# A Theory of Indirection via Approximation

Aquinas Hobor *†
National University of Singapore
hobor@comp.nus.edu.sg

Robert Dockins†
Princeton University
rdockins@cs.princeton.edu

Andrew W. Appel †
Princeton University
appel@princeton.edu

## Abstract

Building semantic models that account for various kinds of indirect reference has traditionally been a difficult problem. Indirect reference can appear in many guises, such as heap pointers, higher-order functions, object references, and shared-memory mutexes.

We give a general method to construct models containing indirect reference by presenting a "theory of indirection". Our method can be applied in a wide variety of settings and uses only simple, elementary mathematics. In addition to various forms of indirect reference, the resulting models support powerful features such as impredicative quantification and equirecursion; moreover they are compatible with the kind of powerful substructural accounting required to model (higher-order) separation logic. In contrast to previous work, our model is easy to apply to new settings and has a simple axiomatization, which is complete in the sense that all models of it are isomorphic. Our proofs are machine-checked in Coq.

***Categories and Subject Descriptors*** D.3.1 [*PROGRAMMING LANGUAGES*]: Formal Definitions and Theory — Semantics; F.3.1 [*LOGICS AND MEANINGS OF PROGRAMS*]: Specifying and Verifying and Reasoning about Programs — Logics of programs; F.4.1 [*MATHEMATICAL LOGIC AND FORMAL LANGUAGES*]: Mathematical Logic — Mechanical theorem proving

***General Terms*** Languages, Theory, Verification

***Keywords*** Indirection theory, Step-indexed models

## 1. Introduction

A recurring problem in the semantics of programming languages is finding semantic models for systems with indirect reference. Indirection via pointers gives us mutable records; indirection via locks can be used for shared storage; indirection via code pointers gives us complex patterns of computation and recursion. Models for program logics for these systems need to associate invariants (or assertions, or types) with addresses in the store; and yet invariants are predicates on the store. Tying this "knot" has been difficult, especially for program logics that support abstraction (impredicativity).

* Supported by a Lee Kuan Yew Postdoctoral Fellowship.

† Supported in part by NSF awards CNS-0627650 and CNS-0910448, and AFOSR award FA9550-09-1-0138.



*POPL'10,* January 17–23, 2010, Madrid, Spain.


Consider general references in the polymorphic $\lambda$-calculus. Here is a flawed semantic model of types for this calculus:

$$\begin{array}{rcl} \mathsf{value} & \equiv & \mathsf{loc\ of\ address} \ + \ \mathsf{num\ of}\ \mathbb{N} \ + \ \ldots \\ \mathsf{type} & \equiv & (\mathsf{memtype} \times \mathsf{value}) \to \mathbb{T} \\ \mathsf{memtype} & \approx & \mathsf{address} \rightharpoonup \mathsf{type} \end{array} \quad (1)$$

Values are a tagged disjoint union, with the tag loc indicating a memory address. $\mathbb{T}$ is some notion of truth values (*e.g.*, the propositions of the metalogic or a boolean algebra) and a natural interpretation of the type $A \to \mathbb{T}$ is the characteristic function for a set of $A$. We write $\approx$ to mean "we wish we could define things this way" and $A \rightharpoonup B$ to indicate a finite partial function from $A$ to $B$.

The typing judgment has the form $\psi \vdash v : \tau$, where $\psi$ is a memory typing (memtype), $v$ is a value, and $\tau$ is a type; the semantic model for the typing judgment is $\psi \vdash v : \tau \equiv \tau(\psi, v)$. Memory typings are partial functions from addresses to types. The motivation for this attempt is that the type "ref $\tau$" can use the memory typing to show that the reference's location has type $\tau$:

$$\mathsf{ref}\ \tau \quad \equiv \quad \lambda(\psi, v).\ \exists a.\ v = \mathsf{loc}(a) \ \wedge \ \psi(a) = \tau.$$

That is, a value $v$ has type ref $\tau$ if it is an address $a$, and according to the memory typing $\psi$, the memory cell at location $a$ has type $\tau$.

Unfortunately, this series of definitions is not well-founded: type contains a contravariant occurrence of memtype, which in turn contains an occurrence of type. A standard diagonalization proves that no solution to these equations exists in set theory.

A recent approach to tackle this problem is to employ *stratification*, yielding a well-founded definition of an appropriate semantic model [AAV03, Ahm04, AMRV07]. A key strength of these models is that they can express general (impredicative) quantified types even though they are stratified. Unfortunately, these models have a disturbing tendency to "leak" into any proofs utilizing them, making modularization difficult, obscuring their essential features, needlessly limiting their power, and contributing to unpleasant tedium. Also, all of these models are specialized to the problem of mutable references, and even for the expert it can be daunting to modify the techniques to apply to other domains.

Hobor *et al.* applied these techniques to Hoare logics, developing a model for a concurrent separation logic (CSL) with first-class locks and threads [HAZ08]. This model was extremely complex to construct because the substructural accounting was woven throughout the stratification. It was also complex to use, exposing more than fifty axioms.Even then, the axiom set was incomplete: from time to time the CSL soundness proof required the statement and proof of additional properties (which were provable from the model). Like the mutable reference models, Hobor *et al.*'s model was a solution to a specific problem, and could not be applied to other problems—not even to other concurrent separation logics.

We present a single solution, *indirection theory*, that can handle these domains as well as numerous others. Our model is characterized by just two axioms, an order of magnitude better than the simplest of the previous solutions. The axioms are equational, orthog-

onal, and complete, and have greater expressive power and cleaner modularity. The modularity enables a graceful extension for substructural accounting that does not need to thread the accounting through the stratification. Moreover, the axioms provide greater insight into the model by directly exposing the approximation at the heart of the stratification technique.

As we show in §2, a key observation is that many domains can be described by the pseudoequation:

$$K \quad \approx \quad F((K \times O) \to \mathbb{T}), \tag{2}$$

where $F(X)$ is a covariant functor, $O$ is some arbitrary set of "other" data and $K$ is the object that we wish to construct. The earlier cardinality argument guarantees we cannot construct $K$ (in set theory), but indirection theory lets us approximate it:

$$K \quad \preceq \quad \mathbb{N} \times F((K \times O) \to \mathbb{T}) \tag{3}$$

Here $X \preceq Y$ means the "small" type $X$ is related to the "big" type $Y$ by two functions: $\mathsf{squash} : Y \to X$ and $\mathsf{unsquash} : X \to Y$. The squash function "packs" an element $y$ of the big type $Y$ into an element of the small type $X$, applying an approximation when the structure of $y$ is too complex to fit into $X$. The unsquash function reverses the process, "unpacking" an element of $X$ into an element of $Y$; since $Y$ is bigger than $X$, unsquash is lossless.

The squash and unsquash functions form a section-retraction pair, meaning that $\mathsf{squash} \circ \mathsf{unsquash} : X \to X$ is the identity function and $\mathsf{unsquash} \circ \mathsf{squash} : Y \to Y$ is an approximation function. Thus $X \preceq Y$ is *almost* an isomorphism, and informally one can read $X \preceq Y$ as "X is approximately Y". With equation (3), indirection theory says that left hand side of pseudoequation (2) is approximately equal to a pair of a natural number and the right hand side of pseudoequation (2).

***Contributions.***

§2 We show numerous examples containing indirect reference and show how they can be characterized with a covariant functor.

§3 We present the two axioms of indirection theory.

§4 We show how to apply the axioms to two of the examples.

§5 We explore some implications of indirection theory.

§6 We induce an impredicative logic from indirection theory.

§7 We combine indirection theory and substructural accounting.

§8 We show a simple way to construct the model once and for all, avoiding the need for application-specific models.

§9 We prove that the axioms of indirection theory completely characterize our model (that is, all models are isomorphic).

Our proofs are completely machine-checked. The Coq implementation of the axioms of indirection theory from §3, the logics and constructions from §6 and §7, the model construction from §8, the uniqueness proof from §9, and the worked example of using indirection theory to prove the type soundness of the polymorphic $\lambda$-calculus with references from §2.1, are available at:

http://msl.cs.princeton.edu/

## 2. Applications for indirection theory

Here we examine a selection of examples involving indirect reference. In each case intuition leads to an impossible model that we can approximate with indirection theory. We do not explain the examples in great detail, and refer interested readers to the original papers, which explain the full motivation and original models. In each case, constructing the original model was a difficult task.

### 2.1 General references in the $\lambda$-calculus

Ahmed *et al.* constructed the first model of a type system for the polymorphic $\lambda$-calculus with general references [AAV03]. Following (1), we want a solution to the pseudoequation

$$\mathsf{memtype} \quad \approx \quad \mathsf{address} \rightharpoonup ((\mathsf{memtype} \times \mathsf{value}) \to \mathbb{T}),$$

which falls neatly into the pattern of pseudoequation (2) with

$$\begin{aligned} F(X) &\equiv \mathsf{address} \rightharpoonup X \\ O &\equiv \mathsf{value}. \end{aligned}$$

By equation (3), indirection theory constructs the approximation

$$\mathsf{memtype} \quad \preceq \quad \mathbb{N} \times (\mathsf{address} \rightharpoonup ((\mathsf{memtype} \times \mathsf{value}) \to \mathbb{T})),$$

and by folding the definition of type (from eqn. 1) we reach

$$\mathsf{memtype} \quad \preceq \quad \mathbb{N} \times (\mathsf{address} \rightharpoonup \mathsf{type}).$$

This model is sufficient to define a powerful type system for the polymorphic $\lambda$-calculus with mutable references. In §4.1, we will show how to construct the types nat and ref $\tau$.

### 2.2 General references in von Neumann machines

Modeling a type system with general references for von Neumann machines was solved first by Ahmed [Ahm04], and then later in a more sophisticated way by Appel *et al.* [AMRV07]. The key is that whereas in the $\lambda$-calculus types are based on sets of values, on a von Neumann machine types are based on sets of register banks. Here is the intuition for a von Neumann machine with 32-bit integer memory addressing and $m$ 32-bit integer registers:

$$\begin{aligned} \mathsf{rbank} &\equiv \mathsf{int}_{32} \times \overset{m-2}{\cdots} \times \mathsf{int}_{32} \\ \mathsf{type} &\equiv (\mathsf{memtype} \times \mathsf{rbank}) \to \mathbb{T} \\ \mathsf{memtype} &\approx \mathsf{int}_{32} \rightharpoonup \mathsf{type}. \end{aligned}$$

The intuition is thus very similar to the $\lambda$-calculus case. We set

$$\begin{aligned} F(X) &\equiv \mathsf{int}_{32} \rightharpoonup X \\ O &\equiv \mathsf{rbank}, \end{aligned}$$

and indirection theory constructs the approximation

$$\mathsf{memtype} \quad \preceq \quad \mathbb{N} \times (\mathsf{int}_{32} \rightharpoonup \mathsf{type}).$$

In the $\lambda$-calculus, the memory maps addresses to values, and values are the $O$ used in constructing the model for types. In the von Neumann machine, the memory maps 32-bit integer addresses to 32-bit integer values; however, $O$ is not a single 32-bit integer but instead a register bank of $m$ 32-bit integers. This minor mismatch makes the type ref $\tau$ slightly harder to define in the von Neumann case, but does not cause any fundamental difficulties.

### 2.3 Object references

Hriţcu and Schwinghammer modeled general references in the setting of Abadi and Cardelli's object calculus [HS08]. The object calculus setting introduces a number of new issues: object creation, method updates, and bounded subtyping. To model the storage of methods in the heap, Hriţcu and Schwinghammer would like to build the following impossible model:

$$\begin{aligned} \mathsf{type} &\equiv (\mathsf{heaptype} \times \mathsf{value}) \to \mathbb{T} \\ \mathsf{heaptype} &\approx \mathsf{address} \rightharpoonup \mathsf{type}. \end{aligned}$$

Again applying our recipe, we choose $F$ and $O$ as

$$\begin{aligned} F(X) &\equiv \mathsf{address} \rightharpoonup X \\ O &\equiv \mathsf{value}, \end{aligned}$$

which yields the approximate model

$$\mathsf{heaptype} \quad \preceq \quad \mathbb{N} \times (\mathsf{address} \rightharpoonup \mathsf{type}).$$

Pleasingly, the normal complications of objects do not appear in the construction of the semantic model. Objects, subtyping and quantification are all dealt with on top of the same simple model of memory/heap typings used for general references.

### 2.4 Substructural state

Ahmed *et al.* used substructural state to model the uniqueness types found in languages such as Clean, Cyclone, and TAL [AFM05]. Here the intuitive model has two changes:

$$\begin{array}{lcl} \mathsf{quals} & \equiv & \{U, R, A, L\} \\ \mathsf{type} & \equiv & (\mathsf{memtype} \times \mathsf{quals} \times \mathsf{value}) \to \mathbb{T} \\ \mathsf{memtype} & \approx & \mathsf{address} \rightharpoonup \mathsf{quals} \times \mathsf{type}. \end{array}$$

The quals indicate substructural restrictions: $U$ indicates unrestricted data, $R$ indicates relevant, $A$ indicates affine, and $L$ indicates linear. This model differs from the previous two by putting quals in both contravariant and covariant positions. We can set

$$\begin{array}{lcl} F(X) & \equiv & \mathsf{address} \rightharpoonup (\mathsf{quals} \times X) \\ O & \equiv & \mathsf{quals} \times \mathsf{value}, \end{array}$$

and use indirection theory to construct the approximation

$$\mathsf{memtype} \preceq \mathbb{N} \times (\mathsf{address} \rightharpoonup (\mathsf{quals} \times \mathsf{type})).$$

### 2.5 Embedding semantic assertions in program syntax

Consider a Hoare logic for a language with function pointers and an **assert** statement. The predicate $f : \{P\}\{Q\}$ means that $f$ is the address of a function with precondition $P$ and postcondition $Q$. The statement **assert**$(P)$ means that logical assertion $P$ holds at the current program point. This **assert** is much more powerful than a traditional assert because it takes an assertion in the *logic* instead of a program expression; *i.e.*, $P$ need not be computable.[1]

There are two ways to represent assertions $P$:

- A *syntax* for assertions, with an associated interpretation;
- A *semantic* predicate of the metalogic (*e.g.*, CiC).

