Constructing Hereditary Worlds Within Worlds

Robert Dockins[1] and Aquinas Hobor[2]

[1] Portland State University rdockins@cecs.pdx.edu
[2] National University of Singapore hobor@comp.nus.edu.sg

Abstract. Although they appear unrelated, the type system of the polymorphic λ-calculus with references and the assertions of concurrent separation logic with first-class locks share a critical feature: an unsound contravariant circularity in their naïve semantic model. We developed indirection theory to automatically construct, and cleanly axiomatize, step-indexed approximations to these naïve models, as well as a large number of others [HDA10b].

Unfortunately, the previous axiomatization had a flaw. One is usually only interested in using *hereditary* predicates: those which are closed under the action of approximation. As previously presented, indirection theory allows nonhereditary predicates to exist in certain parts of the construction. Although not fatal, this flaw requires workarounds that are not entirely obvious to the uninitiated.

We correct this flaw by presenting a new axiomatization of indirection theory that only permits heredity predicates and show that the new interface is sound by constructing a model. The new axiomatization is somewhat more subtle than the previous one, but it retains the same flavor, cleanliness, and metatheoretic properties. In contrast, the new construction is markedly more complex, especially in a mechanized context. Indeed, our Coq mechanization is one of our key contributions, and accordingly we present it in considerable detail.

1 Introduction

In the last decade or so, step-indexing has been a popular method for building semantic domains in which to do programming language research. Step-indexing models have been used to build type systems for core ML [AAV03], program logics for von Neumann machines [AMRV07], models of object references [HS08], type systems for uniqueness types [AFM05], concurrent separation logic [HAZ08], and logics for reasoning about representation independence [ADR09], among many others.

The naïve semantic models for each of these domains contains an unpleasant contravariant circularity whose interpretation in SET leads to a contradiction. Step-indexing allows one to **approximate** this circularity while remaining in SET. Moreover, type systems and program logics built on a step-indexed model can use both impredicative quantification and a contravariant-compatible recursion operator μ, making them considerably more expressive and modular than systems lacking these features.

In previous work we showed that most existing uses of step-indexing fall into a particular pattern, and proposed *indirection theory*, which automatically builds step-indexed models for any domain that falls into that pattern [HDA10b]. Beyond a pushbutton construction, indirection theory's major virtue is a very clean, categorical[3] axioma-

[3] Categorical in the sense that all models of the axioms are isomorphic.

tization, which precisely captures the nature of the underlying approximation. Furthermore, the simple models produced by indirection theory scale up to industrial-strength languages, as evidenced by the Verified Software Toolchain [App11], which uses indirection theory to build a sophisticated program logic for the Clight programming language, a large subset of C used by the Compcert project [Ler09].

However, the indirection theory previously presented was deficient in one important way: the predicates—*e.g.*, assertions of the logic or types in the type system—appearing inside the circular construction do not need to respect the approximation forced by the step indexing, *i.e.*, they do not need to be *hereditary*. This lack causes annoying artifacts—primarily, additional side conditions and extra modal operators—to appear in definitions and proofs built on top of the original version of the theory.

Here we correct this deficiency by showing how to axiomatize and construct indirection theory with hereditary predicates. The axioms of our new version of indirection theory are slightly more complicated than the original, but retain the same conciseness and equational form. The construction of the model, on the other hand, is significantly more difficult and requires some subtle manipulation of dependent types.

The original construction of indirection theory is given in considerable mathematical (paper) detail in [HDA10b, §8]. The real difficulty is on paper, *i.e.*, an experienced (but by no means expert) Coq user should be able to take the given exposition and mechanize the result.[4] In contrast, the construction we will give here, while only moderately more difficult to perform on paper, was vastly more difficult to mechanize. Although sophisticated, the resulting Coq proof development is only about 700 lines, allowing us to present the lion's share directly in this paper. The complete proof script, a stand-alone version of the same construction found in our Mechanized Semantic Library [ADH12], may be found on the accompanying website.[5]

Structure of presentation. In §2 we will introduce indirection theory in general, covering our original version of indirection theory in §2.1 and discussing our new version of indirection theory in §2.2. The heart of our result, the new construction itself, is given in §3. Finally, in §4 we show that indirection theory 2.0 enjoys the same pleasant metatheoretical properties as the original.

Coqery: This paper explains the details of a Coq proof development — when a line number appears in a listing, it refers to that exact line in the development. A deeply interested reader may therefore follow along directly from the source files of the proof. Coq code without line numbers is simply part of the presentation and is not included in the development. To fully grasp the mechanized constructions, the reader will require a basic knowledge of how to read Coq definitions. For readers familiar generally with higher-order logics, but not Coq specifically, some notes about Coq syntax (and our sugaring conventions) have been collected into Appendix A.

Because the development is particularly complex, we will use "Coqery" sections to explain the unusual and/or advanced techniques we were forced to use to make it all hang together.

We have extended Coq's base logic with the axioms of functional and propositional extensionality, which we use freely throughout this paper.

[4] Only one place (equation 55) is likely to cause some difficulty, due to a needed coercion between dependent types. In §3 we will see how this headache is handled via `eq_rect`.

