—based primarily on variants of *linear logic* [20] and *separation logic* [34]—have been proposed as a means of verifying critical semantic properties of higher-order stateful programs, ranging from basic memory safety to full functional correctness. These type systems and their key substructural elements go by a variety of names—*e.g.,* typestate [38, 11], uniqueness [8], regions [39], capabilities [42], Hoare types [27]—but one thing they all have in common is that they give programmers the ability to reason locally about the effects of their code on the state of shared resources.

The essence of this local reasoning is captured by the "frame" property: if an operation f consumes a resource satisfying the type or assertion A and produces one satisfying B, then f can also be seen to transform $A \otimes C$ to $B \otimes C$, where C is an arbitrary "frame" representing assumptions about the greater ambient environment in which f is executed. The $\otimes$ here denotes multiplicative (or "separating") conjunction, which ensures that the resource satisfying $A \otimes C$ can be split into disjoint pieces satisfying A and C, respectively; since f only consumes the resource satisfying A, it is guaranteed to leave the resource satisfying C untouched.

As this discussion suggests, a central element in substructural type systems is the notion of a *resource*, as well as the ability to split a resource into *disjoint* pieces. A resource, in essence, describes (1) the *knowledge* that the consumer of that resource has about the machine state, and (2) what *rights* they have to change the state. Two resources are then considered disjoint if they do not *interfere* with each other, that is: any operation permitted by the rights of one resource should not violate the knowledge of the other.

In some substructural type systems, such as those based on separation logic, resources take the form of entities—such as heaps—that enjoy an immediate *physical* interpretation of disjointness. When an operation consumes a heap h, it has full access to h as a physical object: it knows what h is and has the right to modify it as it pleases. In other systems, resources take the form of "permissions" or "capabilities", which are strictly *logical* descriptions of the knowledge and rights concerning some shared state. In particular, two logical resources may be considered disjoint even if they govern the *same* piece of state. For instance, "fractional" permissions [7] enable the "full" permission to a memory location ($x \mapsto v$)—which gives its consumer the knowledge that x currently points to v and the right to update x's contents—to be split into two "half" permissions ($x \overset{.5}{\mapsto} v \otimes x \overset{.5}{\mapsto} v$)—which provide their respective consumers with the knowledge that x points to v but *not* the right to update it. These half permissions are logically disjoint because they ensure that neither consumer can violate the other's knowledge that x points to v.

However, in nearly all existing systems, the notions of resource and disjointness are fixed at the outset, baked into the model of the language, and fairly coarse-grained in the kinds of sharing they enable. This is unfortunate: ideally, we would like to have a way of defining more fine-grained *custom* logical notions of resource and disjointness on a per-module or per-library basis.

1.1 Motivating Example: A Memory Manager

To take a concrete example, consider a module M implementing an explicit memory manager. M will of course maintain some private data structure representing its free list, and it will expect a certain invariant A of that data structure to hold whenever its methods are invoked. If M is simple enough that this invariant is the *only*

ICFP'12, September 9–15, 2012, Copenhagen, Denmark.

constraint needed on its methods, we can give it an interface like:

$$\begin{array}{rcl} \mathsf{malloc} & : & A \multimap \exists X : \mathsf{Loc}.\ \mathsf{ptr}\ X \otimes \mathsf{cap}\ X\ 1 \otimes A \\ \mathsf{free} & : & \forall X : \mathsf{Loc}.\ \mathsf{ptr}\ X \otimes \mathsf{cap}\ X\ 1 \otimes A \multimap A \end{array}$$

Here, $\mathsf{ptr}\ X$ is a singleton type inhabited only by the pointer X, and $\mathsf{cap}\ X\ 1$ represents the knowledge that X points to a value of unit type 1, along with the full capability to modify it. The invariant A is threaded through the pre- and post-conditions of the operations, but in some type systems it could even be hidden entirely [32].

However, the above interface would *not* work for a more realistic memory manager that required the client to free only memory previously allocated through the manager. For instance, in the Version 7 Unix memory manager—verified (and debugged) recently by Wickerson *et al.* [43]—the implementation internally maintains a chain of pointers to the cells preceding contiguous blocks of memory, both free and allocated. In order to preserve its invariant that the blocks it maintains are contiguous, the manager must only permit its client to free a block that the manager "knows about" (has marked as allocated in its internal chain). The type of the free operation must therefore make the set of allocated blocks explicit, so that it can require the freed location to belong to that set. We can achieve this by parameterizing the manager's invariant A over the set of allocated locations L, and revising its interface as follows:

$$\begin{array}{rcl} \mathsf{malloc} & : & \forall L : \mathsf{LocSet}.\ A(L) \multimap \\ & & \quad \exists X : \mathsf{Loc}.\ \mathsf{ptr}\ X \otimes \mathsf{cap}\ X\ 1 \otimes A(L \uplus \{X\}) \\ \mathsf{free} & : & \forall L : \mathsf{LocSet}.\ \forall X : \mathsf{Loc}. \\ & & \quad \mathsf{ptr}\ X \otimes \mathsf{cap}\ X\ 1 \otimes A(L \uplus \{X\}) \multimap A(L) \end{array}$$

Unfortunately, this latter interface is problematic if the memory manager is used by multiple client modules that one would like to typecheck/verify independently. Each client module only really cares about the locations that *it* allocates/frees, but because the "global" state of the memory manager—*i.e.,* the full set of allocated locations L—is made explicit in the type $A(L)$, each client will in fact be sensitive to interference from other clients. Consequently, each client will need to pollute its own interface with explicit information about how it affects this global state, thereby leaking implementation details in the process.

Ideally, we would like a way of giving each client its own *local* view of the global state. A simple way to provide such a local view would be to allow the memory allocator's invariant to somehow be split up into (and reconstituted from) logically disjoint pieces:

$$\begin{array}{rcl} \mathsf{split} & : & \forall L_1, L_2 : \mathsf{LocSet}.\ A(L_1 \uplus L_2) \multimap A(L_1) \otimes A(L_2) \\ \mathsf{join} & : & \forall L_1, L_2 : \mathsf{LocSet}.\ A(L_1) \otimes A(L_2) \multimap A(L_1 \uplus L_2) \end{array}$$

In particular, given $A(L)$ for some initial L, we could use split to generate any number of copies of $A(\emptyset)$, each of which could be passed to a separate client, thus rendering each client completely oblivious to the existence of the others.

Intuitively, the splitting provided by the split operation is perfectly safe because L_1 and L_2 are disjoint sets, and so the only *right* granted to the owner of $A(L_1)$—namely, the right to free the locations in L_1—cannot possibly violate the *knowledge* of the owner of $A(L_2)$—namely, that the locations in L_2 are allocated. Clearly, though, if we can support such split and join operations, then $A(L)$ no longer means what it did previously: rather than asserting that L *is* the global set of allocated locations, it now asserts merely that L is some *subset* of them that the owner of $A(L)$ has the right to free. In other words, we are treating sets of locations as a kind of splittable resource, and we are using this custom resource to control the knowledge and rights that any one client module has concerning the global, shared state of the memory manager.

The question is: how can we put this intuition on a sound and flexible formal footing, thus enabling any module to develop its own custom notion of splittable resource in a safe, principled way?

1.2 Commutative Monoids to the Rescue!

The goal of this paper is to show that the above example is but one instance of a simple and general pattern, and that there is a simple and general way of supporting such custom resource management within an otherwise standard substructural type system.

The basic idea behind our approach is inspired by some very recent work on separation logic, specifically Jensen and Birkedal's *fictional separation logic* [21], Dinsdale-Young *et al.*'s *views* [12], and Ley-Wild and Nanevski's *subjective concurrent separation logic* [25]. Although these developments are all motivated by different concerns (to be described in Section 5), a common thread running through them is the idea of accounting for various custom notions of splittable resource—along with their attendant notions of *knowledge* and *rights*—in terms of *commutative monoids*.

A commutative monoid is a set S equipped with a commutative, associative *composition* operator $(\cdot) : S \times S \to S$, and a *unit* element $\epsilon \in S$ such that $\forall x \in S.\ x \cdot \epsilon = x$. If one can cast one's notion of custom resource as a commutative monoid, then one can view the global, shared state as the composition $r_L \cdot r_F$ of one's *local* resource r_L with the resource r_F of one's *frame* (*i.e.,* one's *environment*). Owning the local r_L gives one the *knowledge* that the global state must be some "extension" of r_L (*i.e.,* it must equal $r_L \cdot r_F$ for some frame resource r_F). It also gives one the *right* to update the global state however one likes, so long as the new global state satisfies $r'_L \cdot r_F$ for some r'_L. In other words, one may change one's local resource to an arbitrary r'_L, so long as the change is *frame-respecting*, *i.e.,* it leaves the frame resource r_F alone.

The notion of logical resource that we suggested for the memory manager module in Section 1.1 is expressible very naturally as a commutative monoid: sets of locations, with composition defined as disjoint union ($\uplus$) and $\epsilon = \emptyset$. Furthermore, the malloc and free operations are both frame-respecting, due to their universal quantification over the framing location set L.

But many other notions of splittable resource are instances of commutative monoids as well. In their work on fictional separation logic (FSL) [21], Jensen and Birkedal thus propose a way of allowing different modules in a program to specify their interfaces in terms of assertions—such as $A(L)$ in Section 1.1—about different, module-specific notions of resource, encoded as different commutative monoids. This ability to encode module-specific protocols governing shared state was in fact already present to a large extent in earlier work on *deny-guarantee reasoning* [15] and *concurrent abstract predicates* [13]; the main selling point of FSL in comparison is that it adopts a simpler and more abstract monoidal view of resource that is not bound up with concurrency-related concerns. (For a more detailed analysis, see Section 5.)

In this paper, we show how to lift the ideas of FSL and its predecessors from the first-order setting of separation logic to the higher-order setting of a substructural type system.

1.3 Contributions

We make two main contributions: one syntactic, the other semantic.

Our **syntactic** contribution is to propose a new typing rule, which we call *the sharing rule*, that gives the author of a module fine-grained control over how the module's private linear resources (*e.g.,* the full capability to access its internal data structures) are shared with its clients. In particular, as witnessed in our motivating example, the sharing rule allows the capabilities to access this shared state to be split (by $\otimes$) into pieces that are logically disjoint according to a custom commutative monoid of one's choosing. This means that our type system is only *superficially substructural*: under the hood, the A and B in $A \otimes B$ may both be capabilities to read and write *the very same* shared state, albeit in ways that are guaranteed not to interfere with each other.

We present the sharing rule in the context of a fairly standard affine type system (Section 2), supporting a combination of features from Dependent ML [45] and $\mathbf{L}^3$ [5]. While this language is not as expressive as, say, Hoare Type Theory [27]—which we would eventually like to target as well—it is nevertheless rich enough to encode interesting examples (Section 3), while simple enough to focus our attention on the sharing rule itself.

Compared with the support for custom monoids in FSL, our sharing rule is more flexible because it enables types indexed by different commutative monoids to be freely intermingled using the language's general-purpose $\otimes$ type. In contrast, FSL's "indirect Hoare triples" must be indexed explicitly by a particular monoid, and composing specifications that are indexed by different monoids requires additional, somewhat inconvenient, machinery.

However, in the face of arbitrary higher-order programs, our implementation of the sharing rule necessarily carries a dynamic cost, namely the use of a lock to protect updates to the shared state from unsafe re-entrancy. We do not believe this imposes a serious practical restriction on the use of the sharing rule in our sequential setting, but it is clear that a better approach is needed if we wish to scale to the concurrent setting. For special cases of the rule—*e.g.,* where the primitive operations on the shared state do not invoke unknown functions—it is possible to show that locking is not needed, but we leave a thorough examination of such optimizations to future work. We discuss another, more complicated but potentially more scalable approach in Section 5.

Our **semantic** contribution is a novel step-indexed Kripke logical-relations model of shared state, which facilitates a clean semantic proof of the soundness of our sharing rule (Section 4). The structure of our Kripke model directly reflects the intuition behind the sharing rule. In particular, its "possible worlds" W—which encode representation invariants on shared state—take the form of tuples of commutative monoids (think: one monoid for each application of the sharing rule). Associated with each monoid is a resource predicate that says how to interpret an element of the monoid as an invariant on some underlying resources—*e.g.,* in our motivating example, how the full set of allocated locations L maps to an invariant on the memory manager's internal state. Crucially, this resource predicate may describe invariants not only on the physical heap, but also on *logical* resources (expressed as a tuple of elements of all the monoids in W), thus enabling applications of the sharing rule to be soundly layered on top of one another.

We conclude the paper with a detailed comparison to related work (Section 5) and a discussion of future work (Section 6).

2. The Core Language

Our core calculus is an implicitly-typed version of affine F_ω, extended with domains that index types. We call these domains *sorts* and their elements index terms. With a sufficiently rich language of index terms, and propositions and type-level quantification over them, we retain much of the flexibility of dependent types for giving rich type-based specifications for programs, without requiring the index terms to coincide with program terms, thus avoiding the problematic issue of affine variable occurrences in types.

Figure 1 lists the syntactic forms of the language, including sorts, index terms, propositions over index terms, kinds, types, terms, contexts (for the static semantics) and heaps (for the dynamic semantics). Judgments for checking well-formedness of kinds, index terms, propositions and contexts, as well as logical inference, type equality and typing are listed in Figure 2, but we elide the standard rules for inferring these judgments.

Sorts, Index Terms, and Propositions Sorts, ranged over by the metavariable σ, include mathematical domains such as natural numbers, tuples, functions, locations and sets, denoted by the metavariable t. Sorts are interpreted as plain mathematical sets and new sorts can be added if needed. For precise specification of properties of index terms, we allow propositions of first-order logic over the index domains to appear in our types. The standard judgment $\Sigma; \Pi \vdash P$ means that P can be inferred from the assumptions in Π, for all instances of the free variables in Σ.