We choose to use semantic assertions because they give additional flexibility and save quite a bit of effort. With syntactic assertions, the syntax of assertions must be fixed at the same time as the syntax of the language. However, it is not until one is writing programs (well after the syntax of the language is fixed) that one knows which assertions will be required. With semantic assertions, one can enrich the assertion language at any time by simply defining its meaning in the metalogic. This "late binding" allows users of the programming language to define entire type disciplines or program logics, should they wish. Moreover, by using semantic assertions we get to reuse the binding structure of the metalogic *for free*, which relieves us of the burden of encoding binding structure for quantifiers using de Bruijn indices or some other technique.

In a conventional semantics of Hoare logic with function pointers, $f : \{P\}\{Q\}$ is defined as "$f$ is a function with body $b$ such that $\{P\}\ b\ \{Q\}$" [Sch06, AB07]. This means that assertions are predicates over program syntax. This is simple if assertions are not also embedded in syntax, but if they are then one desires the following:

$$\begin{array}{lcl} \mathsf{predicate} & \equiv & (\mathsf{program} \times \mathsf{memory} \times \ldots) \to \mathbb{T} \\ \mathsf{syntax} & \equiv & \textbf{assert}\ \mathsf{of\ predicate}\ + \\ & & \textbf{assign}\ \mathsf{of\ ident} \times \mathsf{expr}\ +\ \ldots \\ \mathsf{program} & \approx & \mathsf{address} \rightharpoonup \mathsf{syntax}. \end{array}$$

Here the Hoare logic assertions (predicate) judge the program, the memory, and other unspecified objects (*e.g.*, local variables). The program is used to check function-pointer assertions. Program syntax is a tagged disjoint sum and includes the **assert** statement as well as numerous others (*e.g.*, assignment, function call, loops). The program is a partial function from addresses to syntax. We set

$$\begin{array}{lcl} F(X) & \equiv & \mathsf{address} \rightharpoonup (\textbf{assert}\ \mathsf{of}\ X\ +\ \ldots) \\ O & \equiv & \mathsf{memory} \times \ldots, \end{array}$$

and use indirection theory to construct the approximation

$$\mathsf{program} \preceq \mathbb{N} \times (\mathsf{address} \rightharpoonup \mathsf{syntax}).$$

This model for assertions has not been presented previously; the conventional presentation of embeddable assertions requires syntactic predicates. Modifying the techniques of Ahmed *et al.* [AAV03, Ahm04], Appel *et al.* [AMRV07], etc. to build a model for this situation appears intimidating.

The reason directly stratifying over syntax seems difficult is not an issue of theoretical power but rather one of proof engineering. The stratification in those models needs to be built through program syntax, which greatly increases the number of tedious details in the construction. The key innovation that makes constructing the model easy in our approach is the introduction of the functor $F$, which abstracts away the boring "plumbing".

### 2.6 Concurrent separation logic with first-class locks

Concurrent Separation Logic (CSL) is a novel way to reason about concurrent programs [O'H07]. However, originally it had several limitations, most importantly a lack of first-class locks (locks that can be created/destroyed dynamically). Hobor *et al.* and Gotsman *et al.* independently extended CSL to handle first-class locks as well as a number of other features [HAZ08, GBC$^+$07].

The key idea in CSL is that acquiring a lock allows a thread access to additional resources (memory), and releasing a lock relinquishes said resources. The "shape" of the resources acquired or relinquished is described by a predicate of separation logic. First-class locks are challenging to model because the resource invariant of one lock can describe the binding of a resource invariant to another lock. The intuitive model for this contains a circularity:

$$\begin{array}{lcl} \mathsf{res} & \equiv & \mathsf{VAL\ of\ value} + \mathsf{LK\ of\ (share} \times \mathsf{bool} \times \mathsf{pred}) \\ \mathsf{pred} & \equiv & (\mathsf{heap} \times \mathsf{locals}) \to \mathbb{T} \\ \mathsf{heap} & \approx & \mathsf{address} \rightharpoonup \mathsf{res} \end{array}$$

*Heaplets* (heap) are partial functions mapping locations to *resources* (res): either regular data (VAL) or locks (LK). Regular data locations contain *values*. Since multiple threads can each own part of a lock, each lock is associated with a *share*, which tracks *how much* of the lock is controlled. Locks also have a boolean, which is true if *this thread* holds the lock; and a predicate (pred) specifying the lock's resource invariant. We set

$$\begin{array}{rcl} F(X) & \equiv & \mathsf{address} \rightharpoonup (\mathsf{VAL\ of\ value} + \mathsf{LK\ of\ (share} \times \mathsf{bool} \times X)) \\ O & \equiv & \mathsf{locals}, \end{array}$$

and then can use indirection theory to construct the approximation

$$\mathsf{heap} \preceq \mathbb{N} \times (\mathsf{address} \rightharpoonup \mathsf{res}).$$

This example is particularly interesting because the resource maps play dual roles: not only do resource maps allow us to solve the indirect reference problems arising from first-class locks, but they also give us a way to define a separation logic. In §4.2 we will show how to define the points-to and is-a-lock assertions, and in §6 and §7 will develop the rest of a higher-order separation logic.

***Inversion.*** In reasoning about resources one wants to do inversion (case analysis) on the res type; in reasoning about syntax (§2.5), one wants to do inversion on syntax. Indirection theory supports full inversion, unlike our previous models [HAZ08, Hob08].

[1] It is not that we intend to build a machine that can compute whether some arbitrary assertion $P$ holds; it is that we can use $P$ to reason statically about the program, and then erase to an executable program.

### 2.7 Industrial-strength CSL model for Concurrent C minor

Indirection theory is not only mathematically elegant, but also capable of constructing significantly more complicated models than the previous examples. We have used indirection theory to build a model for CSL for Concurrent C minor, a dialect of C with dynamic locks, function pointers, first-class threads, pointer arithmetic, byte and word addressability, and sophisticated sequential control flow [Ler06, HAZ08, Hob08]. Required features included:

- First-class function pointer specifications that can relate function pre- and postconditions and arguments/return values.
- Language independence, *i.e.*, assertions are *not* predicates over program syntax. This is (therefore) a different treatment of the Hoare triple than sketched in §2.5; see [Hob08, Ch. 10].
- First-class locks (*i.e.*, the resource invariant of a lock can refer to other locks, function pointers, etc.).
- Impredicative universal and existential quantification (to enable general polymorphic lock pointers, etc.).
- Equirecursive invariants (to describe list data structures, etc.).
- Semantic assertions embeddable in program syntax.
- Support for a stack pointer, and both local and global variables.
- Substructural accounting to model separating conjunction, using the sophisticated share models of Dockins *et al.* [DHA09].
- Byte addressability, along with a requirement that the four bytes of a word-sized lock must not get separated.

Thus, the model requires a significant superset of the features provided by the models in §2.5 and §2.6. Hobor *et al.* developed the first model that combined all these elements, but it was extremely complicated [HAZ08]; later Hobor presented a partially simplified model, which took dozens of pages to explain [Hob08]. Indirection theory can define the model much more easily.

$$
\begin{array}{lcl}
\mathsf{kind} & \equiv & \mathsf{VAL} + \mathsf{LK} + \mathsf{FUN} + \mathsf{CT} + \mathsf{RES} \\
\mathsf{pred} & \equiv & (\mathsf{rmap} \times \mathsf{mem} \times \mathsf{locals} \times \mathsf{sp} \times \mathsf{gmap}) \to \mathbb{T} \\
\mathsf{preds} & \equiv & \Sigma(A : \mathtt{Type}).\ \mathsf{list}\ (\mathsf{A} \to \mathsf{pred}) \\
\mathsf{res} & \equiv & \mathsf{NO} + \mathsf{YES\ of}\ (\mathsf{kind} \times \mathsf{share} \times \mathsf{preds}) \\
\mathsf{wf\_rm} & : & (\mathsf{address} \to \mathsf{res}) \to \mathbb{T} \quad \equiv \\
\multicolumn{3}{l}{\quad \lambda\psi.\ \forall a, \pi, P.\ \psi(a) = \mathsf{YES}(\mathsf{LK}, \pi, P) \Rightarrow \big((a \bmod 4 = 0) \wedge} \\
\multicolumn{3}{l}{\quad \psi(a+1) = \psi(a+2) = \psi(a+3) = \mathsf{YES}(\mathsf{CT}, \pi, (\mathsf{unit}, \mathsf{nil}))\big)} \\
\mathsf{rmap} & \approx & \Sigma(\psi : \mathsf{address} \to \mathsf{res}).\ \mathsf{wf\_rm}(\psi)
\end{array}
$$

For further explanation see Hobor *et al.* [HAZ08, Hob08]. The function wf_rm enforces alignment for locks. The point is that indirection theory handles this model just like previous examples:

$$\mathsf{rmap} \quad \preceq \quad \mathbb{N} \times (\Sigma(\psi : \mathsf{address} \to \mathsf{res}).\ \mathsf{wf\_rm}(\psi)).$$

### 2.8 Applications using indirection theory

In each of these applications, $F$ is covariant (and the circular pseudoequation on $K$ is contravariant). The point is that it is possible to think of finding all the instances of $X$ "inside" the data structure and performing some operation on each of them. In §10 we discuss extensions to indirection theory, including to bivariant functors.

Most of the applications presented above have been previously solved with a step-indexing model (either on paper or machine-checked). What is new here is the discovery of a uniform pattern into which each of the above examples fit.

We have used indirection theory to prove the type soundness of the polymorphic $\lambda$-calculus with references (from §2.1) in Coq. The proofs took approximately one week to develop.

As mentioned in §2.7, we also use indirection theory to mechanically prove the soundness of a concurrent separation logic with first-class locks and threads for Concurrent C minor [Hob08].

## 3. Axiomatic characterization

***Input.*** Suppose we are given a type $O$ ("other data"), a type $\mathbb{T}$ ("truth values") with distinguished inhabitant $\bot$, and a *covariant functor* $F$. That is, let Type stand for the types of the metalogic; then equip $F : \mathtt{Type} \to \mathtt{Type}$ with a function $\mathsf{fmap} : (A \to B) \to F(A) \to F(B)$ that satisfies the following two axioms:[2]

$$\mathsf{fmap}\ \mathsf{id_A} = \mathsf{id_{F(A)}} \tag{4}$$

$$\mathsf{fmap}\ f \circ \mathsf{fmap}\ g = \mathsf{fmap}\ (f \circ g) \tag{5}$$

The function fmap can be thought of as a generalization of map from functional languages, which applies a function to all the elements in a list. Thus, $\mathsf{fmap}(g)$ should apply $g$ to every $X$ within an $F(X)$. For example, here is fmap for the case of general references in the $\lambda$-calculus from §2.1:

$$\mathsf{fmap} \quad \equiv \quad \lambda g.\ \lambda\psi.\ g \circ \psi$$

Clearly equations (4) and (5) hold for this fmap. The definition of fmap is normally straightforward when the structure of $F$ is known.

***Output.*** Indirection theory now provides provides the following:

$$\mathsf{K} \quad : \quad \mathtt{Type} \tag{6}$$

$$\mathsf{pred} \quad \equiv \quad \mathsf{K} \times O \to \mathbb{T} \tag{7}$$

$$\mathsf{squash} \quad : \quad (\mathbb{N} \times F(\mathsf{pred})) \to \mathsf{K} \tag{8}$$

$$\mathsf{unsquash} \quad : \quad \mathsf{K} \to (\mathbb{N} \times F(\mathsf{pred})) \tag{9}$$

The definitions of K (also called *knot*), squash, and unsquash are abstract. A *predicate* (pred) is a function in the metalogic from a pair of a knot $K$ and other data $O$ to truth values. As explained previously, squash and unsquash are coercion functions between the "small" type knot ($K$) and the "big" type $\mathbb{N} \times F(\mathsf{pred})$.

To coerce an object from the small type to the big one is lossless, but to go from big to small requires approximation. Equations (10) and (11) define the approximation used in indirection theory:

$$\mathsf{level} \quad : \quad \mathsf{K} \to \mathbb{N} \quad \equiv \quad \mathsf{fst} \circ \mathsf{unsquash} \tag{10}$$

$$\mathsf{approx}_n \quad : \quad \mathsf{pred} \to \mathsf{pred} \quad \equiv \quad \lambda p.\ \lambda(k, o).\ \begin{cases} p\ (k, o) & \mathsf{level}\ k\ <\ n \\ \bot & \mathsf{level}\ k\ \geq\ n. \end{cases} \tag{11}$$

The key idea is that knots have *levels*. A knot with a higher level is able to "store more information" than a knot with a lower level. The way to determine the level of a knot is to unsquash it and then take the first component (equation 10). The $\mathsf{approx}_n$ function (equation 11) does the actual approximation by "forgetting" how a predicate behaves on knots with levels greater than or equal to $n$. When an approximated predicate is passed a knot of too high level it just returns the default value $\bot$; if the level is low enough then the underlying original predicate is used.