[5] http://msl.cs.princeton.edu/fancyknots

```
Module Type TY_FUNCTOR.
  Parameter F : Type → Type.
  Parameter fmap : ∀ A B, (A → B) → F A → F B.
  Implicit Arguments fmap [A B].

  Axiom fmap_id : ∀ A, fmap (id A) = id (F A).
  Axiom fmap_comp  : ∀ A B C (f : B → C) (g : A → B),
    fmap f ∘ fmap g = fmap (f ∘ g).

  Parameter other : Type.
End TY_FUNCTOR.
```

Listing 1. Input module type

2 Indirection Theory

In previous work, we showed that many uses of step-indexing fall into a pattern [HDA10b]:

$$K \;\approx\; F((K \times O) \to \mathbb{T}) \tag{1}$$

Here, K is the object we wish to model, F is a **covariant** functor, O is some additional "flat data", and $\mathbb{T}$ is the set of truth values, *i.e.*, $\top$ and $\bot$ (`Prop` in Coq). For example, consider the polymorphic λ-calculus with references. In this well-studied setting [Ahm04], the challenge is to define an approximation to the following (unsound) semantic model:

type	$\stackrel{?}{\equiv}$	$(\mathsf{memtype} \times \mathsf{value}) \to \mathbb{T}$	metatype of types
$\mathsf{memtype}$	$\stackrel{?}{\equiv}$	$\mathsf{address} \rightharpoonup \mathsf{type}$	metatype of memory typings

The unsoundness comes from the contravariant cycle in the definitions, which forces a cardinality inversion (*i.e.*, $|\mathsf{memtype}| < |\mathsf{memtype}|$). Notice, however, that this attempt **does** fall into the pattern of equation (1) by setting

$$F(X) \;\equiv\; \mathsf{address} \rightharpoonup X \qquad \text{and} \qquad O \;\equiv\; \mathsf{value}$$

With this assignment, the desired object K, which in general we call the *knot*, is then equivalent to $\mathsf{memtype}$. Likewise, the function space from $(K \times O)$ to $\mathbb{T}$, which in general we call the space of *predicates* $\mathbb{P}$, is then equivalent to type.

Informally, F can be thought of as a data structure with holes into which we can put arbitrary types; ultimately we want to fill those holes with predicates. Formally, a covariant functor is a function F from types to types equipped with a mapping function fmap that sends functions $f : A \to B$ to functions $\mathsf{fmap}\ f : F(A) \to F(B)$. Informally, $\mathsf{fmap}\ f$ finds the As inside $F(A)$ and applies f to them, yielding an $F(B)$. This boils down to requiring that the function fmap have two properties: first, $\mathsf{fmap}\ (\lambda x.x) = \lambda x.x$, *i.e.*, fmap of the identity function is the identity function; and second, $(\mathsf{fmap}\ f) \circ (\mathsf{fmap}\ g) = \mathsf{fmap}(f \circ g)$, *i.e.*, fmap respects functional composition.

In Coq we use the module type `TY_FUNCTOR`, given in listing 1, to capture the precise input we require; note that we have sugared functional composition as $\circ$ and that `id X` is the identity function on the type `X`. The `Implicit Arguments` directive tells Coq to automatically infer parameters at call sites; this allows us to write, e.g., `fmap f`, without having to explicitly give the type parameters.

For relatively simple domains it is almost trivial for a client to build a module matching the signature TY_FUNCTOR. For example, here is the appropriate instantiation for the semantic model of types for the polymorphic λ-calculus with references:

```
Definition address : Type := ···
Definition value : Type := ···

Module LamCalc_TyFunc <: TY_FUNCTOR.
  Definition F (X : Type) : Type := address → option X.
  Definition fmap (A B : Type) (f : A → B) (fA : F A) : F B :=
    fun addr ⇒ match fA addr with
     | None ⇒ None
     | Some x ⇒ Some (f x)
    end.
  Implicit Arguments fmap [A B].

  Lemma fmap_id: ∀ A, fmap (id A) = id (F A).
  Proof. ··· Qed.  (* 2 proof lines elided *)

  Lemma fmap_comp: ∀ A B C (f : B → C) (g : A → B),
    fmap f ∘ fmap g = fmap (f ∘ g).
  Proof. ··· Qed.  (* 2 proof lines elided *)

  Definition other : Type := value.
End LamCalc_TyFunc.
```

One of the key selling points of indirection theory is that a client often need give only a handful of lines to receive a powerful and complex semantic model in return.

2.1 Indirection Theory 1.0

As explained, an attempt to find an exact solution to pseudoequation (1) in SET leads to a contradiction via cardinality inversion. The point of indirection theory [HDA10b] is instead to develop an approximation by constructing an object K satisfying:

$$K \;\preceq\; \mathbb{N} \times F((K \times O) \to \mathbb{T})$$

Here the $\mathbb{N}$ component indicates how much "information" is in the knot K (*i.e.*, the number of stratification levels). By "$\preceq$", we mean that K and $F((K \times O) \to \mathbb{T})$ are related by a pair of related functions, squash and $\mathsf{unsquash}$. The squash function takes a data structure with predicates in it (an F predicate) and turns it into a knot. Conversely, the $\mathsf{unsquash}$ function unpacks a knot and returns its level of approximation and an F predicate. Because the desired domain equation (1) does not admit a solution in SET, squash and $\mathsf{unsquash}$ are not inverses. Instead, indirection theory gives us the weaker fact that $(\mathsf{unsquash}, \mathsf{squash})$ form a section-retraction pair:

$$\begin{array}{lcl} (\mathsf{squash} \circ \mathsf{unsquash})(k) & = & k \\ (\mathsf{unsquash} \circ \mathsf{squash})(n, f) & = & (n, \mathsf{fmap}\ \mathsf{approx}_n\ f) \end{array}$$

That is, $\mathsf{squash} \circ \mathsf{unsquash}$ is the identity and $\mathsf{unsquash} \circ \mathsf{squash}$ generates an approximation to the original by applying the function approx_n to its predicates via fmap.