Sorts	σ	::=	$\mathbb{N} \mid 1 \mid \sigma \times \sigma \mid 2 \mid \mathsf{Loc} \mid \sigma \to \sigma$ $\mid \sigma_\bot \mid \mathsf{seq}\ \sigma \mid \mathcal{P}(\sigma) \mid \mathbb{Q} \mid \ldots$
Index Terms	t	::=	$X \mid n \mid \mathsf{tt} \mid \mathsf{ff} \mid \ldots$
Propositions	P, Q	::=	$\top \mid P \wedge Q \mid P \supset Q \mid \bot \mid P \vee Q$ $\mid \forall X : \sigma.\ P \mid \exists X : \sigma.\ P$ $\mid t = u \mid t > u \mid \ldots$
Kinds	κ	::=	$\circ \mid \sigma \to \kappa$
Types	A	::=	$1 \mid A \otimes B \mid A \multimap B \mid !A$ $\mid \mathsf{ptr}\ t \mid \mathsf{cap}\ t\ A$ $\mid \forall \alpha : \kappa.\ A \mid \exists \alpha : \kappa.\ A$ $\mid \forall X : \sigma :: P.\ A \mid \exists X : \sigma :: P.\ A \mid$ $\mid \mathsf{bool}\ t \mid \mathsf{nat}\ t \mid [A]$ $\mid \mathsf{if}(t, A, B) \mid \alpha \mid \lambda X : \sigma.\ A \mid A\ t$
Terms	e	::=	$x \mid \langle\rangle \mid \langle e, e' \rangle \mid \mathsf{let}\ \langle x, y \rangle = e\ \mathsf{in}\ e'$ $\mid \lambda x.\ e \mid e\ e' \mid\ !v \mid \mathsf{let}\ !x = e\ \mathsf{in}\ e'$ $\mid \mathsf{new}(e) \mid \mathsf{get}_{e'}\ e \mid e :=_{e''} e'$ $\mid \mathsf{tt} \mid \mathsf{ff} \mid \mathsf{if}(e, e_1, e_2)$ $\mid n \mid \mathsf{case}(e, 0 \to e_1, \mathsf{s}\ x \to e_2)$ $\mid \mathsf{fix}\ f(x).\ e \mid \mathsf{share}(e, \overline{v_i}) \mid \bullet$
Eval Contexts	E	::=	$[\,] \mid \langle E, e \rangle \mid \langle v, E \rangle \mid$ $\mid \mathsf{let}\ \langle x, y \rangle = E\ \mathsf{in}\ e \mid E\ e \mid v\ E$ $\mid\ !E \mid \mathsf{let}\ !x = E\ \mathsf{in}\ e$ $\mid \mathsf{new}(E) \mid \mathsf{get}_e\ E \mid \mathsf{get}_E\ v \mid E :=_{e'} e$ $\mid v :=_{e'} E \mid v :=_E v' \mid \mathsf{if}(E, e, e')$ $\mid \mathsf{case}(E, 0 \to e_1, \mathsf{s}\ x \to e_2) \mid \mathsf{share}(E, \overline{v_i})$
Values	v	::=	$\langle\rangle \mid \langle v, v' \rangle \mid \lambda x.\ e \mid\ !v$ $\mid \ell \mid \mathsf{fix}\ f(x).\ e \mid n \mid \mathsf{tt} \mid \mathsf{ff} \mid \bullet \mid x$
Heaps	h	::=	$\cdot \mid h, \ell : v$

Contexts

Index/Type	Σ	::=	$\cdot \mid \Sigma, \alpha : \kappa \mid \Sigma, X : \sigma$
Proposition	Π	::=	$\cdot \mid \Pi, P$
Unrestricted	Γ	::=	$\cdot \mid \Gamma, x : A$
Affine	Δ	::=	$\cdot \mid \Delta, x : A$
Combined	Ω	::=	$\Sigma; \Pi; \Gamma; \Delta$

Figure 1. Syntax

$\Sigma \vdash A : \kappa$	Well-kindedness
$\Sigma \rhd t : \sigma$	Well-sortedness
$\Sigma \rhd P : \mathsf{prop}$	Well-formedness of propositions
$\Sigma \vdash \Pi\ \mathsf{ok}$	Well-formedness of propositional context
$\Sigma \vdash \Gamma\ \mathsf{ok}$	Well-formedness of hypothetical context
$\Sigma; \Pi \vdash P$	Logical entailment
$\Sigma; \Pi \vdash A \equiv B : \kappa$	Type constructor equality
$\Omega \vdash e : A$	Well-typedness

Figure 2. Judgments

Types and Terms We use a standard affine type system, whose rules are shown in Figures 4 and 5. The natural presentation of typing has four contexts; to increase the legibility of the rules, we abbreviate these with a single symbol Ω, and define notations for adding hypotheses and merging contexts in Figure 3.

As expected, the unit term $\langle\rangle : 1$ types in a context with any set of resources Δ; the typing rule for $\langle e_1, e_2 \rangle : A \otimes B$ splits its

$$
\begin{array}{lcll}
\Omega, x : A & = & \Sigma; \Pi; \Gamma; \Delta, x : A & \text{if} \quad \Omega = \Sigma; \Pi; \Gamma; \Delta \\
\Omega, x :_! A & = & \Sigma; \Pi; \Gamma, x : A; \Delta & \text{if} \quad \Omega = \Sigma; \Pi; \Gamma; \Delta \\
\Omega, \alpha : \kappa & = & \Sigma, \alpha : \kappa; \Pi; \Gamma; \Delta & \text{if} \quad \Omega = \Sigma; \Pi; \Gamma; \Delta \\
\Omega, X : \sigma & = & \Sigma, X : \sigma; \Pi; \Gamma; \Delta & \text{if} \quad \Omega = \Sigma; \Pi; \Gamma; \Delta \\
\Omega, P & = & \Sigma; \Pi, P; \Gamma; \Delta & \text{if} \quad \Omega = \Sigma; \Pi; \Gamma; \Delta \\
\\
\Omega_1, \Omega_2 & = & \Sigma; \Pi; \Gamma; \Delta_1, \Delta_2 & \text{if} \quad \Omega_1 = \Sigma; \Pi; \Gamma; \Delta_1 \\
& & & \quad\;\; \Omega_2 = \Sigma; \Pi; \Gamma; \Delta_2
\end{array}
$$

Figure 3. Context Manipulation Operations

$$\boxed{\Omega \vdash e : A}$$

$$\frac{x : A \in \Gamma}{\Sigma; \Pi; \Gamma; \Delta \vdash x : A} \qquad \frac{x : A \in \Delta}{\Sigma; \Pi; \Gamma; \Delta \vdash x : A} \qquad \frac{}{\Omega \vdash \langle\rangle : 1}$$

$$\frac{\Omega \vdash v : A}{\Omega \vdash \bullet : [A]} \qquad \frac{\Omega_1 \vdash e_1 : A \qquad \Omega_2 \vdash e_2 : B}{\Omega_1, \Omega_2 \vdash \langle e_1, e_2 \rangle : A \otimes B}$$

$$\frac{\Omega_1 \vdash e : A \otimes B \qquad \Omega_2, x : A, y : B \vdash e' : C}{\Omega_1, \Omega_2 \vdash \mathsf{let}\ \langle x, y \rangle = e\ \mathsf{in}\ e' : C}$$

$$\frac{\Omega, x : A \vdash e : B}{\Omega \vdash \lambda x.\ e : A \multimap B} \qquad \frac{\Omega_1 \vdash e : A \multimap B \qquad \Omega_2 \vdash e' : A}{\Omega_1, \Omega_2 \vdash e\ e' : B}$$

$$\frac{\Sigma; \Pi; \Gamma; \cdot \vdash v : A}{\Sigma; \Pi; \Gamma; \Delta \vdash\ !v : !A} \qquad \frac{\Omega_1 \vdash e : !A \qquad \Omega_2, x :_! A \vdash e' : C}{\Omega_1, \Omega_2 \vdash \mathsf{let}\ !x = e\ \mathsf{in}\ e' : C}$$

$$\frac{\Omega \vdash e : A}{\Omega \vdash \mathsf{new}(e) : \exists X : \mathsf{Loc} :: \top.\ !\mathsf{ptr}\ X \otimes \mathsf{cap}\ X\ A}$$

$$\frac{\Omega \vdash e : \mathsf{ptr}\ t \qquad \Omega' \vdash e' : \mathsf{cap}\ t\ A}{\Omega, \Omega' \vdash \mathsf{get}_{e'}\ e : A \otimes \mathsf{cap}\ t\ 1}$$

$$\frac{\Omega_1 \vdash e : \mathsf{ptr}\ t \qquad \Omega_2 \vdash e' : A \qquad \Omega_3 \vdash e'' : \mathsf{cap}\ t\ 1}{\Omega_1, \Omega_2, \Omega_3 \vdash e :=_{e''} e' : \mathsf{cap}\ t\ A}$$

$$\frac{\Sigma; \Pi; \Gamma, f : A \multimap B; x : A \vdash e : B}{\Sigma; \Pi; \Gamma; \cdot \vdash \mathsf{fix}\ f(x).\ e : A \multimap B}$$

$$\frac{}{\Omega \vdash \mathsf{tt} : \mathsf{bool}\ \mathsf{tt}} \qquad \frac{}{\Omega \vdash \mathsf{ff} : \mathsf{bool}\ \mathsf{ff}} \qquad \frac{}{\Omega \vdash n : \mathsf{nat}\ n}$$

$$\frac{\Omega \vdash e : \mathsf{bool}\ t \qquad \Omega', t = \mathsf{tt} \vdash e_1 : C \qquad \Omega', t = \mathsf{ff} \vdash e_2 : C}{\Omega, \Omega' \vdash \mathsf{if}(e, e_1, e_2) : C}$$

$$\frac{\begin{array}{c}\Omega \vdash e : \mathsf{nat}\ t \\ \Omega', t = 0 \vdash e_1 : C \qquad \Omega', X : \mathbb{N}, t = \mathsf{s}\ X, x : \mathsf{nat}\ X \vdash e_2 : C\end{array}}{\Omega, \Omega' \vdash \mathsf{case}(e, 0 \to e_1, \mathsf{s}\ x \to e_2) : C}$$

Figure 4. Typing Rules

resource context Δ into two disjoint parts for checking subterms e_1 and e_2; and, to type the affine function $\lambda x.\ e : A \multimap B$, we add the hypothesis $x : A$ to the affine context to check the body e.

The exponential $!A$ is subject to a *value restriction* — we can type terms $!v$ at type $!A$ only when v is a value. The intuition for this restriction is that (following the standard affine interpretation) a term of type $!A$ is duplicable, so the value it evaluates to must not depend on affine resources. If v were not a value, then its evaluation could create new affine resources on which its result depended (e.g., the evaluation might allocate fresh memory and return it).

The base types $\mathsf{bool}\ t$ and $\mathsf{nat}\ u$ are *singleton* types, indexed by the Boolean sort 2 and the natural number sort $\mathbb{N}$, respectively. So,

$$\boxed{\Omega \vdash e : A}$$

$$\frac{\Omega, \alpha : \kappa \vdash v : B}{\Omega \vdash v : \forall \alpha : \kappa.\ B} \qquad \frac{\Omega, X : \sigma, P \vdash v : A}{\Omega \vdash v : \forall X : \sigma :: P.\ A}$$

$$\frac{\begin{array}{c}\Omega \vdash e : \forall X : \sigma :: P.\ A \\ \Omega = \Sigma; \Pi; \Gamma; \Delta \qquad \Sigma \rhd t : \sigma \qquad \Sigma; \Pi \vdash [t/X]P\end{array}}{\Omega \vdash e : [t/X]A}$$

$$\frac{\Omega \vdash e : \forall \alpha : \kappa.\ B \qquad \Omega = \Sigma; \Pi; \Gamma; \Delta \qquad \Sigma \vdash A : \kappa}{\Omega \vdash e : [A/\alpha]B}$$

$$\frac{\begin{array}{c}\Omega \vdash e : [t/X]A \\ \Omega = \Sigma; \Pi; \Gamma; \Delta \qquad \Sigma \rhd t : \sigma \qquad \Sigma; \Pi \vdash [t/X]P\end{array}}{\Omega \vdash e : \exists X : \sigma :: P.\ A}$$

$$\frac{\begin{array}{c}\Omega \vdash v : \exists X : \sigma :: P.\ A \\ \Omega', X : \sigma, P, x : A \vdash e : C \qquad X, x \notin \mathrm{FV}(C)\end{array}}{\Omega, \Omega' \vdash [v/x]e : C}$$

$$\frac{\begin{array}{c}\Omega = \Sigma; \Pi; \Gamma; \Delta \\ \Sigma \vdash A : \kappa \qquad \Sigma, \alpha : \kappa \vdash B : \circ \qquad \Omega \vdash e : [A/\alpha]B\end{array}}{\Omega \vdash e : \exists \alpha : \kappa.\ B}$$

$$\frac{\Omega \vdash v : \exists \alpha : \kappa.\ B \qquad \Omega', \alpha : \kappa, x : B \vdash e : C \qquad \alpha \notin \mathrm{FV}(C)}{\Omega, \Omega' \vdash [v/x]e : C}$$

$$\frac{\Omega = \Sigma; \Pi; \Gamma; \Delta \qquad \Omega \vdash e : A \qquad \Sigma; \Pi \vdash A \equiv B : \circ}{\Omega \vdash e : B}$$

$$\frac{\begin{array}{c}\Omega = \Sigma; \Pi; \Gamma; \Delta \\ \Sigma; \Pi \vdash P \vee Q \qquad \Omega, P \vdash e : A \qquad \Omega, Q \vdash e : A\end{array}}{\Omega \vdash e : A}$$

$$\frac{\begin{array}{c}\Omega = \Sigma; \Pi; \Gamma; \Delta \\ \Sigma; \Pi \vdash \exists X : \sigma.\ P \qquad \Omega, X : \sigma, P \vdash e : A \qquad \Sigma \vdash A : \circ\end{array}}{\Omega \vdash e : A}$$

$$\frac{\Omega = \Sigma; \Pi; \Gamma; \Delta \qquad \Sigma; \Pi \vdash \bot \qquad \Sigma \vdash A : \circ}{\Omega \vdash e : A}$$

Figure 5. Typing Rules, Continued

for example, the only value of type $\mathsf{bool}\ \mathsf{tt}$ is tt and the only value of type $\mathsf{nat}\ 17$ is 17.

For access to shared memory, we introduce the singleton type $\mathsf{ptr}\ \ell$. A term of type $\mathsf{ptr}\ \ell$ evaluates to the location ℓ. Additionally, we have a capability type $\mathsf{cap}\ \ell\ A$, which represents the *permission* to dereference the pointer ℓ and obtain a value of type A. Intuitively, $\ell : \mathsf{ptr}\ \ell$ is a freely duplicable pointer, which can be shared, but the capability to use the pointer, of type $\mathsf{cap}\ \ell\ A$, is affine, and can be shared only in a controlled manner using our sharing rule. Since $\mathsf{cap}\ \ell\ A$ only represents a capability, the actual value of type $\mathsf{cap}\ \ell\ A$ is *computationally irrelevant*, and we write it as $\bullet$.