The behavior of squash and unsquash are specified by the following two axioms, which constitute all of indirection theory:

$$\mathsf{squash}\ (\mathsf{unsquash}\ k) \quad = \quad k \tag{12}$$

$$\mathsf{unsquash}\ (\mathsf{squash}(n, x)) \quad = \quad (n, \mathsf{fmap}\ \mathsf{approx}_n\ x). \tag{13}$$

Equation (12) guarantees that squash and unsquash form a section-retraction pair, and demonstrate that unsquash is lossless. In contrast, squash is lossy; equation (13) precisely specifies where the information is lost. *When an $F(\mathsf{pred})$ is squashed to level $n$, all of the predicates inside it are approximated to level $n$.*

The simplicity of the axioms is a major strength of indirection theory. *The axioms are parametric over $F(X)$*, whereas previous models exposed numerous axioms specialized to their domains. For

[2] Readers familiar with category theory will see the obvious inspiration; others will find that they do not need category theory to follow this paper.

example, Hobor *et al.*'s model for higher-order CSL for Concurrent C minor (outlined in §2.7) had more than fifty axioms, all of which follow from equations (12) and (13) once $F(X)$ is chosen. We view this pleasing fact as evidence that the axiomatization is right. In §9 we prove something stronger: the axiomatization is categorical.

***Corollaries.*** There are easy corollaries to the axiomatization of squash and unsquash. First, equation (12) directly implies that unsquash is injective and that unsquash $\circ$ squash is idempotent. (These two facts hold of any section-retraction pair.)

Second, any predicate (pred) "pulled out" of an unsquashed knot has already been approximated to the level of the knot:

$$\mathsf{unsquash}\ k = (n, F) \ \Rightarrow\ F = \mathsf{fmap}\ \mathsf{approx}_n\ F \tag{14}$$

That is, whenever $(n, F)$ is the result of an unsquash then $F$ is unaffected by approximating it to level $n$. All the information "above" $n$ has already been lost, so throwing it away again has no effect. The proof is straightforward from equations (12) and (13).

Third, a predicate $P$ approximated to level n and then approximated again to level m is equal to $P$ approximated to $\mathsf{min}(n, m)$:

$$\mathsf{approx}_n \circ \mathsf{approx}_m = \mathsf{approx}_{\mathsf{min}(n,m)} \tag{15}$$

Proof: directly from the definition of $\mathsf{approx}_n$ in equation (11).

Fourth, if a knot of level $n$ is first unsquashed and then resquashed to a different level $m$, the predicates in the result have been approximated to level $\mathsf{min}(m, n)$:

$$\begin{array}{c} \mathsf{unsquash}\ k = (n, F) \ \Rightarrow \\ \mathsf{unsquash}(\mathsf{squash}\ (m, F)) = (m, F') \ \Rightarrow \\ F' = \mathsf{fmap}\ \mathsf{approx}_{\mathsf{min(n,m)}}\ F \end{array} \tag{16}$$

Proof: follows directly from equations (14), (13), (5), and (15).

***Implementation.*** In §8 we give a model to prove the soundness of indirection theory. We have implemented the soundness proof as a module in Coq matching the **Input** and **Output** signatures; as Appendix A shows, the axiomatization is remarkably concise. Both signatures are matched opaquely, guaranteeing clean modularity.

***Trustworthiness.*** In §2.8 we explained that both of the examples from §2.1 and §2.7 have been established using machine-checked indirection theory—*i.e.*, using the interface given in Appendix A. Since the proofs are machine checked in Coq, we have a high degree of confidence that they do not have mistakes.

But perhaps the proofs, while correct, are proofs of the wrong theorems! How do we know that the approximations we get with indirection theory are powerful enough to prove nontrivial properties? Could the axioms presented above admit a one-point model? We address this question in several parts by examining our type soundness proof of the polymorphic $\lambda$-calculus with references.

First, we took great care that the definition of safety is entirely standard: for any reachable state, the machine can take another step or it has reached a value. That is, the definition of safety *does not refer to approximation or indirection theory*. Since we define safety in the standard way, at the end of the day we know that we have proved something whose meaning is well-understood.

Second, our typing rules (proved as lemmas from our typing definitions, including those given in §4.1) are entirely standard. It is simple to use the typing rules to type a wide variety of expressions, including expressions that manipulate the heap in interesting ways, run forever, and so forth. Thus we know that our semantic types are powerful enough to use with meaningful programs.

The combination of these two points mean that the axioms preclude a one-point model: if our typing judgment were always true, then we would have a mechanical proof that the polymorphic $\lambda$-calculus with references never gets stuck, a contradiction; on the other hand, if our typing judgment were always false then we would not be able to prove the standard typing rules of the calculus and/or would be unable to apply our typing rules to any programs.

Finally, as mentioned earlier, in §8 we will present a construction, which the reader can examine to determine if it is reasonable (for example, that it has more than one point). Since in §9 we present a proof that all models of the axioms are isomorphic, it is quite reasonable to think about the axioms and the construction we present as two sides of the same coin.

Just as in the polymorphic $\lambda$-calculus with references example, in the soundness proof for concurrent separation logic for Concurrent C minor (which uses indirection theory), the final result is established on a concurrent machine with reasonable semantics (although we assume sequential consistency) [Hob08, Ch. 5]; we also establish a usable set of Hoare rules for proving partial correctness of concurrent programs, and apply them to examples. The combination guarantees that we have demonstrated something meaningful.

Indirection theory is a reformulation of previous step-indexed techniques. Step-indexed models have also been extensively used in mechanical type soundness proofs for general references in von Neumann machines as covered in §2.2. In particular, the FPCC project [AAR$^+$09] used a step-indexed model to prove the safety of the output of the SPARC code emitted by a Standard ML compiler.

## 4. Using indirection theory

Indirection theory constructs models that support a number of powerful—but historically difficult to model—features. Many of these features, such as impredicative quantification and equirecursive predicates, do not depend on the structure of $F(X)$; in §6 we will give models for these kinds of features. Here we examine modeling features that *do* depend on the structure of $F(X)$, focusing on two of the examples presented previously in §2. Applying these ideas to other $F(X)$ is usually not difficult.

### 4.1 General references in the $\lambda$-Calculus

We return to the example of modeling a type system with general references in the polymorphic $\lambda$-calculus from §2.1. Set

$$\begin{array}{rcl} F(X) & \equiv & \mathsf{address} \rightharpoonup X \\ O & \equiv & \mathsf{value}, \end{array}$$

and indirection theory constructs the model

$$\begin{array}{rcl} \mathsf{value} & \equiv & \mathsf{loc\ of\ address} + \mathsf{num\ of}\ \mathbb{N} + \ldots \\ \mathsf{type} & \equiv & (\mathsf{memtype} \times \mathsf{value}) \to \mathbb{T} \\ \mathsf{memtype} & \preceq & \mathbb{N} \times (\mathsf{address} \rightharpoonup \mathsf{type}). \end{array}$$

Recall that $X \preceq Y$ implies the existence of the squash/unsquash section-retraction pair. Substituting $F(X)$ into equations (8) and (9) demonstrates that in this model squash/unsquash have the types

$$\begin{array}{rcl} \mathsf{squash} & : & (\mathbb{N} \times (\mathsf{address} \rightharpoonup \mathsf{type})) \to \mathsf{memtype} \\ \mathsf{unsquash} & : & \mathsf{memtype} \to (\mathbb{N} \times (\mathsf{address} \rightharpoonup \mathsf{type})). \end{array}$$

Thus squash and unsquash provide easy access to the model.

The semantics of the typing judgment $k \vdash v : \tau$, pronounced "in the context of memory typing $k$, value $v$ has type $\tau$," is $\tau(k, v)$. It is simple to use this model to define both the basic type nat as well as the historically difficult to model type ref $\tau$. A value $v$ has type nat if the value is a $\mathsf{num}(n)$ for some $n$:

$$\mathsf{nat} \quad \equiv \quad \lambda(k, v).\ \exists n.\ v = \mathsf{num}(n). \tag{17}$$

This is natural, and the definition of ref $\tau$ is only slightly harder.

The intuition is that a value $v$ has type ref $\tau$ if the value is an address $a$ and according to the memory typing $\psi$, the memory cell at location $a$ has type $\tau$. That is,

$$\begin{array}{rcl} \mathsf{ref}\ \tau & \stackrel{?}{\equiv} & \lambda(k, v).\ \mathsf{let}\ (n, \psi)\ =\ \mathsf{unsquash}\ k\ \mathsf{in} \\ & & \exists a.\ v = \mathsf{loc}(a)\ \wedge\ \psi(a) = \tau. \end{array} \tag{18}$$

The only problem is that since $\psi(a)$ has been extracted from a knot $k$ of level $n$, by equation (14) we know that $\psi(a)$ has been approximated to level $n$—that is,

$$\psi(a) = \mathsf{approx}_n\ \psi(a).$$

Comparing $\psi(a)$ to $\tau$, which may not have been approximated, is too strong. Instead we introduce the idea of *approximate equality*:

$$P =_n Q \quad \equiv \quad \mathsf{approx}_n\ P\ =\ \mathsf{approx}_n\ Q. \tag{19}$$

That is, two predicates (type in this model) are approximately equal at level $n$ if they are equal on all knots of level less than $n$.

With approximate equality it is easy to fix equation 18:

$$\begin{array}{rcl} \mathsf{ref}\ \tau & \equiv & \lambda(k,v).\ \mathsf{let}\ (n,\psi)\ =\ \mathsf{unsquash}\ k\ \mathsf{in} \\ & & \exists a.\ v = \mathsf{loc}(a)\ \wedge\ \psi(a) =_n \tau \end{array} \tag{20}$$

This definition for $\mathsf{ref}\ \tau$ is correct and can type general references in the polymorphic $\lambda$-calculus.[3] We will return to this example in §5.1 to show how a memory and a memory typing are related; we also refer readers interested in more details to the mechanization.

### 4.2 Concurrent separation logic with first-class locks

Our second example is modeling the assertions of a Concurrent Separation Logic with first-class locks from §2.6. We set

$$\begin{array}{rcl} F(X) & \equiv & \mathsf{address} \rightharpoonup (\mathsf{VAL\ of\ value} + \mathsf{LK\ of\ (share} \times \mathsf{bool} \times X)) \\ O & \equiv & \mathsf{locals}, \end{array}$$

and indirection theory constructs the model

$$\begin{array}{lcl} \mathsf{res} & \equiv & \mathsf{VAL\ of\ value} + \mathsf{LK\ of\ (share} \times \mathsf{bool} \times \mathsf{pred}) \\ \mathsf{pred} & \equiv & (\mathsf{heap} \times \mathsf{locals}) \to \mathbb{T} \\ \mathsf{heap} & \preceq & \mathbb{N} \times (\mathsf{address} \rightharpoonup \mathsf{res}). \end{array}$$

Recall that a *share* $\pi$ tracks *how much* of a lock is owned since multiple threads can each own part of the same lock. Here we will define the assertions points-to "$a \mapsto v$" and is-a-lock "$a \overset{\pi}{\rightsquigarrow} P$". The points-to assertion is standard; is-a-lock has a share $\pi$ that indicates how much of the lock is controlled and a predicate $P$ that is the lock's resource invariant. Both of these assertions depend on the structure of $F(X)$. In §7 we will show how to define the separating conjunction and the rest of separation logic applicable to any F(X) that forms a *separation algebra*.

The intuition for points-to is that if $a$ points to $v$ ($a \mapsto v$), then the heaplet $\phi$ contains $\mathsf{VAL}(v)$ at location $a$ and nothing else:

$$\begin{array}{rcl} a \mapsto v & \equiv & \lambda(k,\rho).\ \mathsf{let}\ (n,\phi)\ =\ \mathsf{unsquash}\ k\ \mathsf{in} \\ & & \phi(a) = \mathsf{VAL}(v)\ \wedge\ \mathsf{domain}(\phi) = \{a\}. \end{array} \tag{21}$$

Defining the is-a-lock assertion is equally straightforward if one uses the notion of approximate equality given in equation (19):

$$\begin{array}{rcl} a \overset{\pi}{\rightsquigarrow} P & \equiv & \lambda(k,\rho).\ \mathsf{let}\ (n,\phi)\ =\ \mathsf{unsquash}\ k\ \mathsf{in} \\ \exists P', b.\ \phi(a) = \mathsf{LK}(\pi, b, P') \wedge \mathsf{domain}(\phi) = \{a\} \wedge P =_n P'. & & \end{array} \tag{22}$$

That is, location $a$ is a lock with share $\pi$ and resource invariant $P$ if the resource map $\phi$ contains a $\mathsf{LK}(\pi, b, P')$ for some boolean $b$ and predicate $P'$ at location $a$, $\phi$ is empty everywhere else, and $P$ is approximately equal to $P'$. We do not restrict $b$ because is-a-lock gives permission to (attempt to) lock the lock, but not to read the lock value. This definition is sufficient to model first-class locks.