To define the "predicate approximater" function approx_n, first recall that the natural number n indicates how much information is in the knot k. For a given k we can access this number, called the *level* of k and written $|k|$, by first $\mathsf{unsquashing}$ k and then taking the first projection, *i.e.*, $|k| \equiv (\pi_1\ (\mathsf{unsquash}\ k))$. Given this definition, it is simple

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

  Parameter knot : Type.

  Definition predicate := (knot × other) → Prop.

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

  Definition level (k:knot) := fst (unsquash k).

  Definition approx (n:nat) (p:predicate) : predicate :=
     fun w ⇒ level (fst w) < n ∧ p w.

  Axiom squash_unsquash : ∀ (k : knot),
    squash (unsquash k) = k.
  Axiom unsquash_squash : ∀ (n : nat) (f : F predicate),
    unsquash (squash (n,f)) = (n,fmap (approx n) f).
End KNOT.

Module Knot(TF':TY_FUNCTOR):KNOT with Module TF:=TF'.
   (* About 400 lines to construct Indirection Theory v1.0 elided.
      See [knot.v, ADH12] and [HDA10b, section 8]. *)
End Knot.

(* Instantiating the construction with our example TY_FUNCTOR LamCalc_TyFunc *)
Module K := Knot(LamCalc_TyFunc). Import K.

(* Supplemental definitions, after we have instantiated the construction. *)
Definition knot_age1 (k:knot) : option knot :=
  match unsquash k with
  | (O,_) ⇒ None
  | (S n,x) ⇒ Some (squash (n,x))
  end.

Definition age k k' : Prop := age1 k = Some k'.

Definition hereditary (P : predicate) : Prop := ∀ k k' o,
  age k k' → P (k, o) → P (k', o).
```

Listing 2. Indirection Theory 1.0

to define the approximation function approx_n, which "cuts down" a predicate P by throwing away all behavior of P on knots of level $\geq n$, as follows:

$$\mathsf{approx}_n(P) \;:\; \mathsf{predicate} \quad \equiv \quad \lambda k. \begin{cases} P(k) & \text{when } |k| \;<\; n \\ \bot & \text{when } |k| \;\geq\; n \end{cases}$$

The straightforward axiomatization of indirection theory in Coq is given by the interface `KNOT` from listing 2. Although brief, it is categorical (*i.e.*, it completely describes the underlying model), so a client need never look behind the module boundary.

Constructing and instantiating indirection theory 1.0. After the module type `KNOT`, listing 2 contains the specification of the module functor `Knot`, which is responsible for the original construction. The construction is parameterized, and thus independent of the particular choice of `TY_FUNCTOR`. The original mechanization itself is elided, but follows the presentation in [HDA10b, §8] closely (except around equation 55, which requires `eq_rect`); interested readers should consult the Coq script [ADH12, `knot.v`].

Just after the elided construction, we show the single line that instantiates it. We say indirection theory is "pushbutton" because clients need only build a `TY_FUNCTOR` (*e.g.*, `LamCalc_TyFunc`) and then include a single line to construct the model.

Consequences of indirection theory. One of the corollaries of these axioms is that if we $\mathsf{unsquash}$ a knot k to (n, f), then $f = \mathsf{fmap}\ \mathsf{approx}_n\ f$, *i.e.*, for all predicates P that are stored in f, $P = \mathsf{approx}_n(P)$. Because $\mathsf{approx}_n(P)$ "throws out" all the behavior of P for knots of level greater than or equal to n, including k itself, ***P* cannot say anything meaningful about the knot whence it came**. This is not a coincidence: in fact, this is exactly where we weaken pseudoequation (1) to achieve a sound definition.

On the other hand, what is the point of storing a predicate inside a knot if you cannot use it? Suppose you have extracted some predicate P from a knot k. The trick is that instead of applying P to the original knot k, you instead apply P to a very similar, albeit slightly simpler, knot k', found by $\mathsf{unsquashing}$ k and then $\mathsf{resquashing}$ the result back to level $n - 1$. The resulting knot, k', is almost the same at k, except that each predicate inside k' has now been approximated to level $n - 1$ instead of to level n. For historical reasons we call this process *aging the knot*; note that since the naturals are well-founded any concrete knot k of level n can only be aged n times. It is convenient to write this relationship as a mathematical relation $k \rightsquigarrow k'$, defined as follows:

$$k \rightsquigarrow k' \quad \equiv \quad \mathsf{let}\ (n, f) = \mathsf{unsquash}(k)\ \ \mathsf{in}\ \ (n > 1)\ \wedge\ k' = \mathsf{squash}(n - 1, f)$$

Unfortunately, this trick leads to a further problem. Intuitively, aging the knot is a regrettable technicality forced by the underlying model, and should not change whether a predicate is true or not. For example, if in the context of memory typing k a value v has type $\mathsf{ref}\ \tau$, then after k is aged to k' we expect v to still have type $\mathsf{ref}\ \tau$. In other words, we want our predicates to be stable (or monotonic) under the action of approximation. Sadly, it is relatively simple to construct predicates that are not so well behaved:

$$P_{bad}(k, o) \quad = \quad \begin{cases} \top & \text{when } |k| \geq 5 \\ \bot & \text{when } |k| < 5 \end{cases}$$

We call predicates that **do** have this desirable property *hereditary*, defined as follows:

$$\mathsf{hereditary}(P) \quad \equiv \quad \forall k, k', o.\ k \rightsquigarrow k'\ \rightarrow\ P(k, o)\ \rightarrow\ P(k', o)$$

In fact, proofs utilizing step-indexed models only want to use hereditary predicates, but historically the burden of maintaining and tracking the hereditariness of the predicates under consideration has been a major technical hassle in the formal proofs.

The straightforward Coq definitions for aging and hereditariness finish up listing 2.

Further reading. We have now given the original interface to indirection theory and explained some associated key ideas—the impossibility of a predicate judging the knot that contains it, the resulting necessity of aging the knot, and the consequential need for hereditary predicates—so that we can motivate the improved interface presented in §2.2. Unfortunately, due to space limitations we cannot discuss any further aspects

of using indirection theory here. In addition to the original paper on indirection theory [HDA10b], we refer readers to work that has used indirection theory to prove interesting program logics sound, including a clean worked example of the polymorphic λ-calculus with references [HD10, §2] that is included with our Mechanized Semantic Library [ADH12]; concurrent separation logic with first-class locks [Hob08]; time bounds for general function pointers [DH12]; and the Verified Software Toolchain [App11]. Moreover, interested readers can consult [HDA10a] for how we build a general multimodal logic that can mix approximation and separation (including models for the standard logical connectives in §2.4 and for our contravariant-capable μ in §4).

2.2 Indirection Theory 2.0

Our new version of indirection theory is the continuation of a line of work to better manage the hereditariness side conditions of predicates. Initial step-indexing work handled these side conditions in a very ad-hoc way [Ahm04]. Appel *et al.* made the first systematic progress by showing that these side conditions could be tracked by using a modal operator, which they called *necessarily* and wrote $\Box$, in numerous places within the key definitions [AMRV07]. Unfortunately, it was all-to-easy to overlook a place where $\Box$ was needed, and it was not always immediately apparent why a $\Box$ was required to those not already familiar with the issues. We then showed that by utilizing intuitionistic models of certain operators, such as implication $\Rightarrow$ and the "magic wand" $-\!\!*$ of separation logic, some of these $\Box$ modal operators could be safely removed [HDA10b], but their proper placement was still tricky.

Our next observation was that once the knot and predicates had been defined, we could "package up" a predicate with its proof of hereditariness [HDA10a], by defining

$$\mathsf{hered\_pred} \ : \ \mathsf{type} \quad \equiv \quad \{\ P \in \mathsf{predicate} \mid \mathsf{hereditary}(P)\ \}$$

In Coq such a package is done with a dependent type as follows:

```
Record hered_pred : Type :=
  mk_hPred
  { app_hPred : knot × other → Prop
  ; hpredHered : ∀ k k' o,
      age k k' → appPred (k,o) → appPred (k', o) }.
```

Here `Record` defines both a new type, `hered_pred`, as well as `mk_hPred` for its constructor and `app_hPred` and `hpredHered` for projection functions. The vast majority of the proof then uses `hered_pred` instead of `predicate`, which reduces the burden of the hereditariness side conditions considerably.

Unfortunately, this technique does not remove the burden entirely. In particular, the knot itself uses the original predicate. When we are storing a hered_pred P into a knot, this is not so bad—we just forget that P is hereditary. However, on the way out the situation is worse; if we pull P out of a knot, then we must somehow remember that P was hereditary previously. As always in life, forgetting is easy; remembering is hard.

The obvious thing to try is to make the predicates in the knot itself always be hereditary, *i.e.* be a package of a function and a hereditariness proof. This is precisely what our new version of indirection theory achieves. To make this work, there are two thorny

```
81 Module Type KNOT_HERED.
82   Declare Module TF:TY_FUNCTOR. Import TF.
83
84   Parameter knot : Type.
85   Parameter age1 : knot → option knot.
86
87   Definition age (k k':knot) : Prop := age1 k = Some k'.
88
89   Record predicate : Type :=
90     mkPred
91     { appPred : knot × other → Prop
92     ; predHered : ∀ k k' o,
93         age k k' → appPred (k,o) → appPred (k',o) }.
94
95   Parameter squash : (nat × F predicate) → knot.
96   Parameter unsquash : knot → (nat × F predicate).
97
98   Parameter approx : nat → predicate → predicate.
99
100  Axiom squash_unsquash : ∀ (k:knot),
101    squash (unsquash k) = k.
102  Axiom unsquash_squash : ∀ (n:nat) (f:F predicate),
103    unsquash (squash (n,f)) = (n, fmap (approx n) f).
104
105  Definition level (k:knot) : nat := fst (unsquash k).
106
107  Axiom approx_spec :
108    ∀ (n:nat) (p:predicate) (k:knot) (o:other),
109      appPred (approx n p) (k,o) ↔
110      (level k < n ∧ appPred p (k,o)).
111
112  Axiom knot_age1 : ∀ (k:knot),
113    age1 k =
114    match unsquash k with
115    | (0,_) ⇒ None
116    | (S n,x) ⇒ Some (squash (n,x))
117    end.
118 End KNOT_HERED.
```

Listing 3. Indirection Theory 2.0

problems to solve. The first is that "hereditary predicates" are defined above via "predicate" and "hereditary"; which are in turn defined via "age," "squash," "unsquash," and the "knot," which we now wish to define in terms of "hereditary predicates." In other words, our interface itself now contains a cycle!

To avoid this cycle, we make the definition of aging opaque. In the construction, it will be defined in an alternative, noncircular manner—and then we will *prove* that the hidden concrete definition is equivalent to the definition we want clients to use. This idea may be more easily grasped by examining the Coq interface `KNOT_HERED` in Listing 3. Notice that `age1` (line 85) is now declared as an opaque `Parameter`, allowing a noncircular axiomatization. Clients will use `knot_age1` (lines 112–117) to "unfold" `age1` into the definition they expect. You may notice that approx (line 98) is also defined as an opaque constant with an unfolding axiom (lines 107–110); however, in this case it is merely a convenience. We could instead give approx as a direct definition, at the expense of some minor complication of the interface.

The remainder of the interface is exactly as one should expect given the discussion above. As compared to the original in Listing 2, `predicates` are **always** hereditary, but other than that the two axiomatizations are extremely similar, even to the point of

taking the same input module TY_FUNCTOR. This is by design, since it allows for rapid "upgrades" of existing proof developments and guarantees that indirection theory 2.0 enjoys the same clean design as did the original. Most importantly, with this upgrade the previously significant and ad-hoc management of hereditariness side conditions is completely handled by the type system of the theorem prover; there is never a need for the modal operator $\Box$ or any other ad-hoc technique.

With the first thorny problem solved, we are ready to proceed to the second: showing that the new axiomatization is sound by actually constructing a model.

3 Construction

The construction is a module functor which, when given an input TY_FUNCTOR produces a module with type KNOT_HERED. The overall proof strategy is vaguely similar to the one we presented previously [HDA10b, §8], but is modified to account for the additional complexity posed by hereditary predicates.

The construction is best understood as being divided into three distinct phases. In the first phase, we define the basic concepts of "stratified predicates", "knots" containing stratified predicates, and the process of "aging" a knot. In the second phase, we show how the "stratified predicates" are related to ordinary "hereditary predicates". This second phase revolves around two functions, strat and unstrat, which translate between these two different forms. In the final phase, we finish up by defining the remaining elements of the public interface and proving their properties.

Phase 1. Although it only takes a few lines to define the knot type, those lines turn out to be a sophisticated exercise in dependent types. Listing 4 contains the details.