The two types $\mathsf{ptr}\ t$ and $\mathsf{cap}\ t\ A$ are tied to each other by the typing rules for reading and writing memory. For example, the $\mathsf{get}_{e'}\ e$ operation (see Figure 4) dereferences a pointer e of type $\mathsf{ptr}\ t$, but it requires the capability e' of type $\mathsf{cap}\ t\ A$. It returns a pair of type $(\mathsf{cap}\ t\ 1) \otimes A$. The operational semantics of $\mathsf{get}_\bullet\ \ell$ (Figure 6), removes the current value of ℓ from the store, and replaces it with the value $\langle\rangle$. (It cannot also leave the contents of the pointer in place since doing so would violate any affine constraints on the contents. However, such behaviour is encodable for references containing a $!A$, which is duplicable.) The write

$$
\begin{array}{lcl}
\langle h; \mathsf{let}\ \langle x_1, x_2\rangle = \langle v_1, v_2\rangle\ \mathsf{in}\ e\rangle & \hookrightarrow & \langle h; [v_1/x_1, v_2/x_2]e\rangle \\
\langle h; (\lambda x.\ e)\ v\rangle & \hookrightarrow & \langle h; [v/x]e\rangle \\
\langle h; \mathsf{let}\ !x = !v\ \mathsf{in}\ e\rangle & \hookrightarrow & \langle h; [v/x]e\rangle \\
\langle h; \mathsf{new}(v)\rangle & \hookrightarrow & \langle h \uplus [\ell : v]; \langle !\ell, \bullet\rangle\rangle \\
\langle h \uplus [\ell : v]; \mathsf{get}_\bullet\ \ell\rangle & \hookrightarrow & \langle h \uplus [\ell : \langle\rangle]; \langle v, \bullet\rangle\rangle \\
\langle h \uplus [\ell : \langle\rangle]; \ell :=_\bullet v\rangle & \hookrightarrow & \langle h \uplus [\ell : v]; \bullet\rangle \\
\langle h; (\mathsf{fix}\ f(x).\ e)\ v\rangle & \hookrightarrow & \langle h; [\mathsf{fix}\ f(x).\ e/f, v/x]e\rangle \\
\langle h; \mathsf{if}(\mathsf{tt}, e, e')\rangle & \hookrightarrow & \langle h; e\rangle \\
\langle h; \mathsf{if}(\mathsf{ff}, e, e')\rangle & \hookrightarrow & \langle h; e'\rangle \\
\langle h; \mathsf{case}(0, 0 \to e, \mathsf{s}\ x \to e')\rangle & \hookrightarrow & \langle h; e\rangle \\
\langle h; \mathsf{case}(\mathsf{s}\ v, 0 \to e, \mathsf{s}\ x \to e')\rangle & \hookrightarrow & \langle h; [v/x]e'\rangle \\
\\
\langle h; \mathsf{share}(v, \overline{v_i})\rangle & \hookrightarrow & \langle h \uplus [\ell : \mathsf{ff}]; \\
 & & \quad \langle \bullet, !\mathrm{op}_i, !\mathrm{split}, !\mathrm{join}, !\mathrm{promote}\rangle\rangle
\end{array}
$$

$$
\begin{array}{rcl}
\text{where}\ \ \mathrm{op_i} & = & \lambda x.\ \mathsf{let}\ \langle \mathit{flag}, \_\rangle = \mathsf{get}_\bullet\ \ell\ \mathsf{in} \\
 & & \quad \mathsf{let}\ \_ = \ell :=_\bullet \mathsf{tt}\ \mathsf{in} \\
 & & \quad \mathsf{if}\ \mathit{flag}\ \mathsf{then}\ (\mathsf{fix}\ f(x).\ f\ x)\ \langle\rangle \\
 & & \qquad \mathsf{else}\ \ \mathsf{let}\ y = v_i\ x\ \mathsf{in} \\
 & & \qquad\quad \mathsf{let}\ \_ = \ell :=_\bullet \mathsf{ff}\ \mathsf{in}\ y
\end{array}
$$

$$
\mathrm{split} = \lambda x.\ \langle \bullet, \bullet\rangle \qquad \mathrm{join} = \lambda x.\ \bullet \qquad \mathrm{promote} = \lambda x.\ !\bullet
$$

$$
\frac{\langle h; e\rangle \hookrightarrow \langle h'; e'\rangle}{\langle h; E[e]\rangle \hookrightarrow \langle h'; E[e']\rangle}
$$

Figure 6. Operational Semantics

operation $\ell :=_\bullet v'$ takes a pointer ℓ of type $\mathsf{ptr}\ \ell$, new contents v', and a capability $\bullet$ of type $\mathsf{cap}\ X\ 1$.

We generalize the idea of computational irrelevance by introducing the irrelevant type $[A]$, which is inhabited by the dummy value $\bullet$ if there is *some* value inhabiting A (see the typing rule for $[A]$ in Figure 4). The type $[A]$ is employed gainfully in our sharing rule (Section 3). Our semantic model validates several equivalances on irrelevant types, including $[\mathsf{cap}\ t\ A] \equiv \mathsf{cap}\ t\ A$ and $[A \otimes B] \equiv [B \otimes A]$, which we use freely in our examples.

Propositions over index domains are embedded in the type system at quantified types $\forall X : \sigma :: P.\ A$ and $\exists X : \sigma :: P.\ A$. Intuitively, $e : \forall X : \sigma :: P.\ A$ means that for all terms t of sort σ satisfying the proposition P, e has the type $[t/X]A$. The type $\exists X : \sigma :: P.\ A$ has the dual meaning. We also include an inconsistency rule (the last rule in Figure 5): if the propositional context Π is inconsistent (derives false), then any term is well-typed in Π. (The two prior rules give the rules for existentials and disjunctions.)

In addition, we also include type-level computation with indices with the $\mathsf{if}(t, A, B)$ type, which is equal to A if t is true, and B if t is false. There are no explicit introduction or elimination forms for this type; we simply make use of the equality judgment. To assist in this, the typing for the term-level if-then-else construct adds the appropriate equality hypotheses about its index argument in the branches of the conditional. (Similar rules apply for the other index domains, but we suppress them for space reasons.)

Kinds, κ, in our language have the forms $\circ$ (affine types) and $\sigma \to \kappa$ (dependent types). We include type-level lambda-abstraction $\lambda X : \sigma.\ A$, type-level application $A\ t$ and the universal and existential polymorphic types $\forall \alpha : \kappa.\ A$ and $\exists \alpha : \kappa.\ A$. We could also include type constructor polymorphism, but we omit it for simplicity. We need a value restriction for all quantified types because quantifiers are implicitly introduced and eliminated, and do not delay evaluation (unlike in explicit System F).

Our choice of maximal implicitness naturally makes typechecking undecidable. It should be routine to add enough type and proof annotations to make typechecking decidable, and we chose the implicit style both to make our examples more readable, and to reduce the number of clauses in the term syntax. On the whole, our language is a relatively conservative integration of the ideas of Dependent ML [45, 18] into an affine language with a type structure

$$
\frac{\Sigma \vdash A : \sigma \to \circ \quad \Sigma; \Pi; \Gamma; \Delta \vdash e : [A\ t] \qquad \Sigma; \Pi \vdash \mathrm{monoid}_\sigma(\epsilon, (\cdot)) \quad \forall i.\ \Sigma; \Pi; \Gamma; \cdot \vdash v_i : [A/\alpha]\mathrm{spec}_i}{\Sigma; \Pi; \Gamma; \Delta \vdash \mathsf{share}(e, \overline{v_i}) : \exists \alpha : \sigma \to \circ.\ [\alpha\ t] \otimes \overline{!\mathrm{spec}_i} \otimes !\mathsf{splitT} \otimes !\mathsf{joinT} \otimes !\mathsf{promoteT}}
$$

where

$$
\begin{array}{rcl}
\mathrm{spec}_i & = & \forall X : \sigma.\ \forall Y : \sigma_i' :: P_i.\ B_i \otimes [\alpha\ (t_i \cdot X)] \multimap \\
 & & \qquad \exists Z : \sigma_i'' :: Q_i.\ C_i \otimes [\alpha\ (t_i' \cdot X)] \\
 & & \text{where}\ X, \alpha \notin \mathrm{FV}(P_i, Q_i, B_i, C_i, t_i, t_i') \\
\mathsf{splitT} & = & \forall X, Y : \sigma.\ [\alpha\ (X \cdot Y)] \multimap [\alpha\ X] \otimes [\alpha\ Y] \\
\mathsf{joinT} & = & \forall X, Y : \sigma.\ [\alpha\ X] \otimes [\alpha\ Y] \multimap [\alpha\ (X \cdot Y)] \\
\mathsf{promoteT} & = & \forall X : \sigma :: X = X \cdot X.\ [\alpha\ X] \multimap\ ![\alpha\ X] \\
\mathrm{monoid}_\sigma(\epsilon, (\cdot)) & = & \forall X : \sigma.\ \epsilon \cdot X = X \wedge \\
 & & \forall X, Y : \sigma.\ X \cdot Y = Y \cdot X \wedge \\
 & & \forall X, Y, Z : \sigma.\ (X \cdot Y) \cdot Z = X \cdot (Y \cdot Z)
\end{array}
$$

Figure 7. The Sharing Rule

resembling that of $\mathbf{L}^3$ [5]. The primary novelty in our language is encapsulated in the *sharing rule*, which lets us put user-defined logical resources on a first-class footing. We describe this rule and its applications in the following section.

3. The Sharing Rule

A purely affine type discipline is too restrictive for most programs. In this section, we describe the *sharing rule*, our method for introducing controlled aliasing into an affine language. The intuition behind this rule is that if a library has a particular programmer-defined notion of resource, and if all the operations the programmer exposes in the interface respect the frame property for that resource, then we can treat the library's concept of resource separation as an instance of our ambient notion of separation: the tensor product.

Concretely, suppose that we have a type $A : \sigma \to \circ$, representing an affine capability indexed by a monoid σ, along with an operation $f : \forall X : \sigma.\ A(Y_1 \cdot X) \multimap A(Y_2 \cdot X)$. The type of f asserts that it can take the (logical) resource Y_1 to Y_2, and that in so doing, it preserves the frame X. If we knew that f were the only operation transforming capabilities of the form $A(t)$, then it would follow that we could split a capability $A(X \cdot Y)$ into two parts $A(X) \otimes A(Y)$, and manipulate them independently, since the only operation transforming capabilities of the form $A(t)$ is f, and f is parametric in the frame. By taking a value of type $A(X)$ and using it to construct a new abstract type, on which *only* frame-respecting operations are allowed, we can safely share an affine capability.

The sharing rule, given in Figure 7, formalizes this idea. We assert the existence of a type constructor $A : \sigma \to \circ$, where σ is a commutative monoid, and an initial resource $e : [A\ t]$, together with a family of frame-respecting, state-passing operations v_i, which take in an argument of type B_i and a state of type $[A(t_i \cdot X)]$, and return a result of type C_i and a state $[A(t_i' \cdot X)]$.[1] The full type of v_i includes additional index quantifications, which are useful for asserting propositions that connect the input and initial state or output and final state; in our examples, we suppress unused elements of this general type whenever we do not use them. The sharing operator returns a new existential type, exporting the v_i operations together with split, join and promote operations. Splitting and joining allow treating the monoidal composition as a tensor product. The promote operator takes any resource value

[1] Note that all the types of the form $A(t)$ are in proof-irrelevance brackets—this ensures that e represents a logical capability with no dynamic content, which turns out to be useful in the proof of soundness (Section 4). That said, it is possible to lift this restriction at the cost of a more complex implementation of the sharing rule. See footnote 4 in Section 4.

indexed by an idempotent value (*i.e.,* where $X = X \cdot X$), and returns a freely duplicable value.

The type constraints on the operations v_i statically ensure that $\exists X : \sigma.\ A(X)$ holds as an invariant at the beginning and ending of each call. However, if an operation v_i is passed *itself* as an argument, whether directly or indirectly, it may end up *calling* itself when its internal state does not satisfy the invariant; this is the well-known problem of *re-entrant calls* [29, 44] in higher-order imperative programs. One way to address this issue, embodied in Pottier's *anti-frame rule* [32], is to statically check that the invariant holds continuously, but this solution is often too restrictive [29]. We follow Pilkiewicz and Pottier [29] in preventing reentrancy *dynamically*, using a lock. Thus, the operational semantics of the sharing rule, given in Figure 6, is not a pure no-op, and shows how we rely on a *combination* of static and dynamic checking to enforce type safety. Sharing creates a flag variable (the lock), and wraps each operator with code to test the lock and to diverge if it is already held.

The remainder of this section gives a series of examples using our sharing rule to introduce custom notions of resource, culminating with an idealized memory allocator.

Weak References Sharing enables us to model ML-style weak references of type $!A$ that can be aliased. Suppose we have a location $X : \mathsf{Loc}$, a duplicable pointer of type $!\mathsf{ptr}\ X$ and an affine capability of type $\mathsf{cap}\ X\ !A$, and we wish to define *freely duplicable* functions to dereference and assign the location X, with types $!(1 \multimap\ !A)$ and $!(!A \multimap 1)$ respectively. The key idea is to use the share operator to allow these duplicable functions to close over the affine capability. First, we wrap the built-in operators $\mathsf{get}_c\ l$ and $l :=_c v$ in the following functions get_0 and set_0 whose return types resemble those of the second argument of the construct $\mathsf{share}(\_, \_)$. Typing these functions requires the equivalence $[\mathsf{cap}\ X\ !A] \equiv \mathsf{cap}\ X\ !A$, which our semantic model validates.

$$get_0 : \forall X : \mathsf{Loc}.\ !\mathsf{ptr}\ X \multimap\ !([\mathsf{cap}\ X\ !A] \multimap\ !A \otimes [\mathsf{cap}\ X\ !A])$$
$$get_0 = \lambda !l.\ !\lambda c.\ \mathsf{let}\ \langle !v, c\rangle = \mathsf{get}_c\ l\ \mathsf{in}\ \mathsf{let}\ c = (l :=_c !v)\ \mathsf{in}\ \langle !v, c\rangle$$

$$set_0 : \forall X : \mathsf{Loc}.\ !\mathsf{ptr}\ X \multimap\ !(!A \otimes [\mathsf{cap}\ X\ !A] \multimap [\mathsf{cap}\ X\ !A])$$
$$set_0 = \lambda !l.\ !\lambda\, \langle !v, c\rangle\ .\ \mathsf{let}\ \langle !dummy, c\rangle = \mathsf{get}_c\ l\ \mathsf{in}\ l :=_c !v$$