## 5. Aging the knot

We return to the example of general references in the polymorphic $\lambda$-calculus from §2.1 and §4.1 to demonstrate some aspects of using indirection theory. All of the definitions we give here except (23), (25)–(28), and (32) are parametric over $F(X)$ and therefore applicable to any domain. It is usually not difficult to modify the domain-specific definitions to other $F(X)$.

As in section 4.1, we outline the major aspects of using indirection theory, but do not cover the type soundness proof for the polymorphic $\lambda$-calculus with references in exhaustive detail here. We encourage readers interested in the nuts and bolts of a full soundness proof to examine the mechanization.

### 5.1 A multimodal logic

One remaining question is how a memory typing is related to a memory (function from $\mathsf{address} \to \mathsf{value}$). The intuition is that the memory typing $k$ is a valid typing of the memory $m$ (written $k \vdash m\ \mathsf{valid}$) if all the values in $m$ have the type given by $k$:[4]

$$\begin{array}{rcl} k \vdash m\ \mathsf{valid} & \overset{?}{\equiv} & \mathsf{let}\ (n,\psi)\ =\ \mathsf{unsquash}\ k\ \mathsf{in} \\ & & \forall a.\ k \vdash m(a) : \psi(a). \end{array} \tag{23}$$

Unfortunately, this definition is not quite right. The first problem is that as the $\lambda$-calculus with references steps it allows new memory cells to be allocated, such as during the execution of the expression $\mathsf{new}\ e$. To type the new location, the memory typing must grow as well. But how do we know that our old locations are still well-typed under the new memory typing? There are several choices; here we follow the "multimodal" pattern of Dockins *et al.* [DAH08].

A $\mathsf{world}\ w$ is a pair of knot $k$ and other data $o$. Given a relation $R : \mathsf{world} \to \mathsf{world}$, define the modal operator $\Box_R$ as usual:[5]

$$\Box_R \tau \quad \equiv \quad \lambda w.\ \forall w'.\ wRw' \Rightarrow \tau(w'). \tag{24}$$

We call this setting *multimodal* because we allow several different relations $R$, depending on which part of the world we want to reason about. Our next task is thus to define a relation that will let us reason about the extension of the memory typing.

Define the relation $\mathsf{E}$ (for Extends) as follows:

$$\begin{array}{rcl} (k,o)\mathsf{E}(k',o) & \equiv & \mathsf{let}\ (n,\psi)\ =\ \mathsf{unsquash}\ k\ \mathsf{in} \\ & & \quad \mathsf{let}\ (n',\psi')\ =\ \mathsf{unsquash}\ k'\ \mathsf{in} \\ & & \quad n = n'\ \wedge \\ & & \quad \forall a.\ a \in \mathsf{dom}(\psi) \Rightarrow \psi'(a) = \psi(a). \end{array} \tag{25}$$

That is, $(k,o)\mathsf{E}(k',o')$ if $k$ has the same level as $k$ and $k'$ agrees with $k$ everywhere in the domain of $k$, and is otherwise unrestricted; we also require $o = o'$. Note that $\mathsf{E}$ is reflexive and transitive. Now define the modal operator "$\uparrow$", pronounced "extendedly", as follows:

$$\uparrow \tau \quad \equiv \quad \Box_{\mathsf{E}}\, \tau. \tag{26}$$

Thus, $k \vdash v : \uparrow \tau$ means that $k' \vdash v : \tau$ for any $k'$ that is an extension of $k$. With this operator, we can change (23) into:

$$\begin{array}{rcl} k \vdash m\ \mathsf{valid} & \overset{?}{\equiv} & \mathsf{let}\ (n,\psi)\ =\ \mathsf{unsquash}\ k\ \mathsf{in} \\ & & \forall a.\ k \vdash m(a) : \uparrow \psi(a). \end{array} \tag{27}$$

Since the issue of types being stable under the extension of the memory typing is orthogonal to the action of approximation, we will not comment further on this issue and instead refer readers interested in this aspect to the mechanization.

### 5.2 Applying an extracted predicate to its knot

There is a problem with (27) as well. Notice that $\uparrow \psi(a)$ is being applied to $k$—the very same knot from which $\psi$ was extracted.

[3] A reader may worry that it would be easy to write the incorrect definition (18) by mistake. However, in §5.3 we will define a restricted class of *hereditary* predicates (types) which reject (18) but allow (20).

[4] Recall that the typing judgment $k \vdash v : \tau$ is modeled as $\tau(k, v)$.

[5] Until this point, we have not required any special structure for $\mathbb{T}$ other than the existence of a distinguished element $\bot$. To simplify the presentation, from now on we will assume that $\mathbb{T} = \mathtt{Prop}$, the propositions of the metalogic, and that $\bot$ is the proposition **False**. One could choose some other $\mathbb{T}$, *e.g.* any complete Heyting algebra.

Corollary (14) implies $\psi(a)$ has been approximated to level $n$:

$$\psi(a) = \mathsf{approx}_n\ \psi(a)$$

Thus, in (27), $k \vdash m(a) : \uparrow\psi(a)$ is always $k \vdash m(a) : \uparrow\bot$, which in turn is just $k \vdash m(a) : \bot$. This is exactly where we weaken the pseudomodels to achieve a sound definition: *predicates cannot say anything meaningful about the knot whence they came*. We must weaken (27) so that $k$ is valid if the types in it describe the memory *after $k$ has been approximated further:*

$$\begin{array}{rcl} k \vdash m\ \mathsf{valid} & \equiv & \mathsf{let}\ (n,\psi)\ =\ \mathsf{unsquash}\ k\ \mathsf{in} \\ & & \forall a, n'.\ n > n'\ \Rightarrow\ \mathsf{squash}\ (n',\psi) \vdash m(a) : \uparrow\psi(a) \end{array} \tag{28}$$

By equation (16), $\mathsf{squash}\ (n', \psi)$ is the same as $k$ except the predicates inside it have been further approximated to level $n'$.

We call the process of unsquashing a knot and then resquashing it to a lower level, causing it to become more approximate, *aging the knot*. Since we must do so whenever we wish to use a predicate that we pull out of a knot, we have developed some useful auxiliary definitions. The age1 function reduces the level of a knot by first unsquashing and then resquashing at one level lower, if possible:

$$\mathsf{age1}\ k \equiv \begin{cases} \mathtt{Some}\ (\mathsf{squash}\ (n, x)) & \mathsf{unsquash}\ k = (n{+}1, x) \\ \mathtt{None} & \mathsf{unsquash}\ k = (0, x). \end{cases} \tag{29}$$

The relation $\mathsf{A}$ relates $k$ to its aged version and holds $o$ constant:

$$(k, o)\ \mathsf{A}\ (k', o)\ \equiv\ \mathsf{age1}\ k = \mathtt{Some}\ k'. \tag{30}$$

Note that $\mathsf{A}$ is noetherian (i.e., the world can only be aged a finite number of times) because the level of $k'$ is always decreasing towards 0. Let $\mathsf{A}^+$ denote the irreflexive transitive closure of $\mathsf{A}$.[6] Now define an operator "$\triangleright$" from predicate to predicate:

$$\triangleright\tau \quad \equiv \quad \Box_{A^+}\ \tau. \tag{31}$$

Pronounced "approximately $P$", $\triangleright P\ (w)$ means that $P$ holds on all *strictly* more approximate worlds reachable from $w$ via $\mathsf{A}^+$.

We can use the $\triangleright$ operator to rephrase equation (28):

$$\begin{array}{rcl} k \vdash m\ \mathsf{valid} & \equiv & \mathsf{let}\ (n,\psi)\ =\ \mathsf{unsquash}\ k\ \mathsf{in} \\ & & \forall a.\ k \vdash m(a) : \triangleright\uparrow\psi(a). \end{array} \tag{32}$$

Equation (32) is pleasing because it is quite close in form to the intuitive but flawed (27), with the added benefit of being correct.[7] The key to defining the correct (32) instead of the flawed (27) is to remember a simple rule: *to apply a predicate $P$ to the knot from which it was extracted, one must guard $P$ with the $\triangleright$ operator.*

### 5.3 Hereditary predicates

Let $\mathsf{A}^*$ denote the reflexive, transitive closure of $\mathsf{A}$, and define the modal operator "$\Box$" as follows:

$$\Box\tau \quad \equiv \quad \Box_{\mathsf{A}^*}\ \tau.$$

Let $P$ be a predicate and $w$ be a world such that $P(w)$. Suppose we have a (possibly) more approximate world $w'$ (*i.e.*, $w\mathsf{A}^*w'$). We say that $P$ is *hereditary* if $P(w')$—that is, once $P$ holds on $w$ then it will hold on all further approximations of $w$:

$$\forall w.\ P(w) \Rightarrow \Box P\ (w). \tag{33}$$

Now observe that since $\mathsf{A}^*$ is reflexive, we always have

$$\forall w.\ \Box P\ (w) \Rightarrow P(w).$$

Thus we can rephrase (33) very concisely—$P$ is hereditary iff

$$P = \Box P. \tag{34}$$

Are all predicates hereditary? Unfortunately not: for example, the flawed initial definition for ref $\tau$ (18) is not. However, it is easy to define new hereditary predicates using the logical operators defined in §6. Moreover, the correct definitions for references (20), points-to (21) and is-a-lock (22) are all hereditary. In addition, applying the $\triangleright$ operator makes a predicate hereditary since

$$\triangleright P = \Box \triangleright P. \tag{35}$$

Thus, types pulled out of a valid memory typing (32) are hereditary.

### 5.4 Finite-step proofs and initial knot construction

In soundness proofs that utilize indirection theory, we usually have some knot $k$, for example modeling the memory typing or a resource map. One must age this knot whenever the proof needs to pull a pred $P$ out of $k$, since the application $P(k, o)$ will always be $\bot$. For our polymorphic $\lambda$-calculus with mutable references, we need to age at least on memory access, although in the proofs we choose to age on every step to make the action uniform. For our Concurrent Separation Logic with first class locks and threads, we age at function-call, lock-acquire/release, and fork.

Since our knots have only a finite level, we can age only a finite number of times. Therefore, once we have our initial knot, our safety proof is only good for a number of steps equal to the level of that initial knot. Fortunately, constructing an initial knot of arbitrary level is easy. For example, let us construct a knot (*i.e.*, memory typing) of level $n$ for the $\lambda$-calculus with references.

To make it more interesting, suppose the initial memory typing should have "nat" (equation 17) at location 0 and "ref nat" (equation 20) at location 1. If $\psi$ is this partial function, then $k_{\mathsf{init}}$ is:

$$k_{\mathsf{init}} \quad \equiv \quad \mathsf{squash}(n, \psi).$$

One strength of indirection theory is that the construction of the initial knot is much simpler than in previous step-indexed models. Our proofs are good for any number of steps since it is possible to construct an initial knot of any level.

## 6. Modal/intuitionistic logic

We can use the theory of indirection to construct a possible-worlds model of a modal logic in the style of Appel *et al.*'s "Very Modal Model" (VMM) [AMRV07]. In fact, we do better: the VMM is *almost* a Kripke semantics of intuitionistic logic, but lacks the important condition that all the proposition symbols are hereditary.

To fix this problem, we construct the Kripke model for (propositional) intuitionistic logic, explicitly maintaining the invariant that predicates are hereditary. Let $\mathsf{pred}_H$ be the subset of the predicates that are hereditary. We build a Kripke semantics for our logic in the standard way, with $\mathcal{W} = \mathsf{K}\times O$, and forcing $\models$: $(\mathcal{W}\times\mathsf{pred}_H) \to \mathbb{T}$ is just reversed function application. Then we have:

$$\begin{array}{rcl} w \models \mathsf{true} & \equiv & \top \\ w \models \mathsf{false} & \equiv & \bot \\ w \models p \wedge q & \equiv & w \models p \wedge w \models q \\ w \models p \vee q & \equiv & w \models p \vee w \models q \\ w \models p \Rightarrow q & \equiv & \forall w'.\ w\mathsf{A}^*w' \Rightarrow (w' \models p) \Rightarrow (w' \models q) \\ w \models \forall x : A.\ P(x) & \equiv & \forall x : A.\ w \models P(x) \\ w \models \exists x : A.\ P(x) & \equiv & \exists x : A.\ w \models P(x), \end{array}$$

where the left-hand $\wedge, \vee, \Rightarrow, \forall, \exists$ are the synthesized operators of our logic, and the right-hand $\wedge, \vee, \Rightarrow, \forall, \exists$ are operators of the metalogic. It is straightforward to verify that these definitions are hereditary. Note that we directly lift $\forall$ and $\exists$ from the metalogic,

[6] Here we follow the strategy of Appel *et al.* [AMRV07]; what is new is that we have explicitly built $\mathsf{A}^+$ from the operations of indirection theory.

[7] The "$\uparrow$" and "$\triangleright$" operators commute, so the last line of (32) could just as easily have been written $\forall a.\ k \vdash m(a) : \uparrow\triangleright\psi(a)$. For comparison with Appel *et al.* [AMRV07], their "$\triangleright$" is the same as our "$\triangleright\uparrow$"; we have found it easier to reason about these actions separately [DAH08].

which extends the logic into a higher-order logic. The metavariable $A$ is *any* type of the metalogic (*e.g.*, `A:Type` in Coq), so the quantification is fully impredicative: $A = \mathsf{pred}_H$ works just fine.