We want to define a type of "stratified predicates," which are a stack of increasingly-more-accurate predicates. The sprod type function adds one new "level" to a stack. The first component of sprod is the "tail" of the stack and the second component of sprod is a predicate that judges pairs of type $F(A) \times \mathsf{other}$. The second component of sprod is the payload—these predicates are the things we actually care about. A level 0 stack is just the unit type, a level 1 stack is $\mathsf{sprod}(\mathsf{unit})$, a level 2 stack is $\mathsf{sprod}(\mathsf{sprod}(\mathsf{unit}))$, etc. We make a stratified predicate more approximate by throwing away the outermost layer of the stack—*i.e.*, by taking the first projection of the sprod pair.

So far, this is all just the same as the standard construction for indirection theory [HDA10b, §8]. The twist comes in when we want to require that all the predicates in a stack be stable under approximation. This turns out to be quite tricky to define. We want to construct some type, together with a a predicate over that type, where the type itself references the predicate. In this case, we can manage to get the definition we want because both the predicate and the type are stratified into a tower of constructions indexed by $\mathbb{N}$. To do this, we define the type we desire simultaneously with the predicate on that type by recursion on a natural number (the level of the type). This gives us a stratified family of types together with a stratified family of predicates over those types.

The need to carefully track the type of all these constructions makes the formal definitions a little difficult to parse, so first we give the ideas in more familiar set-theoretic notation. The main task is to build up a family of types sinv_n and a family of predicates

hered_n, each indexed by a natural number n and defined via mutual recursion. In the following, π_1 is the first projection from a pair and π_2 is the second projection.

$$
\begin{array}{ll}
\mathsf{sinv}_0 & \equiv \mathsf{unit} \\
\mathsf{sinv}_{n+1} & \equiv \{\ X \in \mathsf{sinv}_n \times \mathcal{P}(F(\mathsf{sinv}_n) \times \mathsf{other})\ \mid\ \mathsf{hered}_n(X)\ \}
\end{array}
$$

$$
\begin{array}{ll}
\mathsf{hered}_0(X) & \equiv \top \\
\mathsf{hered}_{n+1}(X) & \equiv \forall k \in F(\mathsf{sinv}_{n+1}).\ \forall o \in \mathsf{other}. \\
& \qquad (k, o) \in \pi_2(X) \rightarrow (\mathsf{fmap}\ \pi_1\ k, o) \in \pi_2(\pi_1(X))
\end{array}
$$

The main idea is that hered_n requires that judgements about a (k, o) pair made at level $n + 1$ remain true if k and the predicate are each approximated one level. Because hered_n holds at each sinv level, judgements remain true for all levels of approximation.

Coqery: These definitions are not straightforward to mechanize, even in a system, such as Coq, that admits dependent types. Coq usually allows mutually-recursive definitions as long as an argument is structurally decreasing, as is the case with the natural n here. However, Coq does not allow mutually-recursive definitions when one component appears in the ***type*** *of another, which is exactly what is going on: observe that the type of* hered_n *depends on* sinv_n*! Accordingly, in the mechanized proof we must rearrange this construction by manually "tupling up" the mutually recursive definitions via dependent records and define them simultaneously.*

Formally (see listing 4), we represent the tupled-up result type with guppy_ty, which is a dependent record containing a type and a predicate on sprods built using that type; the constructor is `mkGuppy` and the projections are `sinv_n` and `hered_n`.

The main definition that simultaneously defines the stratified family of types and their predicates is guppy, which is defined by recursion on a natural number (the name is suggestive of Grand Unified Package). In the base case, the type constructed is just unit, the type with a unique inhabitant, and the predicate is the totally permissive predicate; this simultaneously defines the base cases for sinv and hered as above.

In the case for $n + 1$, the type we build is defined by guppy_step_ty. Given a previous guppy_ty, guppy_step_ty constructs a new dependent record consisting of an sprod and a proof that it satisfies the hered_n predicate defined at the previous level. This corresponds to the $n + 1$ case of the sinv definition.

Finally, we come to guppy_step_prop, which calculates the $n + 1$-level predicate on level n sprods. The property we require is that the judgments made by level $n + 1$ predicates over level $n + 1$ data structures remain true when we approximate the predicate by one level (by taking first projections) and simultaneously approximate the predicates appearing in the sprod structure (by mapping first projections over the entire structure). This corresponds to the $n + 1$ case for hered.

We are almost done. We define sinv_n via projection from the dependent record, and knot as dependent pairs of a natural number (the level), and the sinv type at that level.

The definition of age1 shows how to take a knot and produce a knot of lower level, provided we started with non-0 level to begin with. Essentially, we just have to map first projections over each sinv within the datastructure. The fact that age1 is defined by direct reference to the internal structure of knot explains why it is an opaque parameter in the axiomatization. Finally, we define age and predicate — these definitions are copied verbatim from the axiomatization.