For any expression $e : !\mathsf{ptr}\ X$, $get_0\ \ e$ and $set_0\ \ e$ have types $!([\mathsf{cap}\ X\ !A] \multimap\ !A \otimes [\mathsf{cap}\ X\ !A])$ and $!(!A \otimes [\mathsf{cap}\ X\ !A] \multimap [\mathsf{cap}\ X\ !A])$, which essentially match the structure of the types spec_i in the definition of the sharing rule. Next, we define the monoid that encodes the logical state of the weak reference we are defining. Since the resource invariant for a weak reference is *fixed* and just states that the reference points to something of type $!A$, we choose the unit monoid $M = (1, \epsilon \stackrel{\text{def}}{=} \langle\rangle, (\cdot) \stackrel{\text{def}}{=} \lambda(x, y).\langle\rangle)$, and we interpret it by instantiating the capability operator A in Figure 7 with $C\ \langle\rangle \stackrel{\text{def}}{=} \mathsf{cap}\ X\ !A$. With these preliminaries, we can apply the sharing rule as follows:

$$\mathsf{share\_ref}\ \langle !l, c\rangle = \mathsf{let}\ \langle !g, !s\rangle = \langle get_0\ !l, set_0\ !l\rangle\ \mathsf{in}\ \mathsf{share}(c, g, s)$$
$$\mathsf{share\_ref} : \forall X : \mathsf{Loc}.\ !\mathsf{ptr}\ X \otimes [\mathsf{cap}\ X\ !A] \multimap$$
$$\exists \alpha : 1 \to \circ.$$
$$[\alpha\ \langle\rangle] \otimes !\mathsf{getType} \otimes !\mathsf{setType}$$
$$\otimes\ !\mathsf{splitT} \otimes !\mathsf{joinT} \otimes !\mathsf{promoteT}$$

where

$$\begin{array}{lll} \mathsf{getType} & \stackrel{\text{def}}{=} & \forall X : 1.\ [\alpha\ X] \multimap\ !A \otimes [\alpha\ X] \\ \mathsf{setType} & \stackrel{\text{def}}{=} & \forall X : 1.\ !A \otimes [\alpha\ X] \multimap [\alpha\ X] \\ \mathsf{promoteT} & \stackrel{\text{def}}{=} & \forall X : 1 :: X = X \cdot X.\ [\alpha\ X] \multimap\ ![\alpha\ X] \end{array}$$

Finally, the unit monoid is idempotent by definition, so we can apply the promote operator to any value of type $[\alpha\ \ t]$ (for any $t : 1$). This allows us to construct the following function that, given a duplicable pointer and an affine capability to it, returns two duplicable functions to read and write to it:

$$\mathsf{MLref} : \forall X : \mathsf{Loc}.\ !\mathsf{ptr}\ X \otimes [\mathsf{cap}\ X\ !A] \multimap\ !(1 \multimap\ !A) \otimes\ !(!A \multimap 1)$$
$$\mathsf{MLref}\ \langle !l, c\rangle =$$
$$\mathsf{let}\ \langle q, !get, !set, \_, \_, !promote\rangle = \mathsf{share\_ref}(\langle !l, c\rangle)\ \mathsf{in}$$
$$\mathsf{let}\ !r = promote(q)\ \mathsf{in}$$
$$\mathsf{let}\ deref = !(\lambda\langle\rangle.\ \mathsf{let}\ \langle v, \_\rangle = get(r)\ \mathsf{in}\ v)\ \mathsf{in}$$
$$\mathsf{let}\ setref = !(\lambda a.\ \mathsf{let}\ \_ = set(a, r)\ \mathsf{in}\ \langle\rangle)\ \mathsf{in}$$
$$\langle deref, setref\rangle$$

Monotonic Counters Next, we show how to construct shared monotonic counters that can be freely incremented by all clients. Since clients can only increment the counter, the local knowledge of each client provides a *lower bound* on the counter's actual value. Suppose our counter is stored at a location $X : \mathsf{Loc}$. We start by defining a simple and standard increment function, next, that takes as argument a pointer l of type $!\mathsf{ptr}\ X$ and a capability of type $\mathsf{cap}\ X\ !(\mathsf{nat}\ n)$, increments the counter, and returns $n + 1$ and a capability of type $\mathsf{cap}\ X\ !(\mathsf{nat}\ (n + 1))$. (We assume here that $+$ is a primitive operation taking unrestricted values of type $\mathsf{nat}\ m_1$ and $\mathsf{nat}\ m_2$ and returning an expression of type $!\mathsf{nat}\ (m_1 + m_2)$.)

$$\mathsf{next}\ \langle !l, c\rangle = \mathsf{let}\ (!n, c) = \mathsf{get}_c\ l\ \mathsf{in}$$
$$\mathsf{let}\ c = (l :=_c n + 1)\ \mathsf{in}\ \langle n + 1, c\rangle$$
$$\mathsf{next} : \forall n.\ !\mathsf{ptr}\ X \otimes \mathsf{cap}\ X\ !(\mathsf{nat}\ n)$$
$$\multimap\ !\mathsf{nat}\ (n + 1) \otimes \mathsf{cap}\ X\ !(\mathsf{nat}\ (n + 1))$$

We wish to share the counter by passing the function next as the second argument of the $\mathsf{share}(\_, \_)$ operator. To do that, we must massage the type of next into a compatible form, capturing the fact that, once the counter is aliased, its local knowledge only provides a lower bound on its value. We define the monoid $M = (\mathbb{N}, \epsilon \stackrel{\text{def}}{=} 0, (\cdot) \stackrel{\text{def}}{=} \max)$, the type $C(n) \stackrel{\text{def}}{=} \mathsf{cap}\ X\ !(\mathsf{nat}\ n)$ (to correspond to the type A in Figure 7) and observe that next can also be given the following *weaker* type:

$$\mathsf{next} : \mathsf{nextType}(C)$$
$$\mathsf{nextType}(\alpha) = \quad \forall Y, Z : \mathbb{N}.\ \ !\mathsf{ptr}\ X \otimes [\alpha(Z \cdot Y)]$$
$$\multimap \exists U : \mathbb{N} :: U > Z.\ !\mathsf{nat}\ U \otimes [\alpha(U \cdot Y)]$$

The weaker type, $\mathsf{nextType}(C)$, only asserts that if the initial value of the counter is $\max(Z, Y)$, then its value after the next operation is $\max(U, Y)$, for some $U > Z$. Intuitively, Z is the local context's initial lower bound on the counter, Y is the frame's lower bound on the counter, and U is the local context's lower bound on the counter after the increment operation. This weaker type is exactly in the form of the second argument of $\mathsf{share}(\_, \_)$, so we can define a counter sharing function that creates an abstract, shared counter from a given capability to $X : \mathsf{Loc}$ and the next function.

$$\mathsf{mkCnt}\ c = \mathsf{share}(c, \mathsf{next})$$
$$\mathsf{mkCnt} : \forall\ X : \mathsf{Loc}, Y : \mathbb{N}.\ [\mathsf{cap}\ X\ !(\mathsf{nat}\ Y)] \multimap$$
$$\exists \alpha : \mathbb{N} \to \circ.$$
$$[\alpha\ Y] \otimes\ !\mathsf{nextType}(\alpha) \otimes\ !\mathsf{splitT} \otimes !\mathsf{joinT} \otimes !\mathsf{promoteT}$$

Since $\max(x, x) = x$, every element of our monoid is idempotent, so we can take any counter and make it freely duplicable using the resulting function of type promoteT (as in the previous example). This permits multiple clients to make use of the same counter. Each client knows that its own use of the counter will yield monotonically increasing elements, and does not have to worry about interference with other clients of the counter.

Fractional Permissions We provide an encoding of fractional permissions that is parametric in the underlying affine resource that we wish to share. Let σ be an index sort, and let $\alpha : \sigma \to \circ$ be the type of an affine resource on top of which we want to layer a fractional permissions algebra. For example, to model fractional permissions over ref cells of type A, we could choose $\sigma = \mathsf{Loc}$ and $\alpha\ \ X = \mathsf{cap}\ X\ A$. We define a sort of fractional (rational) numbers, called Frac, and a sort of fractional permissions over σ, called $\mathsf{FPerm}(\sigma)$:

$$\begin{array}{lll} \mathsf{Frac} & \stackrel{\text{def}}{=} & \{a \in \mathbb{Q}\ \mid\ 0 < a \leq 1\} \\ \mathsf{FPerm}(\sigma) & \stackrel{\text{def}}{=} & \{\epsilon, \bot, \mathsf{Empty}\} \cup \{(a, m)\ \mid\ a \in \mathcal{S}[\![\mathsf{Frac}]\!], m \in \mathcal{S}[\![\sigma]\!]\} \end{array}$$

A fractional permission is either ϵ (essentially a 0 permission), $\bot$ (for an invalid permission), Empty (denoting that there is no resource currently in place to be fractionally shared), or (a, m) (denoting fractional permission a to the resource represented by m). Here, $\mathcal{S}[\![\sigma]\!]$ denotes the set of elements in the sort σ. Fractional permissions form a monoid M with unit ϵ and operation $(\cdot)$ defined on non-unit elements as follows:

$$(a,m)\cdot(a',m') = \begin{cases}(a+a',m) & 0 < a+a' \leq 1,\ m = m'\\ \bot & \text{otherwise}\end{cases}$$

$$\mathsf{Empty}\cdot x = \begin{cases}\mathsf{Empty} & x = \epsilon\\ \bot & \text{otherwise}\end{cases}$$

$$\bot \cdot x = \bot$$

Next, we define the affine type family FracTy_α which we actually share (this type family is called A in Figure 7). As required by the sharing rule, the type is indexed by the monoid $\mathsf{FPerm}(\sigma)$. Here, void denotes the empty type $\exists X : \mathbb{N} :: \bot.\ 1$.

$$\mathsf{FracTy}_\alpha\ \epsilon \stackrel{\text{def}}{=} \mathsf{void} \quad \mathsf{FracTy}_\alpha\ \bot \stackrel{\text{def}}{=} \mathsf{void} \quad \mathsf{FracTy}_\alpha\ \mathsf{Empty} \stackrel{\text{def}}{=} 1$$

$$\mathsf{FracTy}_\alpha\ (a,m) \stackrel{\text{def}}{=} \begin{cases}\alpha\ m & \text{when } a = 1\\ \mathsf{void} & \text{when } a \neq 1\end{cases}$$

Notice that this type family is uninhabitable except at the extremes. This is important because, concretely, either the whole resource will be available to the fractional permissions module as hidden state, or nothing will be, even though the fractional permissions superficially represent partial ownership.

We now show how to represent fractional permissions over the affine type α when α supports only one fractionally-shareable operation, readonlyop, that maps $\alpha\ M$ to $\alpha\ M$, possibly with auxiliary inputs and outputs (our construction generalizes very easily when there is more than one operation). Let this only operation, readonlyop, have type ReadOnlyOp defined by:

$$\begin{aligned}&\mathsf{ReadOnlyOp} \stackrel{\text{def}}{=}\\ &\quad\forall X : \mathsf{FPerm}(\sigma).\ \forall Y : \sigma' \times \mathsf{Frac} :: P.\\ &\qquad\beta\ (\pi_1(Y)) \otimes [\mathsf{FracTy}_\alpha\ ((\pi_2(Y), M)\cdot X)] \multimap\\ &\qquad\quad\exists Z : \sigma'' :: Q.\ \gamma\ Z \otimes [\mathsf{FracTy}_\alpha\ ((\pi_2(Y), M)\cdot X)]\end{aligned}$$

This type has been constructed specifically to match the "spec" type for the sharing rule, and thereby provide maximal generality. We quantify over an arbitrary fraction as the second component of Y.

We can directly apply the $\mathsf{share}(\_,\_)$ operator with this operation as the second argument to obtain a shareable abstract type α' but, to make the fractional permissions useful, we would also like to provide two operations that allow clients to exchange "full" resources for "full" fractional permissions and vice versa. These two operations should have types defined below:

$$\begin{aligned}\mathsf{ToFrac} &\stackrel{\text{def}}{=} \forall X : \sigma.\ [\alpha\ X] \otimes [\alpha'\ \mathsf{Empty}] \multimap [\alpha'\ (1, X)]\\ \mathsf{FromFrac} &\stackrel{\text{def}}{=} \forall X : \sigma.\ [\alpha'\ (1, X)] \multimap [\alpha\ X] \otimes [\alpha'\ \mathsf{Empty}]\end{aligned}$$

Accordingly, we would like to pass to $\mathsf{share}(\_,\_)$ two additional operations of types $\mathsf{ToFrac}[\mathsf{FracTy}_\alpha/\alpha']$ and $\mathsf{FromFrac}[\mathsf{FracTy}_\alpha/\alpha']$, respectively. Fortunately, given our definition of FracTy_α, these operations can be trivially defined as $\lambda\langle x, \langle\rangle\rangle.\ x$ and $\lambda x.\ \langle x, \langle\rangle\rangle$.

Tying everything together, we now define the following term mkFrac, a generic (polymorphic) module for layering fractional permissions over a resource α. The module provides an empty fractional permission at the outset, which can then be transferred to a full permission using ToFrac and back using FromFrac. (The type FracOp is defined as ReadOnlyOp with α' in place of FracTy_α.)

$$\begin{aligned}&\mathsf{mkFrac} = \lambda!f.\ \mathsf{share}(\langle\rangle, f,\ \lambda\langle x, \langle\rangle\rangle.\ x,\ \lambda x.\ \langle x, \langle\rangle\rangle)\\ &\mathsf{mkFrac} :\\ &\quad\forall\alpha : \sigma \to \circ.\ \forall\beta : \sigma' \to \circ.\ \forall\gamma : \sigma'' \to \circ.\\ &\qquad!\mathsf{ReadOnlyOp}\\ &\qquad\quad\multimap \exists\alpha' : \mathsf{FPerm}(\sigma) \to \circ.\\ &\qquad\qquad[\alpha'\ \mathsf{Empty}] \otimes !\mathsf{FracOp} \otimes !\mathsf{ToFrac} \otimes !\mathsf{FromFrac} \otimes\\ &\qquad\qquad\qquad!\mathsf{splitT} \otimes !\mathsf{joinT} \otimes !\mathsf{promoteT}\end{aligned}$$

Memory Allocator We now give a stylized memory allocator with a non-monotonic resource invariant, inspired by (but much simpler than) Wickerson *et al.*'s [43] proof of the Unix malloc function, and show how the allocator can be shared safely. The basic idea is that the memory allocator's free list is represented by an array, each entry of which contains a pair of a boolean flag and a location; the flag is true when the location is free, and false when it has been allocated to a client. For free locations, the allocator also owns a capability to access the memory of that location. For the allocated locations, it does not. To formalize this idea, we first assume a family of types for affine arrays:

$$\begin{aligned}&\mathsf{arr}_A : \mathsf{Loc} \times \mathbb{N} \times (\mathbb{N} \to \sigma) \to \circ\\ &\mathsf{alength}_A : \forall X : \mathsf{Loc}, n : \mathbb{N}, f : \mathbb{N} \to \sigma.\\ &\qquad!\mathsf{ptr}\ X \otimes [\mathsf{arr}_A(X, n, f)] \multimap !\mathsf{nat}\ n \otimes [\mathsf{arr}_A(X, n, f)]\\ &\mathsf{aswap}_A : \forall X : \mathsf{Loc}, n : \mathbb{N}, f : \mathbb{N} \to \sigma, i : \mathbb{N}, x : \sigma :: i < n.\\ &\qquad!\mathsf{ptr}\ X \otimes !\mathsf{nat}\ i \otimes A(x) \otimes [\mathsf{arr}_A(X, n, f)]\\ &\qquad\quad\multimap A(f\ i) \otimes [\mathsf{arr}_A(X, n, \lambda j.\ \mathsf{if}(i = j, x, f(j)))]\\ &\mathsf{aread}_A : \forall X : \mathsf{Loc}, n : \mathbb{N}, f : \mathbb{N} \to \sigma, i : \mathbb{N}, x : \sigma :: i < n.\\ &\qquad!\mathsf{ptr}\ X \otimes !\mathsf{nat}\ i \otimes (A(f\ i) \multimap B \otimes A(f\ i))\\ &\qquad\quad\otimes [\mathsf{arr}_A(X, n, f)] \multimap B \otimes [\mathsf{arr}_A(X, n, f)]\end{aligned}$$

Here, A is a σ-indexed type constructor, and the index information for the array of type $\mathsf{arr}_A(X, n, f)$ consists of its location X, its length n and a function f, such that for each $i < n$, the i-th element of the array contains a value of type $A(f\ i)$. So f serves as a representation function for the array. To modify the array, we make use of a swapping operation aswap_A, which takes an array pointer, an index, a value, and a memory capability for the array, and uses it to replace the contents of that index. In the process, it also updates the representation function f. To read the array, we make use of a reading function aread_A, which takes an array pointer, an index, an array capability, and an observer function, which takes a value at the given location and returns the array capability plus an observation of type B.