These definitions form the standard Kripke model of intuitionistic logic, to which we add the modal operators $\triangleright$ and $\Box$. In comparison to the VMM, the $\Box$ operator is now less important, as all of our logical operators produce hereditary formulae from hereditary subformulae; in contrast, the VMM's implication operator did not do so. However $\Box$ is still useful for coercing an arbitrary $\mathsf{pred}$ (which might not be hereditary) into a $\mathsf{pred}_H$.

We define recursive predicates over contractive operators in the style of the VMM [AMRV07]. We support full equirecursion and so do not require dummy operational steps to pack and unpack recursive data. This gives us the power to reason about traditionally difficult low-level constructs such as compiled function closures.

## 7. Separation logic

Indirection theory, like previous models of impredicative indirection, is used to reason about higher-order invariants on the *contents* of state, *e.g.*, mutable references, locks, objects, or function pointers. On the other hand, Separation Logic is useful for reasoning about the *aliasing* and *noninterference* of these resources. One wants to use both kinds of reasoning simultaneously. Separation Logic combines much more smoothly with indirection theory than with previous models.

We proceed by extending the Kripke semantics of §6 to encompass the the connectives of the logic of Bunched Implications (BI). Our Kripke semantics is similar but not identical to those of Pym [Pym02, Chapter 4] and Restall [Res00, Chapter 11].

We take inspiration from Calcagno *et al.* [COY07], who define structures they call *separation algebras* (SA), which they use as semantic models of separation logic. The main idea is that SAs define a partial operation $\oplus$ which combines two "disjoint" resources. The operation $\oplus$ is used to give semantics to the separating conjunction.

A separation algebra (SA) is a tuple $\langle A, \oplus \rangle$ where $A$ is a set (or `Type`) and $\_ \oplus \_ = \_$ is a ternary relation on $A$ satisfying:

$$x \oplus y = z_1 \;\Rightarrow\; x \oplus y = z_2 \;\Rightarrow\; z_1 = z_2 \tag{36}$$

$$x_1 \oplus y = z \;\Rightarrow\; x_2 \oplus y = z \;\Rightarrow\; x_1 = x_2 \tag{37}$$

$$x \oplus y = z \;\Rightarrow\; y \oplus x = z \tag{38}$$

$$x \oplus y = a \;\Rightarrow\; a \oplus z = b \;\Rightarrow\; \exists c.\, y \oplus z = c \wedge x \oplus c = b \tag{39}$$

$$\forall x.\, \exists u.\, u \oplus x = x \tag{40}$$

$$a \oplus a = b \;\Rightarrow a = b \tag{41}$$

This definition of separation algebras is somewhat different from Calcagno *et al.*'s. First, our definition uses ternary relations rather than partial binary operations. Second, Calcagno *et al.* present a more restrictive version of axiom (40): $\exists u.\, \forall x.\, u \oplus x = x$. Their axiom requires that an SA have a unique unit, while axiom (40) allows different elements of an SA to have different units. This permits knots to be considered as SAs; each stratification level will have its own unit. Third, Calcagno *et al.* lack axiom (41); this axiom is useful and well-justified, but is unrelated to indirection theory. We refer the reader to our paper on separation algebras for further discussion of the advantages of our definition [DHA09].

***A bunched-implication indirection model.*** In §3, we presented an input signature for indirection theory; one of the elements of this signature is a covariant functor $F(X)$ in the category of types with total functions. Here we instead require that $F$ be a functor in the category of separation algebras with *join homomorphisms* (defined below). This means that $F$ is function between SAs and its associated $\mathsf{fmap}$ function preserves the property of being a join homomorphism. If $f$ is a join homomorphism, then so is $\mathsf{fmap} f$.

Let $\langle A, \overset{A}{\oplus} \rangle$ and $\langle B, \overset{B}{\oplus} \rangle$ be SAs. We say that a function $f : A \to B$ is a *join homomorphism* iff, for all $x, y, z \in A$

$$x \overset{A}{\oplus} y = z \;\Rightarrow\; f(x) \overset{B}{\oplus} f(y) = f(z). \tag{42}$$

We equip the naturals $\mathbb{N}$ with the trivial join relation, $i \overset{\mathbb{N}}{\oplus} j = k$ iff $i = j = k$. We require that $\mathbb{T}$ be equipped with some join relation, and that $\bot$ be an identity for that relation: $\bot \overset{\mathbb{T}}{\oplus} \bot = \bot$. We also require that the type $O$ of "other data" be equipped with a join relation; if there is no "interesting" separation to be done on $O$, then the trivial join relation will suffice.

Now we build an SA on "unsquashed" knots. We start by building the SA over $\mathsf{K} \times O \to \mathbb{T}$ by defining the join relation pointwise:

$$p \oplus q = r \;\;\text{iff}\;\; \forall k\, o.\, p\,(k,o) \overset{\mathbb{T}}{\oplus} q\,(k,o) = r\,(k,o). \tag{43}$$

Note that we do not need an SA over $\mathsf{K}$ for this definition. Next, we get an SA on $F(\mathsf{K} \times O \to \mathbb{T})$ because we required $F$ to be a functor over SAs. Finally, we get an SA over $\mathbb{N} \times F(\mathsf{K} \times O \to \mathbb{T})$ by joining componentwise (with the trivial SA on $\mathbb{N}$). Verifying the SA axioms for these constructions is straightforward [DHA09].

The SA on knots is defined by simply stating:

$$k_1 \oplus k_2 = k_3 \;\Leftrightarrow\; \mathsf{unsquash}\; k_1 \oplus \mathsf{unsquash}\; k_2 = \mathsf{unsquash}\; k_3. \tag{44}$$

The SA axioms for this construction follow from the fact that $\mathsf{squash} \circ \mathsf{unsquash}$ is the identity function (one of the axioms of indirection theory) and the fact that $\mathsf{unsquash} \circ \mathsf{squash}$ is a join homomorphism. This follows because $F$ preserves join homomorphisms and because $\mathsf{approx}_n$ is a join homomorphism, which we deduce from its definition and the fact that $\bot$ is an identity element.

Thus we see that the separation algebra on knots is determined by the SA structure embedded in the functor $F$, which allows the user to specify the interesting part of the SA as input.

***An example.*** Consider the model for first-class locks from §2.6:

$$\begin{aligned} F(X) &\equiv \mathsf{address} \rightharpoonup (\mathsf{VAL\ of\ value} + \mathsf{LK\ of\ (share} \times \mathsf{bool} \times X)) \\ O &\equiv \mathsf{locals}. \end{aligned}$$

Given an SA over $X$, we want to define an SA over $F(X)$. The intuition here is that we want the partial functions to join pointwise such that undefined locations join with any defined location, value locations join *only* with undefined locations, and lock locations join provided that at most one thread holds the lock at a time.

Formally, we add an extra (pseudo)element $\mathsf{u}$ to the right-hand side of $\rightharpoonup$ to represent unowned locations. Assume we have a join relation on shares (see [DHA09]). Define an SA on booleans:

$$b_1 \overset{B}{\oplus} b_2 = b_3 \;\text{ iff }\; (b_1 \,\&\&\, b_2 = \mathbf{F}) \;\wedge\; (b_1 \,||\, b_2 = b_3).$$

The join relation of the RHS is the smallest relation such that:

$$\mathsf{u} \oplus a = a \tag{a}$$

$$a \oplus \mathsf{u} = a \tag{b}$$

$$\begin{aligned} s_1 \overset{S}{\oplus} s_2 = s_3 \;\Rightarrow\; b_1 \overset{B}{\oplus} b_2 = b_3 \;\Rightarrow\; x_1 \overset{X}{\oplus} x_2 = x_3 \\ \Rightarrow\; \mathsf{LK}\,(s_1, b_1, x_1) \oplus \mathsf{LK}\,(s_2, b_2, x_2) = \mathsf{LK}\,(s_3, b_3, x_3). \end{aligned} \tag{c}$$

In equation (c), the $s_i$ are required to be *nonunit* shares.

Finally we define the SA over heaplets pointwise on the address:

$$f \oplus g = h \;\text{ iff }\; \forall \ell.\, f(\ell) \oplus g(\ell) = h(\ell)$$

This model gives rise to an SA where heaplets must have disjoint value domains, but where separated heaplets may each have some share of a lock. The SA over shares keeps track of lock *visibility* (the ability to compete to acquire the lock) whereas the boolean value keeps track of whether this thread holds the lock.

When defining an SA over $O$ (the local variables), we can either choose to use a trivial SA, or we can choose to define a similar pointwise SA. The second choice leads to a "variables as resources" [PBC06] style of separation logic, whereas the former choice leads to a presentation where local variables are not separated.

SAs for other models can be built by using various combinations of products, coproducts, and functions, etc. See Dockins *et al.* for examples and explicit constructions of share models [DHA09].

***Operators of separation logic.*** Using the separation algebra over knots we are almost ready to state the Kripke semantics for BI. The Kripke semantics is defined over pairs $\mathsf{K} \times O$, so we lift the SA over $\mathsf{K}$ to pairs, componentwise.

$$\begin{aligned} w \models \mathsf{emp} &\equiv w \oplus w = w \\ w \models p * q &\equiv \exists w_1\ w_2.\ w_1 \oplus w_2 = w \wedge w_1 \models p \wedge w_2 \models q \\ w \models p \mathrel{-\!\!*} q &\equiv \forall w'\ w_1\ w_2.\ w\, \mathsf{A}^* w' \Rightarrow \\ &\qquad w' \oplus w_1 = w_2 \Rightarrow w_1 \models p \Rightarrow w_2 \models q \end{aligned}$$

In order for these definitions to be valid, we must show that they are hereditary. The fact that $-\!\!*$ is hereditary follows immediately from the definition. In order to show that $\mathsf{emp}$ and $*$ are hereditary, we require the following technical facts about knots:

$$\begin{aligned} &\forall k_1\ k_2\ k_3\ k_1'.\ k_1 \oplus k_2 = k_3 \Rightarrow k_1 \mathsf{A} k_1' \Rightarrow \\ &\quad \exists k_2'\ k_3'.\ k_1' \oplus k_2' = k_3' \wedge k_2 \mathsf{A} k_2' \wedge k_3 \mathsf{A} k_3' \end{aligned} \tag{45}$$

$$\begin{aligned} &\forall k_1\ k_2\ k_3\ k_3'.\ k_1 \oplus k_2 = k_3 \Rightarrow k_3 \mathsf{A} k_3' \Rightarrow \\ &\quad \exists k_1'\ k_2'.\ k_1' \oplus k_2' = k_3' \wedge k_1 \mathsf{A} k_1' \wedge k_2 \mathsf{A} k_2' \end{aligned} \tag{46}$$

These facts follow easily from the definition of the SA over knots, the definition of $\mathsf{A}$, and the fact that $\mathsf{approx}_n$ is a join homomorphism. Hereditarity for $\mathsf{emp}$ and $*$ then follows by induction on the transitivity of $\mathsf{A}^*$, using (45) and (46) in the base cases.

It is relatively easy to show that these definitions, together with the definitions from §6, form a model of the logic of Bunched Implications. This is most easily done by proving that the model satisfies the axioms of a Hilbert system of BI [Pym02, Table 3.1].

# 8. Model construction

Here we construct a model for the axioms in §3. Indirection theory is built in the Calculus of Inductive Constructions (CiC) with the axiom of functional extensionality.[8] Our model uses a *step-indexing* technique originally developed by Ahmed *et al.* [AAV03], but we have made many improvements and simplifications.

Indirection theory is so compact that the definition, construction, and soundness proofs require only 400 lines to implement in Coq. The construction is parameterized over $F(X)$, making it directly applicable to all of the examples given in §2. Moreover, the construction is more powerful than previous models; for example, as explained in §2.6, indirection theory supports full inversion while the previous models of Hobor *et al.* do not [HAZ08, Hob08].

***Use of dependent types.*** The proofs are tricky to mechanize due to the dependent types in the indexed-model construction. There are no dependent types in the *axiomatization*, so users of indirection theory are not burdened with them. This means that indirection theory can be used (although not, we conjecture, proved sound) in metalogics without dependent types, such as the simple theory of types (HOL). This is a significant strength of our approach.

***Presentation.*** We elide the handling of the "other" data $O$ which appears in the axiomatization. Adding it presents no fundamental difficulties, but it clutters the explanation.

[8] We have also developed an axiom-free version by working in the category of setoids (whose objects are types equipped with an equivalence relation and whose morphisms are equivalence-preserving functions). The proofs are not much more difficult to carry out, just more difficult to state.

## 8.1 Tying the knot.

We wish to define a type $\mathsf{K}$ that is similar to the pseudodefinition in equation (2), reproduced here (without $O$):

$$\mathsf{K} \quad \approx \quad F(\mathsf{K} \to \mathbb{T}).$$

We proceed by defining an approximation to $\mathsf{pred}$ called $\mathsf{pred}_n$, a finitely stratified type constructor indexed by the natural $n$:

$$\mathsf{pred}_n \equiv \begin{cases} \mathsf{unit} & n = 0 \\ \mathsf{pred}_{n'} \times (F(\mathsf{pred}_{n'}) \to \mathbb{T}) & n = n' + 1. \end{cases} \tag{47}$$

For all $n > 0$, $P : \mathsf{pred}_n$ is a pair whose second component $P.2$ is an approximation to the recursive pseudodefinition (2) and whose first component $P.1$ is a simpler approximation $P' : \mathsf{pred}_{n-1}$.