```
Definition sprod A := A × ((F A × other) → Prop).
Record guppy_ty : Type :=
  mkGuppy { sinv_n : Type; hered_n : sprod sinv_n → Prop }.

Record guppy_step_ty (Z:guppy_ty) : Type :=
  mkGuppyStep { gupT : sprod (sinv_n Z); gupPrf : hered_n Z gupT }.
Implicit Arguments gupT [[Z]].

Definition guppy_step_prop (Z:guppy_ty)
  : sprod (guppy_step_ty Z) → Prop :=
  fun X ⇒ ∀ (k:F (guppy_step_ty Z)) (o:other),
    snd X (k,o) →
    snd (gupT (fst X)) (fmap (fst ∘ gupT) k,o).

Fixpoint guppy (n:nat) : guppy_ty :=
  match n with
  | 0   ⇒ mkGuppy unit (fun _ ⇒ True)
  | S m ⇒ mkGuppy (guppy_step_ty (guppy m))
                  (guppy_step_prop (guppy m))
  end.

Definition sinv (n:nat) : Type := sinv_n (guppy n).

Record knot' : Type :=
  mkKnot { klevel : nat; kstruct : F (sinv klevel) }.
Definition knot := knot'.   (* Module system misfeature *)

Definition age1 (k:knot) : option (knot) :=
  match k with
  | mkKnot 0 _ ⇒ None
  | mkKnot (S m) f ⇒ Some (mkKnot m (fmap (fst ∘ gupT) f))
  end.

Definition age (k1 k2:knot) : Prop := age1 k1 = Some k2.

Record predicate : Type :=
  mkPred
  { appPred : knot × other → Prop
  ; predHered : ∀ k k' o,
      age k k' → appPred (k,o) → appPred (k',o) }.
```

Listing 4. Knot Construction

Coqery: Note that the definitions appearing in listing 4 are more granular than seems to be strictly required. In particular, the fixpoint definition guppy *could be given all at once by using Σ-types and inlining the various preceding definitions. However, the breakdown listed here serves two purposes. First, the more granular definitions are somewhat easier for humans to examine and parse. However, it also seems to be important for the computer! When we first attempted to build the definition of* guppy *as one monolithic definition, the Coq typechecking kernel displayed very poor performance while checking later definitions and proofs (*i.e.*, it took hours and then ran out of memory on Qed steps). We hypothesize that the monolithic recursive definition was somehow causing the typechecker to unroll the definition more than strictly required, leading to repeated, expensive and unnecessary unifications. Introducing the intermediate definitions produced dramatic improvements in runtime. We have been unable to replicate this behavior on recent versions of Coq, leading us to believe that the behavior is dependent on some subtle internal quirk of the typechecking algorithm which has subsequently been changed.*

Phase 2. Now that we have constructed the sinv and predicate, we need to build methods to coerce between them. After all, the whole idea of the sinv construction was to be an approximate version of predicates. First we define an operation stratify that takes a predicate and produces an approximated sinv; see listing 5.