To specialize this to the memory allocator ADT we described briefly above, we choose $\sigma = 2 \times \mathsf{Loc}$, where $2 = \{\mathsf{tt}, \mathsf{ff}\}$ and $A = \mathsf{contents}$, which is defined below:

$$\begin{aligned}&\mathsf{contents} : (2 \times \mathsf{Loc}) \to \circ\\ &\mathsf{contents}(\mathsf{tt}, X) = !\mathsf{bool}\ \mathsf{tt} \otimes !\mathsf{ptr}\ X \otimes [\mathsf{cap}\ X\ 1]\\ &\mathsf{contents}(\mathsf{ff}, X) = !\mathsf{bool}\ \mathsf{ff} \otimes !\mathsf{ptr}\ X \otimes 1\\ &\\ &\mathsf{freelist} : \mathsf{Loc} \times \mathbb{N} \times (\mathbb{N} \to (2 \times \mathsf{Loc})) \to \circ\\ &\mathsf{freelist}(X, n, f) = \exists\_ :: \mathsf{inj}(\pi_2 \circ f).\ [\mathsf{arr}_{\mathsf{contents}}(X, n, f)]\end{aligned}$$

We define the type freelist (the type of the free list of our memory allocator) as an array of contents, with the invariant that the second projection of the representation function f be injective (*i.e.,* the array has at most one entry for each location). The type operator contents takes a boolean b and a location X, where b reflects whether X is free. The capability to access X is held in the array contents only for free cells.

Next, we use the function aswap to define functions malloc_at and free_at to allocate and free locations at a particular index in the free list, respectively. The function malloc_at takes an index i in the free list, which maps to location Y, a proof that the location is free ($f(i) = (\mathsf{tt}, Y)$) and returns the capability of type $[\mathsf{cap}\ Y\ 1]$ stored in the location, swapping it with a unit value. free_at does the opposite.

$$\begin{aligned}&\mathsf{flag\_loc} : \forall b : 2, l : \mathsf{Loc}.\\ &\qquad\mathsf{contents}(b, l) \multimap (!\mathsf{bool}\ b \otimes !\mathsf{ptr}\ l) \otimes \mathsf{contents}(b, l)\\ &\mathsf{flag\_loc}\ \langle !b, !l, m\rangle = \langle\langle !b, !l\rangle, \langle !b, !l, m\rangle\rangle\\ &\\ &\mathsf{malloc\_at} :\\ &\quad\forall X, Y : \mathsf{Loc}, n : \mathbb{N}, f : \mathbb{N} \to (2 \times \mathsf{Loc}), i : \mathbb{N} :: i < n \wedge f(i) = (\mathsf{tt}, Y).\\ &\qquad!\mathsf{ptr}\ X \otimes !\mathsf{nat}\ i \otimes \mathsf{freelist}(X, n, f)\\ &\qquad\multimap !\mathsf{ptr}\ Y \otimes [\mathsf{cap}\ Y\ 1] \otimes \mathsf{freelist}(X, n, \lambda j.\ \mathsf{if}(i = j, (\mathsf{ff}, Y), f(j)))\end{aligned}$$

$$
\begin{array}{l}
\mathsf{malloc} : \forall S : \mathcal{P}(\mathsf{Loc})_\perp, X : \mathsf{Loc}. \\
\qquad !\mathsf{ptr}\ X \otimes [C(S)] \\
\qquad \multimap \exists Y : \mathsf{Loc}.\ !\mathsf{ptr}\ Y \otimes [\mathsf{cap}\ Y\ 1] \otimes [C(S \cdot \{Y\})] \\
\mathsf{malloc}(!a, m) = \\
\quad \mathsf{let}\ (!n, m) = \mathsf{alength}(!a, m)\ \mathsf{in} \\
\quad \mathsf{let\ rec}\ \mathit{loop}(m, !i) = \\
\qquad \mathsf{if}\ i < n\ \mathsf{then} \\
\qquad\quad \mathsf{let}\ \langle \langle !b, !l \rangle, m \rangle = \mathsf{aread}(!a, !i, \mathsf{flag\_loc}, m)\ \mathsf{in} \\
\qquad\quad \mathsf{if}(b, \mathsf{malloc\_at}(!a, !i, m), \mathit{loop}(m, i+1)) \\
\qquad \mathsf{else} \\
\qquad\quad (\mathsf{fix}\ f(x).\ f\ x)\ \langle\rangle \\
\quad \mathsf{in}\ \mathit{loop}(m, !0)
\end{array}
$$

$$
\begin{array}{l}
\mathsf{free} : \forall S : \mathcal{P}(\mathsf{Loc})_\perp, X : \mathsf{Loc}, Y : \mathsf{Loc}. \\
\qquad !\mathsf{ptr}\ X \otimes !\mathsf{ptr}\ Y \otimes [\mathsf{cap}\ Y\ 1] \otimes [C(S \cdot \{Y\})] \multimap [C(S)] \\
\mathsf{free}\ (!a, !l, c, m) = \\
\quad \mathsf{let}\ (!n, m) = \mathsf{alength}(!a, m)\ \mathsf{in} \\
\quad \mathsf{let\ rec}\ \mathit{loop}(c, m, !i) = \\
\qquad \mathsf{let}\ \langle \langle !b, !l' \rangle, m \rangle = \mathsf{aread}(!a, !i, \mathsf{flag\_loc}, m)\ \mathsf{in} \\
\qquad \mathsf{if}(l = l', \mathsf{free\_at}(!a, !i, c, m), \mathit{loop}(c, m, i+1)) \\
\quad \mathsf{in}\ \mathit{loop}(c, m, !0)
\end{array}
$$

Figure 8. The Memory Allocator

$$
\begin{array}{l}
\mathsf{malloc\_at}(!a, !i, m) = \\
\quad \mathsf{let}\ \langle \langle !b, !l \rangle, m \rangle = \mathsf{aread}(!a, !i, \mathsf{flag\_loc}, m)\ \mathsf{in} \\
\quad \mathsf{let}\ \langle \langle !b, !l, c \rangle, m \rangle = \mathsf{aswap}(!a, !i, \langle !\mathsf{ff}, !l, \langle\rangle \rangle, m)\ \mathsf{in}\ \langle !l, c, m \rangle
\end{array}
$$

$$
\begin{array}{l}
\mathsf{free\_at} : \\
\quad \forall X, Y : \mathsf{Loc}, n : \mathbb{N}, f : \mathbb{N} \to (2 \times \mathsf{Loc}), i : \mathbb{N} :: i < n \wedge f(i) = (\mathsf{ff}, Y). \\
\qquad !\mathsf{ptr}\ X \otimes !\mathsf{nat}\ i \otimes [\mathsf{cap}\ Y\ 1] \otimes \mathsf{freelist}(X, n, f) \\
\qquad \multimap\ \mathsf{freelist}(X, n, \lambda j.\ \mathsf{if}(i = j, (\mathsf{tt}, Y), f(j)))
\end{array}
$$

$$
\begin{array}{l}
\mathsf{free\_at}(!a, !i, c, m) = \\
\quad \mathsf{let}\ \langle \langle !b, !l \rangle, m \rangle = \mathsf{aread}(!a, !i, \mathsf{flag\_loc}, m)\ \mathsf{in} \\
\quad \mathsf{let}\ \langle \langle !b, !l, \langle\rangle \rangle, m \rangle = \mathsf{aswap}(!a, !i, \langle !\mathsf{tt}, !l, c \rangle, m)\ \mathsf{in}\ m
\end{array}
$$

Next, we consider the monoid $\mathcal{P}(\mathsf{Loc})_\perp$, whose elements are sets of locations. The unit is the empty set $\epsilon = \emptyset$, and concatenation is defined by disjoint union $\uplus$, with non-disjoint sets going to $\perp$. We use this monoid to define the type $C(S)$ for a given location X pointing to the head of the free list:

$$
\begin{array}{l}
C(\perp) = \mathsf{void} \\
C(S) = \exists n : \mathbb{N}, f : \mathbb{N} \to (2 \times \mathsf{Loc}) :: S = \{l \mid \exists i < n.\ f(i) = \langle \mathsf{ff}, l \rangle\}. \\
\qquad \mathsf{freelist}(X, n, f)
\end{array}
$$

Intuitively, for $S \neq \perp$, $C(S)$ is a free list whose allocated pointers coincide exactly with S.

Using $C(S)$, we can define operations malloc and free (Figure 8). The malloc operation traverses the free array until it finds an unallocated element, updates the flag, and returns that element. If the free list is fully allocated, then we go into an infinite loop — more realistic implementations would signal an error or resize the free list. The free operation also iterates over the array until it finds the element it was passed as an argument, but it does *not* have to perform a bounds check as it iterates: the type $C(S \cdot \{Y\})$ guarantees that the location Y will be found in the free list, and hence that i is always in bounds. Note that the type of the location comparison operation $=$ used in free is $\forall X, Y : \mathsf{Loc}.\ \mathsf{ptr}\ X \otimes \mathsf{ptr}\ Y \multimap \mathsf{bool}\ (X = Y)$.

More sophisticated versions of this pattern arise frequently in the implementation of free lists, connection pools, and other resource managers. The critical feature of our invariant is that we can only free a piece of memory if it originally came from *this* memory allocator in the first place. Furthermore, it is a non-monotonic invariant, since the same piece of memory can go in and out of the free list, which means that the size of the free list in the predicate can grow and shrink as the program executes.

However, we can nevertheless share the memory allocator, since the frame conditions on the specifications express the constraint that interference between different clients is benign—up to partial correctness, no client cares what allocations or deallocations other clients perform:

$$
\begin{array}{l}
\mathsf{mkAllocator} : \forall X, n, f, S :: S = \{l \mid \exists i < n.\ f(i) = \langle \mathsf{ff}, l \rangle\}. \\
\qquad \mathsf{freelist}(X, n, f) \\
\qquad \multimap \exists \alpha : \mathcal{P}(\mathsf{Loc})_\perp \to \circ. \\
\qquad\quad [\alpha(S)] \otimes\ !\mathsf{mallocType} \otimes !\mathsf{freeType} \\
\qquad\quad !\mathsf{splitT} \otimes !\mathsf{joinT} \otimes !\mathsf{promoteT} \\
\mathsf{mkAllocator}\ m = \mathsf{share}(m, \mathsf{malloc}, \mathsf{free})
\end{array}
$$

The memory manager's state can be split up and shared among many different clients. The key is to observe that for any state S, we know that $\alpha(S) = \alpha(S \uplus \emptyset)$. Thus we can pass each client a copy of $\alpha(\emptyset)$, which it can use to allocate and free locally-owned memory without knowledge of the allocation behavior of other clients.

$$
\begin{array}{rcl}
\mathrm{World}_n & \stackrel{\text{def}}{=} & \left\{ W = (k, \omega) \;\middle|\; \begin{array}{l} k < n, \exists j.\ \omega \in \mathrm{Island}_k^{j+1} \\ \omega[0] = \mathrm{HIsland}_k \end{array} \right\} \\
\mathrm{Island}_n & \stackrel{\text{def}}{=} & \left\{ \iota = (M, \cdot, \epsilon, I) \;\middle|\; \begin{array}{l} (M, \cdot, \epsilon)\ \text{comm. monoid}, \\ I \in M \to \mathrm{ResPred}_n \end{array} \right\} \\
\mathrm{HIsland}_n & \stackrel{\text{def}}{=} & \left(\begin{array}{l} \mathrm{Heap}_\perp, \uplus, \emptyset, \\ \lambda h. \{(W, \epsilon) \mid W \in \mathrm{World}_n, h \neq \perp\} \end{array} \right) \\
\mathrm{ResPred}_n & \stackrel{\text{def}}{=} & \left\{ \varphi \subseteq \mathrm{ResAtom}_n \;\middle|\; \begin{array}{rl} \forall W' \sqsupseteq W.\ (W', r) \in \varphi \\ \implies & (W', r) \in \varphi \end{array} \right\} \\
\mathrm{ResAtom}_n & \stackrel{\text{def}}{=} & \left\{ (W, r) \;\middle|\; \begin{array}{l} W \in \mathrm{World}_n,\ \forall i.\ a_i \in W.\omega[i].M, \\ r = (a_0, \ldots, a_{m-1}), m = |W.\omega| \end{array} \right\} \\
\mathrm{ValPred} & \stackrel{\text{def}}{=} & \left\{ V \subseteq \mathrm{ValAtom} \;\middle|\; \begin{array}{l} \forall W' \sqsupseteq W.\ (W, (r, v)) \in V \\ \Rightarrow \forall r'. (W', (r \cdot r', v)) \in V \end{array} \right\} \\
\mathrm{ValAtom} & \stackrel{\text{def}}{=} & \{(W, (r, v)) \mid \exists n.\ (W, r) \in \mathrm{ResAtom}_n \}
\end{array}
$$

$$
\begin{array}{rcl}
\triangleright(k+1, \omega) & \stackrel{\text{def}}{=} & (k, \lfloor \omega \rfloor_k) \\
\lfloor (\iota_1, \ldots, \iota_n) \rfloor_k & \stackrel{\text{def}}{=} & (\lfloor \iota_1 \rfloor_k, \ldots, \lfloor \iota_n \rfloor_k) \\
\lfloor (M, \cdot, \epsilon, I) \rfloor_k & \stackrel{\text{def}}{=} & (M, \cdot, \epsilon, \lambda a. \lfloor I(a) \rfloor_k) \\
\lfloor \varphi \rfloor_k & \stackrel{\text{def}}{=} & \{(W, r) \in \varphi \mid W.k < k\}
\end{array}
$$

$$
\begin{array}{rcl}
(\iota'_1, \ldots, \iota'_{n'}) \sqsupseteq\ (\iota_1, \ldots, \iota_n) & \stackrel{\text{def}}{=} & n' \geq n,\ \ \forall i \leq n.\ \iota'_i = \iota_i \\
(k', \omega') \sqsupseteq_j (k, \omega) & \stackrel{\text{def}}{=} & k' = k - j,\ \ \omega' \sqsupseteq \lfloor \omega \rfloor_{k'}
\end{array}
$$

$$
\begin{array}{rcl}
(s, r) : W & \stackrel{\text{def}}{=} & s = s_0 \cdot \ldots \cdot s_{m-1},\ m = |W.\omega|, \\
& & \forall i \in 0..m-1.\ \ (\triangleright W, s_i) \in W.\omega[i].I((s \cdot r)[i])
\end{array}
$$

Figure 9. Possible Worlds and Related Definitions

4. The Semantic Model

In this section, we justify the soundness of our type system. The main challenge, of course, is validating the sharing rule. We gain traction by characterizing the behavior of well-typed terms through a step-indexed Kripke logical relation (SKLR). While SKLRs have been used previously to give clean semantic soundness proofs of related substructural calculi [5], ours is novel in its treatment of resources. We therefore begin by laying down some conceptual groundwork and terminology concerning resources.