A knot ($\mathsf{K}$) hides the index with a dependent sum:

$$\mathsf{K} \equiv \Sigma(n : \mathbb{N}).\ F(\mathsf{pred}_n). \tag{48}$$

That is, a knot is a dependent pair of a natural $n$ and an $F$ with elements of $\mathsf{pred}_n$ inside. Define the *level* of a knot $k$ as:

$$|k| = k.1 \qquad \text{that is,} \qquad |(n, f)| = n. \tag{49}$$

A knot's level gives how many layers of stratification it contains.

Now we can define the type of predicates ($\mathsf{pred}$):

$$\mathsf{pred} = \mathsf{K} \to \mathbb{T}. \tag{50}$$

A $\mathsf{pred}$ is an "infinitely stratified" $\mathsf{pred}_n$ that can judge a knot containing any amount of stratification.

## 8.2 Stratification and unstratification.

We define a function $\mathsf{strat}_n : \mathsf{pred} \to \mathsf{pred}_n$ that collapses an infinitely stratified $\mathsf{pred}$ into a finitely stratified $\mathsf{pred}_n$:

$$\mathsf{strat}_n(P) = \begin{cases} () & n = 0 \\ \big(\mathsf{strat}_{n'}(P),\ \lambda f.\ P(n', f)\big) & n = n' + 1. \end{cases} \tag{51}$$

The $\mathsf{strat}_n$ function constructs the list structure of $\mathsf{pred}_n$ by recursively applying $P$ to knots of decreasing level. The finitely stratified type $\mathsf{pred}_n$ is not big enough to store the behavior of $P$ on knots of level $\geq n$, and so that information is thrown away.

The $\mathsf{strat}_n$ function is a standard feature in step-indexing constructions [AAV03, AMRV07, Hob08]. A key innovation in our new construction is the definition of a partial inverse to $\mathsf{strat}_n$.[9]

First we define a "floor" operator that given a $P : \mathsf{pred}_{n+m}$ constructs a $P' : \mathsf{pred}_n$ by stripping off the outer $m$ approximations:

$$\lfloor P \rfloor_n^m \equiv \begin{cases} P & m = 0 \\ \lfloor P.1 \rfloor_n^{m'} & m = m' + 1. \end{cases} \tag{52}$$

Now observe that for any $a$, $b \in \mathbb{N}$,

$$b > a \quad \Leftrightarrow \quad \exists m.\ b = a + m + 1. \tag{53}$$

We now define $\mathsf{unstrat}_n : \mathsf{pred}_n \to \mathsf{pred}$, which takes a finitely stratified $\mathsf{pred}_n$ and constructs an infinitely stratified $\mathsf{pred}$ from it. We use (53) to take cases and apply (52) in a constructive way:

$$\mathsf{unstrat}_n(P) = \lambda k.\ \begin{cases} (\lfloor P \rfloor_{|k|+1}^m .2)\ k.2 & n = |k| + m + 1 \\ \bot & n \leq |k|. \end{cases} \tag{54}$$

When given a knot of level $< n$, the $\mathsf{unstrat}_n$ function uses the floor operator to "look up" how to behave. When applied to a knot $k$ of level $\geq n$, the $\mathsf{unstrat}_n$ function returns $\bot$ since the $\mathsf{pred}_n$ $P$ does not contain any way to judge $k$.

[9] Although the supporting proofs for VMM [AMRV07] contained a similar inverse, it was not explained in the paper and played only a supporting role. Here we show that $\mathsf{unstrat}$ is actually of central importance.

Below we will need the following two technical facts. First, the floor operation commutes with taking the first projection:

$$\lfloor P \rfloor^{m}_{n+1}.1 = \lfloor P.1 \rfloor^{m}_{n} \tag{55}$$

*Proof of (55).* By induction on $m$. □

Due to the dependent types, the statement of this lemma and its proof are somewhat more complicated in the Coq development; nonetheless, the overall proof idea is straightforward.

The second technical lemma specifies what happens when two similar finitely stratified predicates are restratified at a lower level.

$$\begin{array}{l}\lfloor P_1 \rfloor^{m_1}_{n} = \lfloor P_2 \rfloor^{m_2}_{n} \;\Rightarrow \\ (\mathsf{strat}_n \circ \mathsf{unstrat}_{n+m_1})\ P_1 = (\mathsf{strat}_n \circ \mathsf{unstrat}_{n+m_2})\ P_2\end{array} \tag{56}$$

If $P_1 : \mathsf{pred}_{n+m_1}$ and $P_2 : \mathsf{pred}_{n+m_2}$ are two finitely stratified predicates that are equal in their first $n$ layers, then they are equal after they have gone through $\mathsf{unstrat}_{n+j}$ followed by $\mathsf{strat}_n$.

*Proof of (56).* By induction on $n$. The base case is easy; the inductive case follows by the induction hypothesis and (55). □

### 8.3 Composing $\mathsf{strat}_n$ and $\mathsf{unstrat}_n$.

What happens when we compose $\mathsf{strat}_n$ and $\mathsf{unstrat}_n$? We take this in two parts. First we wish to show:

$$\mathsf{strat}_n \circ \mathsf{unstrat}_n \;=\; \mathsf{id}_{\mathsf{pred}_n} \tag{57}$$

where $\mathsf{id}_\alpha$ is the identity function on type $\alpha$ (*i.e.*, $\mathsf{id}_\alpha \equiv \lambda x : \alpha.\ x$).

*Proof of (57).* By induction on $n$. In the base case, the claim follows because $\mathsf{strat}_0 f = ()$ for all $f$ and the unit type has a unique value. In the inductive case, where $n = n' + 1$, after unfolding $\mathsf{strat}_{n'+1}$ we must show (for arbitrary $P$ and $f$):

$$\begin{array}{rcll}\mathsf{strat}_{n'}(\mathsf{unstrat}_{n'+1}\ P) & = & P.1 & \textit{and} \\ \mathsf{unstrat}_{n'+1}(P)\ (n', f) & = & P.2\ f. & \end{array}$$

To demonstrate the first equation, first observe

$$\lfloor P \rfloor^{1}_{n'} = \lfloor P.1 \rfloor^{0}_{n'}$$

so we may rewrite using (56) into $\mathsf{strat}_{n'}(\mathsf{unstrat}_{n'}(P.1))$ and apply the induction hypothesis. In the second equation, note that

$$n' + 1 = |(n', f)| + 0 + 1,$$

which means that

$$\mathsf{unstrat}_{n'+1}(P)(n', f) = (\lfloor P \rfloor^{0}_{n'+1}.2)\ ((n', f).2) = P.2\ f.$$

This completes the first part. □

For $\mathsf{unstrat}_n \circ \mathsf{strat}_n$, we cannot do as well since $\mathsf{strat}_n$ "forgets" information that $\mathsf{unstrat}_n$ cannot recover. Give $\mathbb{T}$ a flat partial order by defining $\sqsubseteq$ as the minimal relation such that:

$$x \sqsubseteq x \qquad \text{reflexivity} \tag{58}$$

$$\bot \sqsubseteq x \qquad \text{pointedness} \tag{59}$$

It is clear that $\sqsubseteq$ is antisymmetric. With $\sqsubseteq$ we can state three cases:

$$(\mathsf{unstrat}_n \circ \mathsf{strat}_n)\ P\ k \sqsubseteq P\ k \quad | \ \forall\, |k| \tag{60}$$

$$P\ k \sqsubseteq (\mathsf{unstrat}_n \circ \mathsf{strat}_n)\ P\ k \ \mid\ |k| < n \tag{61}$$

$$(\mathsf{unstrat}_n \circ \mathsf{strat}_n)\ P\ k \sqsubseteq \bot \quad | \ |k| \geq n \tag{62}$$

*Proof of (60).* By induction on $n$. In the base case $n = 0$ and $\mathsf{unstrat}_0(P) = \bot$ for all $P$; the claim follows from (59). In the inductive case $n = n' + 1$. Consider the two subcases of the definition of $\mathsf{unstrat}$. When $n \leq |k|$, $\mathsf{unstrat}_n(P) = \bot$ for all $P$, and we are done, again by (59). In the case where $n = |k| + m + 1$, we proceed by subcases on the value of $m$. If $m = 0$, then observe that $n' = |k|$, and we have:

$$\begin{array}{rcl}\mathsf{unstrat}_{n'+1}(\mathsf{strat}_{n'+1}\ P)\ k & & \\ = & (\lfloor \mathsf{strat}_{n'+1}\ P \rfloor^{0}_{|k|+1}.2)\ (k.2) & \\ = & (\mathsf{strat}_{n'+1}\ P).2\ (k.2) & \\ = & P\ (n', k.2) = P\ (|k|, k.2) = P\ k. & \end{array}$$

Then the claim follows by (58). In the case that $m = m' + 1$, then:

$$\begin{array}{rcl}\mathsf{unstrat}_{n'+1}(\mathsf{strat}_{n'+1}\ P)\ k & & \\ = & (\lfloor \mathsf{strat}_{n'+1}\ P \rfloor^{m'+1}_{|k|+1}.2)\ (k.2) & \\ = & (\lfloor (\mathsf{strat}_{n'+1}\ P).1 \rfloor^{m'}_{|k|+1}.2)\ (k.2) & \\ = & (\lfloor \mathsf{strat}_{n'}\ P \rfloor^{m'}_{|k|+1}.2)\ (k.2) & \\ = & (\mathsf{unstrat}_{n'}(\mathsf{strat}_{n'}(P))\ k & \end{array}$$

The final line follows $n' = |k| + m' + 1$, which allows us to fold $\mathsf{unstrat}_{n'}$. Now we apply the induction hypothesis. □

*Proof of (61).* By induction on $n$. In the base case $n = 0$, and $|k| < 0$ yields a contradiction. In the inductive case, we have $n = n' + 1$. Since $|k| < n$, we must have $n = |k| + m + 1$ for some $m$. As before, we inspect the value of $m$. if $m = 0$, the claim follows by (58) and calculation identical to the preceding. If $m = m'+1$ then we again perform similar calculation as above and apply the induction hypothesis. We must only verify that $|k| < n'$, which follows because $n' = |k| + m' + 1$. □

*Proof of (62).* Immediate from the definition of $\mathsf{unstrat}$. □

Since $\sqsubseteq$ is pointed and antisymmetric, (60), (61), and (62) imply:

$$(\mathsf{unstrat}_n \circ \mathsf{strat}_n)\ P\ k = \begin{cases} P\ k & |k| < n \\ \bot & |k| \geq n. \end{cases} \tag{63}$$

### 8.4 Squashing and unsquashing.

Define $\mathsf{squash}$ and $\mathsf{unsquash}$ to $\mathsf{fmap}\ \mathsf{strat}_n$ and $\mathsf{unstrat}_n$:

$$\mathsf{squash}(n, f) \;\equiv\; (n, \mathsf{fmap}\ \mathsf{strat}_n\ f) \tag{64}$$

$$\mathsf{unsquash}(k) \;\equiv\; (|k|, \mathsf{fmap}\ \mathsf{unstrat}_{|k|}\ k.2). \tag{65}$$

Recall from §3 the definitions of $\mathsf{level}$ and $\mathsf{approx}_n$ (without $O$):

$$\mathsf{level} \;\equiv\; \mathsf{fst} \circ \mathsf{unsquash} \tag{66}$$

$$\mathsf{approx}_n\ P\ k \;\equiv\; \begin{cases} P\ k & \mathsf{level}\ k < n \\ \bot & \mathsf{level}\ k \geq n. \end{cases} \tag{67}$$

It follows immediately from (64) and (66) that $\mathsf{level}\ k = |k|$. Applying this equality to (63) and (67) shows:

$$\mathsf{approx}_n = \mathsf{unstrat}_n \circ \mathsf{strat}_n. \tag{68}$$

We now prove the two axioms of indirection theory. Proof of (12):

$$\begin{array}{rcll}\mathsf{squash}\ (\mathsf{unsquash}\ k) & & & \\ = & \mathsf{squash}\ (|k|, \mathsf{fmap}\ \mathsf{unstrat}_{|k|}\ k.2) & & (65) \\ = & (|k|, \mathsf{fmap}\ \mathsf{strat}_{|k|}\ (\mathsf{fmap}\ \mathsf{unstrat}_{|k|}\ k.2)) & & (64) \\ = & (|k|, \mathsf{fmap}\ (\mathsf{strat}_{|k|} \circ \mathsf{unstrat}_{|k|})\ k.2)) & & (5) \\ = & (|k|, \mathsf{fmap}\ \mathsf{id}_{\mathsf{pred}_{|k|}}\ k.2) & & (57) \\ = & (|k|, \mathsf{id}_{F(\mathsf{pred}_{|k|})}\ k.2) & & (4) \\ = & k & & (49) \end{array}$$

The second axiom (13) follows in a similar way:

$$\begin{array}{lll} \mathsf{unsquash}\ (\mathsf{squash}\ (n, f)) & & \\ \quad = \quad \mathsf{unsquash}\ (n, \mathsf{fmap}\ \mathsf{strat}_n\ f) & \Big| & (64) \\ \quad = \quad (n, \mathsf{fmap}\ \mathsf{unstrat}_n\ (\mathsf{fmap}\ \mathsf{strat}_n\ f)) & & (65) \\ \quad = \quad (n, \mathsf{fmap}\ (\mathsf{unstrat}_n \circ \mathsf{strat}_n)\ f)) & & (5) \\ \quad = \quad (n, \mathsf{fmap}\ \mathsf{approx}_n\ f) & & (68) \end{array}$$

This completes the construction. □

## 9. Uniqueness of the model

One remarkable property of the axiomatization given in §3 is that it is *categorical*: the axioms determine the model uniquely up to isomorphism. We prove this by constructing a bijection between any two models and prove that the bijection preserves the squash and unsquash operations. For simplicity, we will elide the type $O$.