```
178  Section stratifies.
179    Variable Q:knot × other → Prop.
180    Variable HQ: ∀ k k' o, age k k' → Q (k,o) → Q (k',o).
181
182    Fixpoint stratifies (n:nat) : sinv n → Prop :=
183    match n as n' return sinv n' → Prop with
184    | 0 ⇒ fun _ ⇒ True
185    | S m ⇒ fun (p:sinv (S m)) ⇒
186          stratifies m (fst (gupT p)) ∧
187          ∀ (k:F (sinv m)) (o:other),
188            snd (gupT p) (k,o) ↔ Q (mkKnot m k,o)
189    end.
190
191    Lemma stratifies_unique : ∀ n p1 p2,
192      stratifies n p1 → stratifies n p2 → p1 = p2.
193    Proof. ··· Qed.  (* 11 proof lines elided *)
205
206    Definition stratify : ∀ n:nat, Σ (x:sinv n), stratifies n x.
207    Proof. ··· Qed.  (* 15 proof lines elided *)
222
223  End stratifies.
224
225  Definition strat (n:nat) (p:predicate) : sinv n :=
226    projT1 (stratify (appPred p) (predHered p) n).
```

Listing 5. Stratification

The essence of the stratification we want to compute is given by the mathematical definition below. Here, the symbol $*$ refers to the unique inhabitant of the unit type.

$$\begin{aligned}\mathsf{stratify}_0(Q) &\equiv * \\ \mathsf{stratify}_{n+1}(Q) &\equiv (\mathsf{stratify}_n(Q),\ \lambda(z,o).\ Q((n,z),o))\end{aligned}$$

Each embedded predicate satisfies hered_n because Q is hereditary.

Unfortunately, just like construction of the knot in Listing 4, defining this function in Coq is not as straightforward as one might hope. To produce the sinv we desire from a predicate Q, we must demonstrate that each approximate predicate is closed under aging **as we are constructing it**. To use the fact that Q is hereditary, we must know that the stacked predicate gives the same answers as Q on knots of the appropriate level. Thus, the definition of stratify constructs the stacked predicate we want while at the same time showing that it satisfies the correctness predicate stratifies, which captures what it means for a stacked predicate to properly correspond to a hereditary predicate.

Coqery: The `Section` keyword gives each definition between lines 178 and 223 the additional parameters specified by the `Variable` declarations on lines 179–180. When one such definition references another, the parameters are passed appropriately. Outside of the section the additional parameters must be specified explicitly again, as in line 226, in which e.g. *"`(appPred p)`" is being passed for `Q` and "`(predHered p)`" is being passed for `HQ` the `stratify` function.*

Lemma `stratifies_unique` lets us know that stratifies uniquely specifies a stacked sinv.[6] This proof goes via an easy induction over n.

Coqery: To actually define the stratification function, we use a somewhat unusual technique that allows us to build a definition by using proof tactics rather than by explicitly writing down the term. This works because of how the Coq tactical proof system operates — applications of proof

[6] Here, and throughout this paper, we will elide proof scripts.

```
Fixpoint floor (m:nat) (n:nat) : sinv (m+n) → sinv n :=
  match m as m' return sinv (m'+n) → sinv n with
  | O ⇒ fun p ⇒ p
  | S m' ⇒ fun p ⇒ floor m' n (fst (gupT p))
  end.

Lemma decompose_nat : ∀ (m n:nat),
  (Σ (r:nat), n = (r + S m)) ⊕ ( m ≥ n ).
Proof. ··· Qed.  (* 8 proof lines elided *)

Definition unstratify (n:nat) (p:sinv n) := fun w ⇒
  match w with (mkKnot m z,o) ⇒
    match decompose_nat m n with
    | inl (existT r Hr) ⇒
           snd (gupT (floor r (S m)
             (eq_rect n sinv p (r + S m) Hr))) (z,o)
    | inr H ⇒ False
    end
  end.

Lemma floor_shuffle:
  ∀ (m n : nat) (p : sinv (m + S n))
    (H : (m + S n) = (S m + n)),
    floor (S m) n (eq_rect (m + S n) sinv p (S m + n) H) =
    fst (gupT (floor m (S n) p)).
Proof. ··· Qed.  (* 20 proof lines elided *)

Lemma unstratify_hered : ∀ n p,
  ∀ k k' o, age k k' →
    (unstratify n p) (k,o) → (unstratify n p) (k',o).
Proof. ··· Qed.  (* 36 proof lines elided *)