Physical vs. Logical Resources and the Global Store In the beginning, there is the heap: it is a primitive, *physical* notion of splittable resource, and in the absence of sharing there is little more to say. The affine heap capability $\mathsf{cap}\ \ell\ A$ gives its "owner"—*i.e.,* the term that consumes it—full control over the location ℓ and its contents, and the lack of sharing means that no other parts of the program may contain any knowledge about ℓ or its contents at all.

Each application of the sharing rule, however, introduces a new *logical* notion of splittable resource, represented as a commutative monoid $(M, \cdot, \epsilon)$, which governs access to a piece of shared state. Control over resources of type M becomes a new type of affine capability (written $[\alpha\ \ t]$ in the sharing rule in Figure 7), which

may be consumed by or transferred between different parts of the program just as heap capabilities can.[2] Unlike the heap, which has a direct physical interpretation, M must be given an interpretation in terms of what invariants it imposes on the underlying shared state. Specifically, the capability $[A\ t]$ in Figure 7 describes the invariant that holds of the shared state when the *global store* of M (*i.e.,* the monoidal composition of all resources of type M that are currently in existence) is t. For those readers with a Hoare-logic background, it may be helpful to think of this global store of M as a kind of "ghost state" [22] that instruments the physical heap state with extra logical information.

Atomic vs. Composite Resources As a program executes, a new logical resource is created each time the sharing rule is executed, extending the resource set (which begins life with only the lone physical resouce of the heap). We will say that a resource belonging to any one of these types is an *atomic* resource.

Of course, a term may naturally own many different atomic resources, as a result of being composed from multiple different subterms. For example, it may own the heap capability $\mathsf{cap}\ \ell\ 1$ to control location ℓ, as well as the logical capability $[\alpha\ t]$ (where α is the abstract type constructor created by some application of the sharing rule). In this case, the term owns a physical heap resource ($[\ell : \langle\rangle]$), as well as a logical resource (t) of the monoidal resource type that was created along with α.

In general, a term may own resources of every type currently in existence (and later, when new types of resource are created, it can be implicitly viewed as owning the unit element of those resources). We call such a combination of resources of all the different atomic types a *composite* resource. Given that each atomic resource is a commutative monoid, observe that composite resources form a commutative monoid via the obvious product construction. For convenience, we overload $\cdot$ and write $r_1 \cdot r_2$ to denote the componentwise composition of two composite resources r_1 and r_2.

Composite resources are the fundamental currency of our model. Not only are they what terms consume and produce, but furthermore, when we apply the sharing rule to make some underlying (affine) resource shareable, that underlying resource is a composite resource, and the invariant that governs it takes the form of a predicate on composite resources.

Worlds and Islands Being a Kripke logical relation, our model (presented below) is indexed by *possible worlds*. In previous Kripke models of ML-like languages, these worlds have been used to encode invariants on the physical heap. Here, since we support logical as well as physical resources, we generalize worlds to encode (1) the knowledge of what types of logical resources have been created by applications of the sharing rule, and (2) how to interpret those logical resources as invariants on shared state.

As defined in Figure 9, worlds are tuples of *islands*, with each island describing a different type of resource.[3] (Ignore the "step indices" k and n for now; we explain them below.) An island comprises a commutative monoid $(M, \cdot, \epsilon)$, as well as a *representation invariant* I that interprets elements of M into assumptions (composite resource predicates) on the underlying shared state. Specifically, $I(t)$ denotes the invariant that holds on the shared state when the global store of the island's resource (M) is t.

[2] Note: even if the sharing rule is instantiated twice with the *same* monoid, it nevertheless generates two *distinct* types of logical resources. The distinction is enforced syntactically by the fact that each application of the sharing rule creates a fresh, existentially-quantified capability constructor α; even if two such α's (say, α_1 and α_2) are indexed by the same monoid, instantiations $[\alpha_1\ t]$ and $[\alpha_2\ t]$ will not be confused with each other.

[3] Throughout, we use dot notation like $W.k$ and $W.\omega$ to project named components from structures, and indexing notation like $\omega[i]$ to project the ith component from a tuple.

The first island (island 0) is fixed to be the built-in island for physical heaps ($\mathrm{HIsland}$). Its monoid is the standard partial commutative monoid on heaps, with disjoint union as composition and the empty heap as unit, completed to a total monoid with a bottom element $\bot$. Its representation invariant $I(h)$ is trivial—it asserts no ownership of any underlying shared resource because there is none, but is only satisfied if h is a heap and not $\bot$.

In the other islands, the representation invariant I is more interesting. First and foremost, it is *world-indexed*. For those readers familiar with recent SKLRs [16, 3], which employ similarly world-indexed *heap* invariants, the reason for this world-indexing will likely be self-explanatory: it's needed to account for the presence of higher-order state. For most other readers, it may appear completely mysterious, but it is also a technical point that the reader may safely gloss over (by skipping the next paragraph).

Briefly, the reason for the world-indexing of the resource predicates is as follows: in proving the sharing rule (see the end of this section), we extend the world with a new island, and we want to define its $I(t)$ to require (roughly) that the underlying shared resource of the island must justify the capability $[A\ t]$, where A is the capability constructor in the first premise of the sharing rule (Figure 7). But for arbitrary A, the question of whether some (composite) resource r justifies the capability $[A\ t]$ depends on what the "current" world W is when the question is asked, which might be at some point in the future when new invariants have been imposed by *future* islands. Such a situation would arise, for instance, were we to apply the sharing rule to create a "weak reference" (Section 3) to a value of function type, which is (not coincidentally) the canonical example of higher-order state. The solution is thus to parameterize the resource predicate $I(t)$ over W, knowing that the W parameter will always be instantiated (in the definition of "world satisfaction" below) with the "current" world.

This parameterization trick is by now a very standard move in the SKLR playbook for building models of higher-order state [3, 17]. However, it is also a prime example of Wheeler's adage that "all problems in computer science can be solved by another level of indirection, but that usually will create another problem." Indeed, an unfortunate consequence is that it causes a "bad" circularity in the construction of worlds that cannot be solved directly in sets. The *step-indexed* approach of Ahmed *et al.* [1, 2, 3] handles this problem by stratifying the construction of worlds by $n \in \mathbb{N}$ bounding the number of execution steps for which we observe the program, with n going down by 1 in the world parameter of the resource predicate. The details of this construction are entirely standard, as are the world approximation ($\lfloor\cdot\rfloor_k$) and later ($\triangleright$) operators in Figure 9, and interested readers are referred to the literature [3, 17].

In any case, the resource predicates in the range of I are required to be *monotonic*: adding new islands to a world cannot invalidate the invariants of previous islands (see the definition of $\mathrm{ResPred}$). Finally, when using a composite resource r with j atomic sub-resources in the context of a future world with $j + k$ islands, we silently assume the atomic sub-resources of the last k islands are ϵ.

Local vs. Shared Resources and World Satisfaction In reality, a term e executes under a global heap h. In our model, we think of e as executing, logically, under the *global composite store*, which comprises all the resources currently in existence: specifically, it combines the global store of every atomic resource in existence, including the heap (which is the 0-th island's resource). Some portion r of that global composite store is directly known to (and owned by) e itself—we call this e's *local resource*—while the remaining portion s constitutes the *shared resource*. The shared resource is so named because it is required to contain all the underlying shared resources needed to satisfy the representation invariants of all the islands in the world. (The *local* vs. *shared* terminology is bor-

$$\mathcal{K}[\![\circ]\!] \stackrel{\text{def}}{=} \text{ValPred} \qquad \mathcal{K}[\![\sigma \to \kappa]\!] \stackrel{\text{def}}{=} \mathcal{S}[\![\sigma]\!] \to \mathcal{K}[\![\kappa]\!]$$

$$\begin{array}{rcl}
\mathcal{V}[\![B\ t]\!]^W_\rho & \stackrel{\text{def}}{=} & \{(r, \mathcal{I}[\![t]\!]_\rho)\} \text{ for } B \in \{\mathsf{bool}, \mathsf{nat}, \mathsf{ptr}\} \\
\mathcal{V}[\![\mathsf{cap}\ t\ A]\!]^W_\rho & \stackrel{\text{def}}{=} & \{(r \cdot [\mathcal{I}[\![t]\!]_\rho : v], \bullet) \mid (r, v) \in \mathcal{V}[\![A]\!]^W_\rho\} \\
\mathcal{V}[\![1]\!]^W_\rho & \stackrel{\text{def}}{=} & \{(r, \langle\rangle)\} \\
\mathcal{V}[\![A_1 \otimes A_2]\!]^W_\rho & \stackrel{\text{def}}{=} & \{(r_1 \cdot r_2, \langle v_1, v_2\rangle) \mid (r_i, v_i) \in \mathcal{V}[\![A_i]\!]^W_\rho\} \\
\mathcal{V}[\![A \multimap B]\!]^W_\rho & \stackrel{\text{def}}{=} & \left\{(r, v) \;\middle|\; \begin{array}{r} \forall W' \sqsupseteq W.\ (r', v') \in \mathcal{V}[\![A]\!]^{W'}_\rho \\ \Longrightarrow (r \cdot r', v\ v') \in \mathcal{E}[\![B]\!]^{W'}_\rho \end{array}\right\} \\
\mathcal{V}[\![!A]\!]^W_\rho & \stackrel{\text{def}}{=} & \{(r \cdot r', !v) \mid (r, v) \in \mathcal{V}[\![A]\!]^W_\rho,\ r = r \cdot r\} \\
\mathcal{V}[\![\forall X{:}\sigma :: P.\ A]\!]^W_\rho & \stackrel{\text{def}}{=} & \bigcap \left\{\mathcal{V}[\![A]\!]^W_{\rho[X \mapsto d]} \;\middle|\; \rho[X \mapsto d] \models P\right\} \\
\mathcal{V}[\![\exists X{:}\sigma :: P.\ A]\!]^W_\rho & \stackrel{\text{def}}{=} & \bigcup \left\{\mathcal{V}[\![A]\!]^W_{\rho[X \mapsto d]} \;\middle|\; \rho[X \mapsto d] \models P\right\} \\
\mathcal{V}[\![\forall \alpha : \kappa.\ A]\!]^W_\rho & \stackrel{\text{def}}{=} & \bigcap \left\{\mathcal{V}[\![A]\!]^W_{\rho[\alpha \mapsto V]} \;\middle|\; V \in \mathcal{K}[\![\kappa]\!]\right\} \\
\mathcal{V}[\![\exists \alpha : \kappa.\ A]\!]^W_\rho & \stackrel{\text{def}}{=} & \bigcup \left\{\mathcal{V}[\![A]\!]^W_{\rho[\alpha \mapsto V]} \;\middle|\; V \in \mathcal{K}[\![\kappa]\!]\right\} \\
\mathcal{V}[\![\alpha]\!]^W_\rho & \stackrel{\text{def}}{=} & \rho(\alpha) \\
\mathcal{V}[\![\,[A]\,]\!]^W_\rho & \stackrel{\text{def}}{=} & \{(r, \bullet) \mid \exists v.\ (r, v) \in \mathcal{V}[\![A]\!]^W_\rho\} \\
\mathcal{V}[\![\mathsf{if}(t, A, B)]\!]^W_\rho & \stackrel{\text{def}}{=} & \text{if } \mathcal{I}[\![t]\!]_\rho = \mathsf{tt} \text{ then } \mathcal{V}[\![A]\!]^W_\rho \text{ else } \mathcal{V}[\![B]\!]^W_\rho \\
\mathcal{V}[\![\lambda X : \sigma.\ A]\!]^W_\rho & \stackrel{\text{def}}{=} & \lambda d \in \mathcal{S}[\![\sigma]\!].\ \mathcal{V}[\![A]\!]^W_{\rho[X \mapsto d]} \\
\mathcal{V}[\![A\ t]\!]^W_\rho & \stackrel{\text{def}}{=} & (\mathcal{V}[\![A]\!]^W_\rho)(\mathcal{I}[\![t]\!]_\rho)
\end{array}$$

$$\begin{array}{rcl}
\mathcal{E}[\![A]\!]^W_\rho & \stackrel{\text{def}}{=} & \{\ (r, e) \mid \forall j < W.k,\ (s, r \cdot r_{\mathrm{F}}) : W. \\
& & \quad \text{if} \quad h = (s \cdot r \cdot r_{\mathrm{F}})[0],\ \langle h; e\rangle \hookrightarrow_j \langle h'; e'\rangle \not\hookrightarrow \\
& & \quad \text{then}\ \ \exists W' \sqsupseteq_j W,\ (s', r' \cdot r_{\mathrm{F}}) : W' \\
& & \quad \text{with}\ \ h' = (s' \cdot r' \cdot r_{\mathrm{F}})[0],\ (r', e') \in \mathcal{V}[\![A]\!]^{W'}_\rho\ \}
\end{array}$$

Figure 10. Kripke Logical Relation

rowed from Vafeiadis's work on concurrent separation logics [41], in which a closely analogous distinction arises.)