***Input.*** We assume that we have a single input ($F$, fmap). We then take two (possibly different) constructions $A$ and $B$. To distinguish the functions and properties defined on one construction from the other, we will write, *e.g.*, $\mathsf{squash}_A$ or $\mathsf{squash}_B$.

***Informal goal and core proof idea.*** Our goal is to construct an isomorphism $f : K_A \to K_B$. At first this seems quite easy:

$$f \stackrel{?}{\equiv} \mathsf{squash}_B \circ \mathsf{unsquash}_A. \tag{69}$$

That is, take a $k_A : K_A$, unsquash it to get to a common form (of type $\mathbb{N} \times F(\mathsf{pred})$), and then squash it into an element of type $K_B$. Unfortunately, this approach does not work since $\mathsf{unsquash}_A$ produces an element of type $\mathbb{N} \times F(\mathsf{pred}_A)$, while $\mathsf{squash}_B$ requires an element of type $\mathbb{N} \times F(\mathsf{pred}_B)$. We cannot assume that $\mathsf{pred}_A$ is compatible with $\mathsf{pred}_B$ without begging the question, so (69) is invalid. Part of what makes the problem tricky is that the types are isomorphic but need not be equal, so we must build up significant machinery to coerce the types from one construction to the other.

***Formal goal.*** We want a pair of functions $f : K_A \to K_B$ and $g : K_B \to K_A$. These functions will be inverses:

$$f \circ g = \mathsf{id}_{K_B} \tag{70}$$

$$g \circ f = \mathsf{id}_{K_A}. \tag{71}$$

Thus, $f$ and $g$ are bijective. To be isomorphisms they must also be homomorphisms (preserve squash and unsquash); for this we introduce two derived functions, $\Phi : F(\mathsf{pred}_A) \to F(\mathsf{pred}_B)$:

$$\Phi \equiv \mathsf{fmap}\ (\lambda P_B.\ P_B \circ g), \tag{72}$$

and $\Gamma : F(\mathsf{pred}_B) \to F(\mathsf{pred}_A)$:

$$\Gamma \equiv \mathsf{fmap}\ (\lambda P_A.\ P_A \circ f). \tag{73}$$

Since $f$ and $g$ are inverses, and fmap distributes over function composition (equation 5), $\Phi$ and $\Gamma$ are inverses as well:

$$\Phi \circ \Gamma = \mathsf{id}_{F(\mathsf{pred}_B)} \tag{74}$$

$$\Gamma \circ \Phi = \mathsf{id}_{F(\mathsf{pred}_A)}. \tag{75}$$

We can now state that $f$ and $g$ are homomorphisms as follows:

$$f(\mathsf{squash}_A(n, \psi_A)) = \mathsf{squash}_B(n, \Phi(\psi_A)) \tag{76}$$

$$g(\mathsf{squash}_B(n, \psi_B)) = \mathsf{squash}_A(n, \Gamma(\psi_B)) \tag{77}$$

$$\begin{array}{l} \mathsf{unsquash}_B(f(k_A)) = (n, \Phi(\psi_A)) \\ \quad \mathsf{where}\ (n, \psi_A) = \mathsf{unsquash}_A(k_A) \end{array} \tag{78}$$

$$\begin{array}{l} \mathsf{unsquash}_A(g(k_B)) = (n, \Gamma(\psi_B)) \\ \quad \mathsf{where}\ (n, \psi_B) = \mathsf{unsquash}_B(k_B). \end{array} \tag{79}$$

This will complete the proof that $A$ and $B$ are isomorphic.

***Proof sketch.*** The formal details of the proof are quite technical; here we give an informal outline. We have implemented our proof in Coq and refer readers seeking more detail to the mechanization.

We construct $f$ and $g$ from a sequence of approximations $(f_0, g_0)$,...,$(f_m, g_m)$,.... A pair of functions $(f_i, g_i)$ are inverses and homomorphisms *on knots of level* $\leq i$.

First we construct $f_0$ and $g_0$. By equation (14), all of the predicates inside knots of level 0 are approximated to level 0—that is, they are all constant functions returning $\bot$. We write $\bot_\alpha$ to mean the constant function from type $\alpha$ to $\bot$ (*i.e.*, $\bot_\alpha \equiv \lambda a : \alpha.\ \bot$). The functions $f_0$ and $g_0$ simply swap $\bot_{K_A}$ with $\bot_{K_B}$ and vice versa:[10]

$$\begin{array}{lcll} f_0(k_A) & \equiv & \mathsf{let}\ (n, \psi_A) = \mathsf{unsquash}_A\ k_A\ \mathsf{in} & \\ & & \quad \mathsf{squash}_B(n, [P_A \Rightarrow \bot_{K_B}]\ \psi_A) & (80) \\ g_0(k_B) & \equiv & \mathsf{let}\ (n, \psi_B) = \mathsf{unsquash}_B\ k_B\ \mathsf{in} & \\ & & \quad \mathsf{squash}_A(n, [P_B \Rightarrow \bot_{K_A}]\ \psi_B). & (81) \end{array}$$

**Lemma.** $(f_0, g_0)$ is an isomorphism pair on knots of level 0.

Next, given $(f_i, g_i)$ we wish to construct the next pair in the series $(f_{i+1}, g_{i+1})$. The trick is that due to the contravariance of pred, $f_{i+1}$ is built from $g_i$, and $g_{i+1}$ is built from $f_i$:

$$\begin{array}{lcll} \hat{f}(\gamma)(k_A) & \equiv & \mathsf{let}\ (n, \psi_A) = \mathsf{unsquash}_A\ k_A\ \mathsf{in} & \\ & & \quad \mathsf{squash}_B(n, [P_A \Rightarrow P_A \circ \gamma]\ \psi_A) & (82) \\ \hat{g}(\phi)(k_B) & \equiv & \mathsf{let}\ (n, \psi_B) = \mathsf{unsquash}_B\ k_B\ \mathsf{in} & \\ & & \quad \mathsf{squash}_A(n, [P_B \Rightarrow P_B \circ \phi]\ \psi_B). & (83) \end{array}$$

**Lemma.** If $(f, g)$ is isomorphism pair for knots of level $\leq i$, then $(\hat{f}(g), \hat{g}(f))$ is an isomorphism pair for knots of level $\leq i + 1$.

Now define the function $h$ that takes $n$ and produces $(f_n, g_n)$:

$$h(n) \equiv \begin{cases} (f_0, g_0) & n = 0 \\ \mathsf{let}\ (f', g') = h(n')\ \mathsf{in}\ (\hat{f}(g'), \hat{g}(f')) & n = n' + 1. \end{cases} \tag{84}$$

The $h$ function works by bouncing back and forth; note that even-numbered $f_i$ have $f_0$ as their base while odd-numbered $f_i$ have $g_0$. The $g_i$ are mirrored: even $g_i$ use $g_0$ for a base while odd $g_i$ use $f_0$.

Now we are ready to define $f$ and $g$. The idea is that we will look at the knot $k$ that we have been given and use the $h$ function to construct an $f_i$ or $g_i$ of the appropriate level:

$$f(k_A) \equiv \big((\mathsf{fst} \circ h \circ \mathsf{level}_A)(k_A)\big)(k_A) \tag{85}$$

$$g(k_B) \equiv \big((\mathsf{snd} \circ h \circ \mathsf{level}_B)(k_B)\big)(k_B). \tag{86}$$

**Theorem.** ($f$,$g$) is an isomorphic pair for knots of any level.

***Implications.*** Categorical axiomatizations are sufficiently uncommon that we examine the implications. Most importantly, the axioms of indirection theory given in §3 are in some sense complete: they define a particular class of models in a definitive way. Moreover, there seems to be little point in developing alternatives to the construction we presented in §8, at least in CiC. We view these facts as powerful evidence that our axioms are the correct way to characterize current step-indexing models.

## 10. Extensions of indirection theory

Although many problem domains of interest fall into the simple form of pseudoequation (2), some problem domains require predicates to appear in negative positions (the left side of arrows). For example, the recent result of Ahmed *et al.*, which applies step-indexing to reason about data abstraction [ADR09], requires such flexibility. We have defined, on paper and in Coq, a straightforward

[10] Keeping with our informal style, we will abuse notation by writing $[P_\alpha \Rightarrow X]\ \psi$ to mean the substitution of the predicates $\mathsf{pred}_\alpha$ in $\psi$ with $X$. Formally this is written $\mathsf{fmap}\ (\lambda P_\alpha.\ X)\ \psi$. Note $P_\alpha$ **can** appear in $X$.

extension to indirection theory that handles *bifunctors*, which allow equations containing mixed variance.

In our axiomatization of indirection theory, nonhereditary predicates can pop out of the unsquash function. If we make sure to place $\Box$ in front of any predicate taken from a knot, we avoid this problem; but it is not clear that this approach extends to the bifunctor case. Instead, we can do something even more powerful: we have an alternate axiomatization that guarantees that all predicates in knots are hereditary. We have a model proved in Coq (and on paper), but unfortunately it is not as straightforward to construct. We hypothesize that this model could be used to mechanize the recent result of Dreyer *et al.* [DNRB10].

## 11. Limitations of indirection theory

Indirection theory is a powerful technique for those problems to which it applies, but it is worthwhile to examine some limitations.

First, indirection theory only builds solutions for the *particular class* of recursive domain equations given by pseudoequation (2). Although, as seen in §2, the form of this pseudoequation was carefully chosen to cover many problem domains of interest, it is manifestly not applicable to equations of other forms (although in §10 we cover some possible extensions).

Second, our insistence on remaining in the simple category of sets (or types) means that any solution we build must *necessarily* involve approximation; as we previously noted, a simple cardinality argument shows that we cannot obtain isomorphism solutions. We have found this cost to be quite manageable, especially when managed using the framework of the Very Modal Model [AMRV07].

Finally, step-indexing techniques have thus far been applied only to questions of safety, *partial* correctness, and program equivalence. It is unclear how, or if, step indexing approaches can be applied to questions of liveness and *total* correctness.

## 12. Related work

***Syntactic methods.*** There is a long history of using syntactic methods to handle indirection. Early papers on Hoare logic used syntactic representations of program text to reason about function calls [CO78]. Syntactic accounts of general references have been studied by Harper [Har94], Harper and Stone [HS97], Wright and Felleisen [WF94], and others. Crary developed TALT, a typed assembly language which had indirection due to both data references and code pointers [Cra03].

Defining Hoare-style logics using syntactic methods is a common choice [Sch97, Ohe01, Sch06]. However, even when the Hoare derivations are syntactic, it is nearly ubiquitous in mechanically verified proofs for the assertion language to be purely semantic; that is, assertions are identified with predicates on program states (perhaps together with other auxiliary data) [Nip02]. In these applications, indirection helps solve problems of indirect reference, especially if one wishes to embed semantic assertions (or types) into program syntax or operational semantics.

One could instead use a *fully* syntactic approach, where assertions are also treated as syntax. Using syntactic assertions removes the need to solve tricky contravariant domain equations. Instead, one is left with the problem of adequately representing the syntax and semantics of the assertion logic, a nontrivial problem in its own right. A logic supporting higher-order quantification is especially troublesome to handle in most mechanized proof systems.

***Domain theory.*** Domain theory is an alternate approach to creating semantic models of programming languages [GHK$^+$03]. Domain theory is primarily concerned with solving *recursive domain equations* that describe the desired form of the model. Using domain theory, one can construct actual isomorphism solutions to equations such as pseudoequation (2). Indeed, one can construct models using domain theory for applications where indirection theory does not appear well suited (*e.g.* denotational models of the untyped $\lambda$-calculus). The price one pays is that the solution is constructed in some category of domains rather than in simple sets.

Working in higher categories is especially inconvenient in mechanized higher-order logics. Simply building up the required theory takes quite a bit of effort. Many attempts have been made to mechanize domain theory [Mil72, MNOS99, Age06], including a recent effort in Coq by Benton *et al.* [BKV09]. However, all these systems seem to stop short of developing a full suite of domain theory. For example, we are aware of no mechanization which develops as far as the theory of algebraic bounded-complete partial orders (*i.e.* Scott domains). Benton *et al.* have developed one of only very few mechanizations that includes an inverse-limit construction, allowing the solution of recursive domain equations. Furthermore, once the required base theory is developed, the tedium of verifying the properties of structured morphisms (*e.g*, monotonicity, continuity, *etc.*) can still be a major hurdle. Much of the development effort in HOLCF [MNOS99] was devoted to alleviating this pain, and Benton *et al.* claim that "dealing with all the structural morphisms is still awkward" in their system [BKV09].

***Ultrametric Spaces.*** Birkedal *et al.* solve recursive domain equations using certain classes of ultrametric spaces [BST09]. Their approach is similar in spirit to domain theory, where semantic domains are built in some higher category, but uses a foundation built on metric spaces rather than partially ordered sets. Given a sufficient basis of analysis and category theory, the metric space setup seems to afford a nice simplification of similar domain-theoretic results. To our knowledge, no attempt has yet been made to mechanize this metric space approach.