Formally, the relationship between the local and shared resources is codified by the *world satisfaction relation* $(s, r) : W$, defined in Figure 9, which asserts that s can be split into m composite resources $\overline{s_i}$ (one for each island of W) such that s_i satisfies island i's representation invariant $W.\omega[i].I$. Note that the argument passed to I is $(s \cdot r)[i]$: this is correct because I's argument is supposed to represent the global store of the i-th island, which is precisely the i-th projection of the global composite store, $s \cdot r$. Note also that the world parameter of each island's resource predicate is instantiated with $\triangleright W$, the "current" world W approximated one step-index level down.

Kripke Logical Relation Logical relations characterize program behavior by induction over type structure, lifting properties about *base* type computations to properties at *all* types: a term at a compound type is "well-behaved" if every way of eliminating it yields a "well-behaved" term at some simpler type. *Kripke logical relations* index logical relations by a world W, which places constraints on the machine states under which terms are required to behave well. Although logical relations are often *binary relations* for proving program equivalences [30], it suffices in our case to define *unary predicates*, since we are merely trying to prove safety [6].

Figure 10 presents our Kripke logical relation. We assume a semantics of sorts $\mathcal{S}[\![\sigma]\!]$, index terms $\mathcal{I}[\![t]\!]_\rho$ (where $\mathrm{fv}(t) \subseteq \mathrm{dom}(\rho)$) and propositions $\rho \models P$ (where $\mathrm{fv}(P) \subseteq \mathrm{dom}(\rho)$), all standard from multisorted first-order logic. From the semantics of sorts, we can easily build a semantics of kinds $\mathcal{K}[\![\kappa]\!]$. The value predicate $\mathcal{V}[\![A]\!]^W_\rho$ is indexed by both a world W and a semantic environment ρ, and is satisfied by pairs (r, v) of values v and their supporting (composite) resources r. Because the type system is affine, the resource r may contain some part that is irrelevant to v—and in general, if $(r, v) \in \mathcal{V}[\![A]\!]^W_\rho$ and $W' \sqsupseteq W$ then $(r \cdot r', v) \in \mathcal{V}[\![A]\!]^{W'}_\rho$, an assumption codified in the definition of ValPred. This monotonicity property means that the good behavior of a term can depend on certain islands and resources being present, but *not* on certain islands or resources being absent.

$$\begin{array}{rcl}
\mathrm{Env}[\![\cdot]\!] & \stackrel{\text{def}}{=} & \{\emptyset\} \\
\mathrm{Env}[\![\Sigma, \alpha : \kappa]\!] & \stackrel{\text{def}}{=} & \{\rho, \alpha \mapsto V \mid \rho \in \mathrm{Env}[\![\Sigma]\!],\ V \in \mathcal{K}[\![\kappa]\!]\} \\
\mathrm{Env}[\![\Sigma, X : \sigma]\!] & \stackrel{\text{def}}{=} & \{\rho, X \mapsto d \mid \rho \in \mathrm{Env}[\![\Sigma]\!],\ d \in \mathcal{S}[\![\sigma]\!]\} \\
\mathcal{U}[\![\cdot]\!]^W_\rho & \stackrel{\text{def}}{=} & \{\emptyset\} \\
\mathcal{U}[\![\Gamma, x : A]\!]^W_\rho & \stackrel{\text{def}}{=} & \left\{\gamma, x \mapsto (r, v) \;\middle|\; \begin{array}{l} \gamma \in \mathcal{U}[\![\Gamma]\!]^W_\rho, (r, v) \in \mathcal{V}[\![A]\!]^W_\rho, \\ r = r \cdot r \end{array}\right\} \\
\mathcal{L}[\![\cdot]\!]^W_\rho & \stackrel{\text{def}}{=} & \{\emptyset\} \\
\mathcal{L}[\![\Delta, x : A]\!]^W_\rho & \stackrel{\text{def}}{=} & \{\delta, x \mapsto (r, v) \mid \delta \in \mathcal{L}[\![\Delta]\!]^W_\rho,\ (r, v) \in \mathcal{V}[\![A]\!]^W_\rho\} \\
\pi(\gamma) & \stackrel{\text{def}}{=} & \bigodot \{r \mid x \in \mathrm{dom}(\gamma),\ \gamma(x) = (r, v)\} \\
\pi(\delta) & \stackrel{\text{def}}{=} & \bigodot \{r \mid x \in \mathrm{dom}(\delta),\ \delta(x) = (r, v)\}
\end{array}$$

$$\Sigma; \Pi; \Gamma; \Delta \Vdash e : A \stackrel{\text{def}}{=} \forall W,\ \rho \in \mathrm{Env}[\![\Sigma]\!], \gamma \in \mathcal{U}[\![\Gamma]\!]^W_\rho, \delta \in \mathcal{L}[\![\Delta]\!]^W_\rho.\ \rho \models \Pi \Longrightarrow (\pi(\gamma) \cdot \pi(\delta), \delta(\gamma(e))) \in \mathcal{E}[\![A]\!]^W_\rho$$

Figure 11. Semantics of Open Terms

The definition of the value predicate is essentially standard—in particular, it is essentially an affine version of the model of $\mathbf{L}^3$ [5] outfitted with our monoidal worlds. One difference is our interpretation of the exponential $!A$, which is inhabited by $!v$ only when v can be supported by some *idempotent* portion of the resources—that is, some part of the resources that permits the structural rule of contraction. (In $\mathbf{L}^3$, the heap is the only resource, so only the empty heap is idempotent.) Also, since universal and existential types are introduced implicitly, they are given intersection and union semantics, respectively. The remaining differences are to do with indexed types—*e.g.*, the parameter indexing the base types bool, nat, and ptr must reflect the particular value inhabiting the type—and the computational irrelevance type $[A]$, whose interpretation records the resources needed to justify A but not the value that inhabits it.

The term predicate $\mathcal{E}[\![A]\!]^W_\rho$ captures the crucial property supporting sharing: namely, that computations are frame-respecting. Suppose that a term e owns (composite) resource r. To show e is well-behaved, we quantify over an arbitrary frame resource r_{F} representing the resource of e's evaluation context. Together, $r \cdot r_{\mathrm{F}}$ constitute the *local resource*, *i.e.,* the portion of the global composite store that the program being executed owns. We also quantify over some *shared resource* s such that $(s, r \cdot r_{\mathrm{F}}) : W$. If e reduces to an irreducible term e'—starting from the global heap that is the 0-th projection of $s \cdot r \cdot r_{\mathrm{F}}$—in j steps, where j is less than the world's step-index $W.k$, then it must (1) leave the heap in a state described by a new global composite store $s' \cdot r' \cdot r_{\mathrm{F}}$, such that (2) $(s', r' \cdot r_{\mathrm{F}}) : W'$ for some future world W' of W (whose step-index is $W.k - j$), and (3) the final term e' is in fact a value that, supported by the resource r', obeys the value predicate $\mathcal{V}[\![A]\!]^{W'}_\rho$. Note, however, that the frame resource r_{F} must remain unchanged.

The logical predicates defined in Figure 10 only describe well-behaved *closed* terms. In Figure 11, we lift these to predicates on *open* terms in the standard way: namely, we consider e to be well-behaved at the type A under context Ω, written $\Omega \Vdash e : A$, if it is well-behaved (according to $\mathcal{E}[\![A]\!]$) for all well-behaved closing instantiations of its free variables. These closing instantiations include both values and the resources supporting them; the π operator then multiplies together all the resources supporting a closing instantiation. Note that the resources accompanying the instantiations of the unrestricted variables in Γ are required to be idempotent, so that they may be safely duplicated within the proof of soundness.

Soundness of the Type System The main technical result of the paper is summed up in the following theorems:

Theorem 1 (Fundamental Theorem of Logical Relations)**.**
If $\Omega \vdash e : A$*, then* $\Omega \Vdash e : A$*.*

Theorem 2 (Adequacy)**.**
If $\emptyset \Vdash e : A$ *and* $\langle \emptyset; e\rangle \hookrightarrow_* \langle h; e'\rangle \not\hookrightarrow$*, then* e' *is a value.*

Corollary 3 (Soundness of the Type System)**.**
If $\emptyset \vdash e : A$ *and* $\langle \emptyset; e\rangle \hookrightarrow_* \langle h; e'\rangle \not\hookrightarrow$*, then* e' *is a value.*

The proof of Adequacy is almost trivial. The proof of the Fundamental Theorem essentially proceeds by showing that each rule in our type system is *semantically* sound, *i.e.,* that it holds if all the syntactic $\vdash$'s are replaced by semantic $\Vdash$'s. The proofs for most rules follow previous developments using SKLRs [3, 17, 16]. The most interesting new case, of course, is that of the sharing rule. The proof is quite involved, so here we will just offer a rough idea of how the proof goes, focusing on the most interesting technical constructions. (For the full details, see the technical appendix [24].)

As described above, the intuition behind our worlds W is that each island in W corresponds to an application of the sharing rule. Indeed, the proof that the sharing rule is semantically sound is the only part of our proof that involves extending a given input world W with a new island to form a future world W' (as permitted in the definition of the logical term predicate). Supposing W already had n islands $(0..n-1)$, the new island will have index n.

At first glance, it would seem we want to define this new island to be $(\mathcal{S}[\![\sigma]\!], \cdot, \epsilon, I_{\mathsf{simple}})$, where $(\mathcal{S}[\![\sigma]\!], \cdot, \epsilon)$ is the monoid with which the sharing rule was instantiated, and the representation invariant I_{simple} is defined in terms of the A in the first premise of the rule (and whatever ρ we are given to interpret its free variables):

$$I_{\mathsf{simple}}(x) = \{(W, r) \mid \exists v.\ (r, v) \in \mathcal{V}[\![A]\!]^W_\rho(x)\}$$

This invariant stipulates that the shared resource of island n satisfies the capability $[A\ x]$ when the island's global store is x.

However, we must also take account of the lock ℓ that the dynamic semantics of share creates in order to protect against reentrancy. Intuitively, when the lock ℓ is released, the representation invariant of island n should be much like the above I_{simple}. But when the lock ℓ is held, it means we are in the middle of a call to one of the operations returned by share, during which the representation invariant might not hold at all. The monoid of island n must therefore reflect these two possibilities.

We define island n as $(M, +, U(\epsilon), I)$, where (in ML notation)

$$\mathsf{type}\ M = \ U\ \mathsf{of}\ \mathcal{S}[\![\sigma]\!]\ \mid\ L\ \mathsf{of}\ \mathcal{S}[\![\sigma]\!] \times \mathcal{S}[\![\sigma]\!]\ \mid\ \bot,$$

the composition operator $(+)$ is the commutative closure of

$$\begin{array}{rcl@{\qquad}rcl} U(x) + U(y) & = & U(x \cdot y) & L(\_) + L(\_) & = & \bot \\ L(x, y) + U(z) & = & L(x, y \cdot z) & \bot + \_ & = & \bot, \end{array}$$

and the representation invariant I is defined as

$$\begin{array}{rcl} I(U(x)) & = & \{(W, r \cdot [\ell : \mathtt{ff}]) \mid \exists v.\ (r, v) \in \mathcal{V}[\![A]\!]^W_\rho(x)\} \\ I(L(x, y)) & = & \{(W, [\ell : \mathtt{tt}]) \mid x = y\} \\ I(\bot) & = & \emptyset. \end{array}$$

The idea here is to distinguish between *unlocked* states $U(x)$, where the lock ℓ is released, and *locked* states $L(x, y)$, where ℓ is held. In the former case, I asserts that ℓ points to ff and that the rest of the island's shared resource r can satisfy $[A\ x]$, as required for invoking any of the shared operations.[4] In the latter case, I asserts that ℓ points to tt and that $x = y$ (we explain about that in a moment). Finally, we give the following interpretation for the abstract capability constructor α (returned by the share operation):

$$[\![\alpha]\!] = \lambda x \in \mathcal{S}[\![\sigma]\!].\ \{(W, (r, \bullet)) \mid r[n] = r' + U(x)\}$$

This essentially says that the owner of $[\alpha\ x]$ has control over a $U(x)$ piece of the resource on island n.

The two parameters to L are a technical trick we use to show that the shared operations of the ADT are "frame-preserving". Specifically, the monoid we have defined has the property that if we control $L(y, \epsilon)$ of the resource, then the only possible resource r that the rest of the program could have on island n, such that $I(L(y, \epsilon) + r)$ is satisfiable, is $U(y)$. To see how this is exploited in the soundness proof, suppose that a client owns $[\alpha\ t]$ (*i.e.,* she controls a $U(t)$ piece of island n's resource), and invokes one of the shared operations, whose type spec (see Figure 7) promises to transform $[\alpha\ t]$ into $[\alpha\ t']$ for some t'. (For simplicity, we'll ignore the frame X in the type of the operation. It does not add any fundamental complication.) If the lock is held, the operation will diverge and there is nothing to show. If the lock is released, the definition of I guarantees that the rest of the global store on island n must be of the form $U(y)$ for some y, and that the island's shared resource r satisfies $[A\ (t \cdot y)]$. Here, $U(y)$ represents the control the rest of the program has over the shared state of island n, and we must show that the operation we are about to execute respects it.

Now, before invoking the underlying operation, we acquire the lock, we remove r from the shared resource so that we can transfer ownership of it to the operation, and—this is the key point—we replace the client's local $U(t)$ resource with the resource $L(y, \epsilon)$, thus updating the global store of island n to $L(y, y)$. When we invoke the underlying operation, we place the $L(y, \epsilon)$ in its *frame*, which (by definition of the logical term predicate) it must preserve. Thus, when we get back control from the operation (which must be in a state such that I is satisfiable), the global store of island n must still be $L(y, y)$, of which the client controls $L(y, \epsilon)$ and the rest of the program controls $U(y)$. Also, the frame-preserving nature of the underlying operation's type tells us that it must have returned us a resource r' satisfying the capability $[A\ (t' \cdot y)]$. We can then release the lock, replace the client's $L(y, \epsilon)$ resource with $U(t')$ (which is what the client expects to control when the operation is completed), and transfer ownership of r' back to the island's shared resource, which now satisfies I at the new global store, $U(t' \cdot y)$. But crucially, despite/because of all these shenanigans, the resource $U(y)$ belonging to the rest of the program has been left untouched!

5. Related Work

Dealing with Reentrancy: Locking vs. the Anti-Frame Rule As explained in Section 3, our sharing rule uses a lock to protect against unsafe reentrancy, which can arise in our language due its support for *shared, higher-order* state. Most prior separation logics have not had to deal with such a hard problem because they are done in a first-order setting, where the possibility of reentrancy is syntactically evident; and most prior substructural type systems (*e.g.,* $\mathbf{L}^3$ [5]) have not had to deal with it because they don't support sharing/hiding of state.