***BI Hyperdoctrines.*** BI hyperdoctrines are category-theoretic models of higher-order bunched implications [BBTS07]. BI hyperdoctrines provide: higher-order logic, the standard BI connectives, recursive definitions, and impredicative quantification. But the published BI-hyperdoctrine model does not appear to support *indirection*, *i.e.* the kinds of settings in §2. The techniques used for domain-theoretic models of indirection (*e.g.* Day's construction in functor categories [Lev02]) could perhaps be applied to BI hyperdoctrines. Like the domain-theoretic methods, methods based on advanced category theory require a great deal of background in the field to understand and utilize. Mechanizing models based on BI hyperdoctrines would require considerable effort to build up the necessary base theory.

***Very Modal Model.*** Appel *et al.*'s "Very Modal Model" (VMM) provides a way to simplify the handling of step-indexing models by building a modal logic for reasoning about the indexes [AMRV07].

In our current paper we also simplify the application of step-indexed models, so the techniques are superficially similar. However, although indirection theory was designed to dovetail with the VMM, they are in fact orthogonal and either system can be used without the other. After Appel *et al.*, both Benton and Tabareau [BT09] and Ahmed *et al.* [ADR09] constructed modal models without indirection theory. Indeed, it is possible and useful to build a modal logic even when the setting does not have the contravariant circularities that motivate the current work [BT09, DAB09]. Moreover, although we build a modal logic on top of indirection theory in §6, one can easily work directly with the axioms.

In addition, the logic constructed in §6 is superior to the one from Appel *et al.* because it properly recognizes and handles the heredity condition. The naïve interpretation of implication found there does not preserve heredity. This leads to less elegant reasoning because the operator $\Box$ must be inserted in nonobvious places. Neither do Appel *et al.* show how to incorporate the substructural elements required for separation logic, which we do here.

Finally, the VMM suffers from the same flaw as other step-indexing models: the construction is hardcoded to a particular domain and is difficult to adapt to other settings. Indirection theory is simpler, more powerful, and applies to a wide variety of domains.

***Concurrent separation logic.*** Both Hobor *et al.* [HAZ08] and Gotsman *et al.* [GBC+07] developed formulations of a Concurrent Separation Logic with first-class locks and threads. Both presentations use a semantic model of propositions; however, they take very different paths to handle the contravariant circularity involved in modeling lock invariants.

Hobor *et al.* use a purely semantic method that grew out of the VMM, whereas Gotsman *et al.* use a syntactic reification of the logic to handle the contravariance issue. That is, Gotsman *et al.* associate each occurrence of a concurrent statement (*e.g.*, `lock`, `unlock`, etc.) with a static handle (a "lock sort") pointing into a lookup table of resource invariants. This technique is displeasing from a mathematical perspective: it is preferable to have a semantic model that is independent of syntax. Significantly, the syntactic reification weakens Gotsman *et al.*'s model: for example, there is no polymorphism over lock sorts, so it is impossible to specify the simple function that takes an arbitrary lock and locks it:

$$\begin{array}{ll} \mathit{Spec}: & \forall x, \pi, R.\ f : \{x \overset{\pi}{\rightsquigarrow} R\}\{R * x \overset{\pi}{\rightsquigarrow} R\} \\ \mathit{Body}: & f(x) = (\texttt{lock}\ x). \end{array}$$

Instead, Gotsman *et al.* would have to create one version of this function for each lock sort and then somehow determine which one to call at a given program point. In contrast, indirection theory can handle this case easily with impredicative quantification over $R$.

Finally, the models of Hobor *et al.* and Gotsman *et al.* are difficult to understand. Hobor subsequently made some improvements in mathematical modularity but the model was still much more complex than the theory of indirection presented here [Hob08]. We hope that the formulation of indirection theory as a section-retraction pair (squash-unsquash) makes life easier for the reader.

## 13. Conclusion

Indirection theory is a new approach to modeling impredicative indirection. It is immediately applicable to a wide variety of settings and has an extremely simple, categorical axiomatization. Indirection theory embeds easily into type theory, and its axiomatization embeds easily into the simple theory of types (HOL). Finally, it is powerful: we have used indirection theory as the basis for a machine-checked soundness proof for the Concurrent Separation Logic for Concurrent C minor described in §2.7.

***Acknowledgments.*** We would like to thank Amal Ahmed, Nick Benton, Matthew Parkinson, Andrew Pitts, Peter Sewell, Konrad Slind, and the anonymous referees for their helpful suggestions on earlier drafts of this paper.

## References

[AAR+09] Amal Ahmed, Andrew W. Appel, Christopher D. Richards, Kedar N. Swadi, Gang Tan, and Daniel C. Wang. Semantic foundations for typed assembly languages. *ACM. Trans. on Programming Languages and Systems*, 2009.

[AAV03] Amal Ahmed, Andrew W. Appel, and Roberto Virga. An indexed model of impredicative polymorphism and mutable references. http://www.cs.princeton.edu/~appel/papers/impred.pdf, January 2003.

[AB07] Andrew W. Appel and Sandrine Blazy. Separation logic for small-step C minor. In *20th Int'l Conference on Theorem Proving in Higher-Order Logics (TPHOLs)*, pages 5–21, 2007.

[ADR09] Amal Ahmed, Derek Dreyer, and Andreas Rossberg. State-dependent representation independence. In *POPL '09: Proc. 36th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 340–353, 2009.

[AFM05] Amal Ahmed, Matthew Fluet, and Greg Morrisett. A step-indexed model of substructural state. In *ICFP '05: Proceedings of the tenth ACM SIGPLAN International Conference on Functional programming*, pages 78–91, 2005.

[Age06] Sten Agerholm. Domain theory in HOL. In *Higher Order Logic Theorem Proving and Its Applications*, LNCS, pages 295–309. Springer, 2006.

[Ahm04] Amal J. Ahmed. *Semantics of Types for Mutable State*. PhD thesis, Princeton University, Princeton, NJ, November 2004. Tech Report TR-713-04.

[AMRV07] Andrew W. Appel, Paul-Andre Melliès, Christopher D. Richards, and Jerôme Vouillon. A very modal model of a modern, major, general type system. In *Proc. 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'07)*, pages 109–122, January 2007.

[BBTS07] Bodil Biering, Lars Birkedal, and Noah Torp-Smith. Bi-hyperdoctrines, higher-order separation logic, and abstraction. *ACM Trans. Program. Lang. Syst.*, 29(5):24, 2007.

[BKV09] Nick Benton, Andrew Kennedy, and Carsten Varming. Some domain theory and denotational semantics in coq. In *Theorem Proving in Hogher Order Logics*, LNCS, pages 115–130. Springer, 2009.

[BST09] Lars Birkedal, Kristian Støvring, and Jacob Thamsborg. The category-theoretic solution of recursive metric-space equations. Technical report, 2009. TR-2009-119.

[BT09] Nick Benton and Nicolas Tabareau. Compiling functional types to relational specifications for low level imperative code. In *TLDI '09: Proc. 4th international workshop on Types in language design and implementation*, pages 3–14, 2009.

[CO78] Robert Cartwright and Derek Oppen. Unrestricted procedure calls in Hoare's logic. In *POPL '78: Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of Programming Languages*, pages 131–140, 1978.

[COY07] Cristiano Calcagno, Peter W. O'Hearn, and Hongseok Yang. Local action and abstract separation logic. In *LICS '07: Proceedings of the 22nd Annual IEEE Symposium on Logic in Computer Science*, pages 366–378, 2007.

[Cra03] Karl Crary. Toward a foundational typed assembly language. In *POPL '03: 30th ACM Symp. on Principles of Programming Languages*, pages 198–212, 2003.

[DAB09] Derek Dreyer, Amal Ahmed, and Lars Birkedal. Logical step-indexed logical relations. In *Proceedings 24th Annual IEEE Symposium on Logic in Computer Science (LICS'09)*, 2009.

[DAH08] Robert Dockins, Andrew W. Appel, and Aquinas Hobor. Multi-modal separation logic for reasoning about operational semantics. In *24th Conference on the Mathematical Foundations of Programming Semantics*, pages 5–20. Springer ENTCS, 2008.

[DHA09] Robert Dockins, Aquinas Hobor, and Andrew W. Appel. A fresh look at separation algebras and share accounting. In *The 7th Asian Symposium on Programming Languages and Systems*. Springer ENTCS, 2009. To appear.

[DNRB10] Derek Dreyer, Georg Neis, Andreas Rossberg, and Lars Birkedal. A relational modal logic for higher-order stateful adts. In *POPL '10: Proc. 37th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2010. To appear.

[GBC+07] Alexey Gotsman, Josh Berdine, Byron Cook, Noam Rinetzky, and Mooly Sagiv. Local reasoning for storable locks and threads. In *Proceedings 5th Asian Symposium on Programming Languages and Systems (APLAS'07)*, 2007.

[GHK+03] G. Gierz, K. H. Hofmann, K. Kiemel, J. D. Lawson, M. Mislove, and D. S. Scott. *Continuous Lattices and Domains*, volume 93 of *Encylopedia of Mathematics and its Applications*. Cambridge University Press, Cambridge, UK, 2003.

[Har94] Robert Harper. A simplified account of polymorphic references. *Information Processing Letters*, 51:201–206, 1994.

[HAZ08] Aquinas Hobor, Andrew W. Appel, and Francesco Zappa Nardelli. Oracle semantics for concurrent separation logic. In *Proc. European Symp. on Programming (ESOP 2008) (LNCS 4960)*, pages 353–367. Springer, 2008.

[Hob08] Aquinas Hobor. *Oracle Semanatics*. PhD thesis, Princeton University, Princeton, NJ, November 2008.

[HS97] Robert Harper and Chris Stone. A type-theoretic interpretation of Standard ML. http://foxnet.cs.cmu.edu/papers/rwh-interpret.ps, 1997.

[HS08] Cătălin Hriţcu and Jan Schwinghammer. A step-indexed semantics of imperative objects. International Workshop on Foundations of Object-Oriented Languages (FOOL'08), 2008.

[Ler06] Xavier Leroy. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *POPL'06*, pages 42–54, 2006.

[Lev02] Paul Blain Levy. Possible world semantics for general storage in call-by-value. In *Computer Science Logic, 16th International Workshop, CSL 2002*, volume 2471 of *LNCS*, pages 232–246, Edinburgh, Scotland, UK, September 2002. Springer.

[Mil72] Robin Milner. Logic for computable functions: description of a machine implementation. Technical report, Stanford, CA, USA, 1972.

[MNOS99] Olaf Müller, Tobias Nipkow, David von Oheimb, and Oskar Slotosch. HOLCF = HOL + LCF. *Journal of Functional Programming*, 9:191–223, 1999.

[Nip02] Tobias Nipkow. Hoare logics for recursive procedures and unbounded nondeterminism. In *Computer Science Logic*, volume 2471/2002 of *LNCS*, pages 155–182. Springer, 2002.

[O'H07] Peter W. O'Hearn. Resources, concurrency and local reasoning. *Theoretical Computer Science*, 375(1):271–307, May 2007.

[Ohe01] David von Oheimb. *Analyzing Java in Isabelle/HOL: Formalization, Type Safety and Hoare Logic*. PhD thesis, Technische Universität München, 2001.

[PBC06] Matthew Parkinson, Richard Bornat, and Cristiano Calcagno. Variables as resource in Hoare logics. *Proc. of 21st Annual IEEE Symp. on Logic in Computer Science*, 0:137–146, 2006.

[Pym02] David J. Pym. *The Semantics and Proof Theory of the Logic of Bunched Implications*, volume 26 of *Applied Logic Series*. Kluwer Academic Publishers, 2002.

[Res00] Greg Restall. *An Introduction to Substructural Logics*. Routledge, London, England, 2000.

[Sch97] Thomas Schreiber. Auxiliary variables and recursive procedures. In *TAPSOFT '97: Proceedings of the 7th International Joint Conference CAAP/FASE on Theory and Practice of Software Development*, pages 697–711, London, UK, 1997. Springer-Verlag.

[Sch06] Norbert Schirmer. *Verification of Sequential Imperative Programs in Isabelle/HOL*. PhD thesis, Technische Universität München, 2006.

[WF94] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.

# A. Axiomatization in Coq

```
Module Type TY_FUNCTOR.
  Parameter F : Type → Type.
  Parameter fmap : forall {A B}, (A → B) → F A → F B.

  Axiom fmap_id : forall A : Type , fmap (@id A) = @id (F A).
  Axiom fmap_comp : forall A B C (f:B → C) (g:A → B),
       compose (fmap f) (fmap g) = fmap (compose f g).
  Parameter T : Type.
  Parameter T_bot : T.
  Parameter other : Type.
End TY_FUNCTOR.

Module Type KNOT.
  Declare Module TF:TY_FUNCTOR.
  Import TF.

  Parameter knot : Type.
  Definition predicate := (knot * other) → T.

  Parameter squash : (nat * F predicate) → knot.
  Parameter unsquash : knot → (nat * F predicate).

  Definition level (x:knot) : nat := fst (unsquash x).
  (* le_gt_dec n m : forall n m : nat, {n <= m} + {n > m} *)
  Definition approx (n:nat) (p:predicate) : predicate :=
    fun w ⇒ if le_gt_dec n (level (fst w)) then T_bot else p w.

  Axiom squash_unsquash :
    forall x, squash (unsquash x) = x.
  Axiom unsquash_squash :
    forall n x′, unsquash (squash (n,x′)) = (n,fmap (approx n) x′).
End KNOT.

Module Knot (TF:TY_FUNCTOR) : KNOT with Module TF:=TF.
```