One exception is Pottier's work on the *anti-frame rule* [32], which *does* account for reentrancy in the presence of shared, higher-order state. The anti-frame rule permits a group of functions to operate on a piece of hidden state described by an invariant C. Externally to the anti-frame rule, those functions may have type $!(A \multimap B)$, but internally they have roughly the form $!(A \otimes C \multimap B \otimes C)$ (but not quite, as we explain below). In a substructural setting, the rule therefore gives a way to export, *e.g.,* an affine reference with a set of operations, without treating the operations themselves as affine or forcing the client to thread the the affine reference capability through its code. The restriction to a

[4] Note that if the sharing rule did not require A to represent a *capability* (*i.e.,* to appear in proof-irrelevant brackets), then invoking any of the shared operations would require us to cough up the actual value v witnessing $A\ x$ (whereas here, v is $\exists$-quantified). This could be achieved by changing the implementation of the sharing rule so that it maintains a private reference cell ℓ storing the current witness v, and then updating I to also own $[\ell : v]$.

simple invariant has been subsequently relaxed to support hidden monotonic invariants [35], as well as monotonic "observations" about hidden state [29] (although to our knowledge the last extension has not yet been proven sound).

Pottier's approach provides a more general solution to the reentrancy problem (of which our use of locks would constitute one mode of use), but this comes at the cost of significant additional complexity in the typing rule for hiding (*i.e.,* the anti-frame rule) itself. In particular, the $\otimes$ operator that Pottier employs in the type $!(A \otimes C \multimap B \otimes C)$ above is not a simple tensor, but rather a tensoring operation, which propagates under $\to$ and ref types and comes equipped with a non-standard equational theory. Soundness proofs of the anti-frame rule using traditional syntactic techniques have consequently required years of heroic effort [33]. That said, significantly simpler semantic proofs of the anti-frame rule have also been given using Kripke logical relations [35]. Based on this experience, we chose to use a semantic model in our work, and have been very satisfied with its simplicity.

In this paper, we decided to isolate concerns by focusing on sharing and leaving an improved handling of reentrancy to future work. One possibility would be to consider synthesizing our sharing rule with the anti-frame rule, since they are complementary. The anti-frame rule offers a more general treatment of reentrancy, while the sharing rule offers a more general treatment of sharing. As demonstrated in our weak references example, simple invariants may be encoded via the sharing rule using the unit monoid, and subsequently hidden. More novel, however, is our support for a variety of interesting uses of sharing involving both monotonic state and *non-monotonic* state (*e.g.,* the memory manager example). Furthermore, our use of monoids lets clients divide, transfer, and recombine resources as they need, without restricting to a one-way increase in information as the anti-frame rule does.

Fictions of Separation From the outset, substructural reasoning about state has relied on the notion of disjointly supported assertions for local reasoning, but only gradually has the flexibility of that notion become clear. Early models of logically (but not physically) separable resources like fractional permissions [7, 10] and trees [9] treat those resources as primitive, either baking them into the operational semantics or, in simple cases, relying on a fixed interpretation into an underlying heap. To handle higher-level notions of separation, Krishnaswami *et al.* [23] embedded "domain-specific separation logics" into higher-order separation logic, and Dinsdale-Young, Gardner, and Wheelhouse named the general phenomenon "fictional disjointness" and justified its support of local reasoning by employing data refinement and axiomatic semantics [14].

Contemporaneously, *concurrent abstract predicates* (CAP, [13]) combined fictional disjointness with several other important ideas—the two most relevant being abstract predicates [28] and rights-as-resources [15]. CAP allows the specification of each module to include abstract predicates which, like the abstract data types introduced by our sharing rule, represent local knowledge and rights about a shared underlying resource. Hence, just as the tensor $\otimes$ is the all-purpose notion of separation for us, so separating conjunction $*$ is for CAP. On the other hand, CAP is built on more specific and complex forms of knowledge and rights, inherited from deny-guarantee [15] and intended for reasoning about concurrency.

In very recent work, several groups of researchers have simultaneously proposed variants of commutative monoids as an abstract way to capture fictional separation. Their original goals were quite distinct: Jensen and Birkedal's fictional separation logic (FSL) [21] is explicitly intended as a simple axiomatization of fictional disjointness within separation logic; Dinsdale-Young *et al.*'s views [12] are intended as a more abstract account of CAP (and compositional reasoning about concurrency in general); and Ley-Wild and Nanevski's subjective concurrent separation logic (SCSL) [25] is geared toward compositional reasoning about ghost state.

The three frameworks also share a shortcoming: the separating conjunction $*$ of the assertion language is tied to a single, specific monoid. With views and SCSL, this monoid is fixed at the outset, when the framework is instantiated. FSL, in contrast, is based on *indirect Hoare triples* parameterized by an *interpretation map*, which explicitly records a monoid together with its interpretation as a predicate on underlying resources. An interpretation map is akin to an island in our model (Section 4), which means that the assertions within an indirect Hoare triple must all be given in terms of a single abstract resource. While FSL enables interpretation maps to be stacked in layers or combined as a product (resembling our worlds), such structure must be explicitly managed within both assertions and proofs.

Our sharing rule also employs commutative monoids for fictional separation, but it associates a *different* monoid with each abstract data type it introduces. Consequently, our tensor product constructor $\otimes$ implicitly mediates between all resources "currently" in existence, both the physical resources and a dynamically-growing set of user-defined logical resources.

Temporarily Structural Types Most substructural type systems are not *completely* substructural: they permit, by a variety of means, linear or affine types to coexist with unrestricted types. Keeping a strict distinction between the two kinds of types is crucial for ensuring the soundness of *e.g.* strong updates, but it is also impractical for large programs with complex data structures. There have been numerous proposals for safely allowing the rules to be bent [37, 36], a well known example being Fähndrich and DeLine's *adoption and focus* [19]. At the root of these designs for "temporarily structural types" is the ability to *revoke* access to previously aliased data, providing a freshly linear view of that data. When unrestricted access is later restored, however, there must be some way of ensuring that the aliases still have an appropriate type, and the simplest way of doing that is to keep the type fixed.

Our sharing rule, on the other hand, does not commit to a particular aliasing discipline. The abstract resources supported by a shared underlying resource can be created and aliased to whatever extent their governing monoid allows, and can be strongly updated at any time without risk of invalidating non-local assertions. It remains to be seen whether our monoidal approach is flexible enough to recover the sophisticated rule-bending of the "temporarily structural" typing disciplines mentioned above.

Per-Module Notions of Resources Two recent languages—Tov's *Alms* [40] and Mazurak and Zdancewic's F° [26]—have been proposed for general-purpose, practical programming with substructural types. The generality of these languages stems from their ability to perform *substructural sealing*: they can seal an unrestricted value with an abstract type at a substructural kind, thereby preventing clients from freely aliasing the value. Substructural sealing, like our sharing construct, provides a way to introduce per-module notions of resource. But substructural sealing is used to impose a *more* restrictive interface on a *less* restrictive value, while sharing goes the other way around, allowing aliasing of affine resources. This difference is apparent in the work done by a typechecker in both cases: for substructural sealing, there is little to check, because it is always safe to tighten the interface to a value; for sharing, the exported operations must be shown to respect their frame. Ultimately, these two forms of resource introduction seem complementary, and indeed, the language we have presented supports both.

Kripke Logical Relations Kripke logical relations have long been used to reason about state in higher-order, ML-like languages [31].

Ahmed *et al.* [5, 4] have given Kripke logical relations for linear languages with state, using a simple notion of possible world

corresponding to strict heap separation. The structure of our logical relation is quite similar to this earlier work, but the structure of our worlds is significantly different, since we must account for interaction between an unbounded number of abstract resource types, each of which is governed by a distinct monoid.

More recently, Ahmed *et al.* [3] and Dreyer *et al.* [16] have given models for higher-order *structural* state based on the concept of *transition systems*, which facilitate the modeling of protocol-based uses of state, as well as the "well-bracketed" state changes possible in languages without control. Since transition systems can be modeled as monoids, our current model fully supports transition systems as a mode of use. With a small extension (whose proof is in the appendix [24]), we can also model Dreyer *et al.*'s "public" vs. "private" transitions for reasoning about well-bracketed state changes, although proofs based on their techniques are arguably more direct than ours. (We plan to report on this in future work.)

6. Conclusion and Future Work

In this paper, we have shown how to put programmer-defined resource abstractions on the same footing as built-in resources such as the heap, yielding a type system that permits the flexible use of aliased data while retaining the simple intuitions of substructural logic. To do so, we combined exciting new ideas from separation logic with classical type-theoretic techniques such as refinement types and data abstraction.

An immediate direction for future work is to study how to optimize the sharing rule, both via the model (*i.e.,* proving that locks are not needed for specific implementations), and via type-theoretic extensions that we could use to avoid locking (*e.g.,* via formalizing the concept of "first-order data" as a modality, or via a sharing modality [36]). Another natural direction for future work is to examine if our methods extend to full-blown value-dependent types (*e.g.,* as in HTT [27]). This poses interesting questions, since methods based on step-indexing have historically had challenges dealing with semantic equalities (as opposed to approximation), and our sharing rule deeply connects existential types and state.

References

[1] A. Ahmed. *Semantics of Types for Mutable State*. PhD thesis, Princeton University, 2004.

[2] A. Ahmed. Step-indexed syntactic logical relations for recursive and quantified types. In *ESOP*, 2006.

[3] A. Ahmed, D. Dreyer, and A. Rossberg. State-dependent representation independence. In *POPL*, 2009.

[4] A. Ahmed, M. Fluet, and G. Morrisett. A step-indexed model of substructural state. In *ICFP*, 2005.

[5] A. Ahmed, M. Fluet, and G. Morrisett. $\mathbf{L}^3$: A linear language with locations. *Fundamenta Informaticae*, 77:397–449, 2007.

[6] A. Appel, P.-A. Melliès, C. Richards, and J. Vouillon. A very modal model of a modern, major, general type system. In *POPL*, 2007.

[7] J. Boyland. Checking interference with fractional permissions. In *SAS*, 2003.

[8] T. Brus, M. C. J. D. van Eekelen, M. van Leer, M. J. Plasmeijer, and H. P. Barendregt. Clean: A language for functional graph rewriting. In *FPCA*, 1987.

[9] C. Calcagno, P. Gardner, and U. Zarfaty. Context logic and tree update. In *POPL*, 2005.

[10] C. Calcagno, P. W. O'Hearn, and H. Yang. Local action and abstract separation logic. In *LICS*, 2007.

[11] R. DeLine and M. Fähndrich. Enforcing high-level protocols in low-level software. In *PLDI*, 2001.

[12] T. Dinsdale-Young, L. Birkedal, P. Gardner, M. Parkinson, and H. Yang. Views: Compositional reasoning for concurrency, 2012. Submitted for publication.

[13] T. Dinsdale-Young, M. Dodds, P. Gardner, M. Parkinson, and V. Vafeiadis. Concurrent abstract predicates. In *ECOOP*, 2010.

[14] T. Dinsdale-Young, P. Gardner, and M. Wheelhouse. Abstraction and refinement for local reasoning. In *VSTTE*, 2010.

[15] M. Dodds, X. Feng, M. J. Parkinson, and V. Vafeiadis. Deny-guarantee reasoning. In *ESOP*, 2009.

[16] D. Dreyer, G. Neis, and L. Birkedal. The impact of higher-order state and control effects on local relational reasoning. In *ICFP*, 2010.

[17] D. Dreyer, G. Neis, A. Rossberg, and L. Birkedal. A relational modal logic for higher-order stateful ADTs. In *POPL*, 2010.

[18] J. Dunfield. *A Unified System of Type Refinements*. PhD thesis, Carnegie Mellon University, 2007.

[19] M. Fähndrich and R. DeLine. Adoption and focus: Practical linear types for imperative programming. In *PLDI*, 2002.

[20] J.-Y. Girard. Linear logic. *TCS*, 50(1):1–102, 1987.

[21] J. Jensen and L. Birkedal. Fictional separation logic. In *ESOP*, 2012.

[22] C. B. Jones. The role of auxiliary variables in the formal development of concurrent programs. In *Reflections on the work of C.A.R. Hoare*, pages 167–188. Springer, 2010.

[23] N. R. Krishnaswami, L. Birkedal, and J. Aldrich. Verifying event-driven programs using ramified frame properties. In *TLDI*, 2010.

[24] N. R. Krishnaswami, A. Turon, D. Dreyer, and D. Garg. Superficially substructural types (Technical appendix), 2012. URL: `http://www.mpi-sws.org/~dreyer/papers/supsub/`.

[25] R. Ley-Wild and A. Nanevski. Subjective concurrent separation logic, 2012. Submitted for publication.

[26] K. Mazurak, J. Zhao, and S. Zdancewic. Lightweight linear types in System F°. In *TLDI*, 2010.

[27] A. Nanevski, G. Morrisett, and L. Birkedal. Hoare Type Theory, polymorphism and separation. *JFP*, 18(5&6):865–911, Sept. 2008.

[28] M. J. Parkinson and G. M. Bierman. Separation logic and abstraction. In *POPL*, 2005.

[29] A. Pilkiewicz and F. Pottier. The essence of monotonic state. In *TLDI*, 2011.

[30] A. Pitts. Typed operational reasoning. In B. C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 7. MIT Press, 2005.

[31] A. Pitts and I. Stark. Operational reasoning for functions with local state. In *HOOTS*, 1998.

[32] F. Pottier. Hiding local state in direct style: a higher-order anti-frame rule. In *LICS*, 2008.

[33] F. Pottier. Syntactic soundness proof of a type-and-capability system with hidden state, 2011. Submitted for publication.

[34] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, 2002.

[35] J. Schwinghammer, L. Birkedal, F. Pottier, B. Reus, K. Støvring, and H. Yang. A step-indexed Kripke model of hidden state. *Mathematical Structures in Computer Science*, 2012. To appear.

[36] R. Shi, D. Zhu, , and H. Xi. A modality for safe resource sharing and code reentrancy. In *ICTAC*, 2010.

[37] F. Smith, D. Walker, and G. Morrisett. Alias types. In *ESOP*, 2000.

[38] R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering*, 12(1):157–171, 1986.

[39] M. Tofte and J.-P. Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997.

[40] J. Tov. *Practical Programming with Substructural Types*. PhD thesis, Northeastern University, 2012.

[41] V. Vafeiadis. *Modular fine-grained concurrency verification*. PhD thesis, University of Cambridge, 2008.

[42] D. Walker, K. Crary, and G. Morrisett. Typed memory management via static capabilities. *TOPLAS*, 22:701–771, 2000.

[43] J. Wickerson, M. Dodds, and M. Parkinson. Explicit stabilisation for modular rely-guarantee reasoning. In *ESOP*, 2010.

[44] N. Wolverson. *Game semantics for an object-oriented language*. PhD thesis, University of Edinburgh, 2008.

[45] H. Xi and F. Pfenning. Dependent types in practical programming. In *POPL*, 1999.