Definition unstrat (n:nat) (p:sinv n) : predicate :=
  mkPred (unstratify n p) (unstratify_hered n p).
```

Listing 6. Unstratification

tactics actually build up a proof term in the language which is later typechecked by the core. However, the tactic system can be used to build things that are not usually considered proofs.

The Σ-type appearing in the type of stratify *is quite similar to the existential quantifier $\exists$. However, it might be more properly read "one can construct," because it indicates that a constructive witness may be found. Thus, we can read the type of* stratify *as: "for every n, one can construct an x of type* sinv_n *such that* stratifies $n\ x$*." The definition of* stratify *also goes by induction on n. We do not actually need to know what term is produced by the tactical definition — this is because we proved that the resulting* sinv *satisfies* stratifies $n\ x$*, and we know that there is a* unique *such x. In other words, all the properties we care about are captured by the specification* stratifies*. Therefore we end the definition using* `Qed`*, which produces an opaque proof term.*

Finally, we define the function strat as the first projection of stratify.

In the reverse direction, we need to "unstratify" a sinv and get back a predicate. The main idea here is to transform an sinv into a predicate. Thus, we need to be able to apply the sinv to an arbitrary knot $\times$ other pair. To do this, we unpack the knot to look at its level. If the level is greater or equal to n, the sinv doesn't have enough information to judge the knot and we must therefore reject it. However, if the level of the knot is less than the level of the sinv, we can unwind the sinv until their levels match, and then apply the predicate contained in the sinv.

In mathematical notation, what we want to do is define a floor function $\lfloor x \rfloor_n$ that unwinds an sinv n times. This unwinding function is used to define unstratify by cases.

$$\begin{aligned}
\lfloor X \rfloor_0 &\equiv X \\
\lfloor X \rfloor_{n+1} &\equiv \lfloor \pi_1\ X \rfloor_n \\
\mathsf{unstratify}_n(X) &\equiv \left\{ ((m,z),o) \;\middle|\; \begin{array}{ll} (z,o) \in \pi_2 \lfloor X \rfloor_r & \text{when } n = r + (1+m) \\ \bot & \text{when } m \geq n \end{array} \right\}
\end{aligned}$$

The formal definition for unstratification appears in listing 6. The function floor unwinds an sinv to a specific level. Getting the typechecker to accept the definition requires a dependent pattern match, but the actual content of the function is quite simple; apply first projections until the desired level is reached.

Coqery: Several of the definitions and lemma statements in the remainder of the listing are complicated by the need to insert equality coercions. This issue arises because we need to convert a value between two types that we know are provably, but not definitionally, equal. The typechecking core of Coq is willing two types as equal if they are "convertible," that is, if one can be rewritten into the other via the reduction rules of the base calculus. Convertibility captures the notion that is usually called "definitional equality." However, when working with dependent types, it sometimes happens that one wishes to work with types that are only provably equal. For example, if n is a natural number variable, it is possible to prove that $n + 1 = 1 + n$ by a simple induction. However, these two terms are not convertible, so the typechecker will not automatically, for example, treat a term of type sinv $(n + 1)$ *as a term of type* sinv $(1 + n)$.

Instead, we must use the term eq_rect *to coerce a term of one type to a provably equal type. For example, in the definition* unstratify, *we use* eq_rect *to coerce* p : sinv n *into type* sinv $(r + S\ m)$. *In general,* (eq_rect $x\ f\ t\ y\ H$) *takes a term* t *of type* $(f\ x)$ *and produces a term of type* $(f\ y)$, *provided* H *is a proof of* $x = y$.

Technically, eq_rect *is the* `Type`*-sorted eliminator for the equality type and is defined in terms of dependent pattern matching. Nonetheless, is useful to think of* eq_rect *as a special coercion, such as one might find in a calculus with explicit subtyping. The odd argument order stems from the fact that Coq automatically generates* eq_rect *from the inductive definition of equality.*

The `floor_shuffle` lemma lets us rearrange how sinvs get unwound. The intellectual content of the lemma says "if we unwind $m + 1$ times, we get the same result as if we unwind $1 + m$ times." However, like with unstratify, the statement is complicated by the need to insert an equality coercion. Once properly stated, the lemma goes by a fairly straightforward induction on m.

The main lemma in this section is `unstratify_hered`. It shows that whenever you unstratify an sinv, the result is a hereditary predicate. The proof goes by case analysis on the definition of unstratify; the only interesting case uses the `floor_shuffle` lemma. Finally, unstrat packages together the definition of unstratify with the proof that it is hereditary to make a predicate.

The next thing we need to do is relate the properties of stratification and unstratification together. The results of listing 7 show that composing strat and unstrat in one way results in the identity function, whereas the other way is equal to approx.

```
Lemma stratifies_unstratify_more :
  ∀ (n m1 m2:nat) (p1:sinv (m1+n)) (p2:sinv (m2+n)),
    floor m1 n p1 = floor m2 n p2 →
    (stratifies (unstratify (m1+n) p1) n (floor m1 n p1) →
     stratifies (unstratify (m2+n) p2) n (floor m2 n p2)).
Proof. ··· Qed.  (* 42 proof lines elided *)

Lemma stratify_unstratify : ∀ n p H,
  projT1 (stratify (unstratify n p) H n) = p.
Proof. ··· Qed.  (* 28 proof lines elided *)

Lemma strat_unstrat : ∀ n, strat n ∘ unstrat n = id (sinv n).
Proof. ··· Qed.  (* 7 proof lines elided *)

Definition approx_def (n:nat) (p:predicate) :=
  fun w ⇒ klevel (fst w) < n ∧ appPred p w.

Lemma approx_hered : ∀ n p k k' o,
  age k k' → approx_def n p (k,o) → approx_def n p (k',o).
Proof. ··· Qed.  (* 9 proof lines elided *)

Definition approx (n:nat) (p:predicate) : predicate :=
  mkPred (approx_def n p) (approx_hered n p).

Lemma unstratify_Q : ∀ n (p:sinv n) (Q:knot × other → Prop),
  stratifies Q n p →
  ∀ (k:knot) o, klevel k < n →
    (unstratify n p (k,o) ↔ Q (k,o)).
Proof. ··· Qed.  (* 23 proof lines elided *)

Lemma unstrat_strat : ∀ n, unstrat n ∘ strat n = approx n.
Proof. ··· Qed.  (* 29 proof lines elided *)

Lemma strat_unstrat_Sn : ∀ n,
  fst ∘ gupT = strat n ∘ unstrat (S n).
Proof. ··· Qed.  (* 57 proof lines elided *)
```

Listing 7. Properties of `strat` and `unstrat`

In mathematical notation, the main results are:

$$\mathsf{approx}_n(Q) \;\equiv\; \{\, ((m,z),o) \in Q \mid m < n \,\}$$

$$\mathsf{strat}_n \circ \mathsf{unstrat}_n = \mathsf{id}$$
$$\mathsf{unstrat}_n \circ \mathsf{strat}_n = \mathsf{approx}_n$$
$$\mathsf{strat}_n \circ \mathsf{unstrat}_{n+1} = \pi_1$$

Listing 7 contains the formal proof sketch. The first lemma is a technical lemma needed to prove `stratify_unstratify`; its proof goes by induction on n and several applications of `floor_shuffle`. To prove `stratify_unstratify`, we use the fact that stratified predicates are unique. We then just have to show that both p and $(\mathsf{stratify}\ (\mathsf{unstratify}\ n\ p)\ H\ n)$ are stratifications of $(\mathsf{unstratify}\ n\ p)$. The second case was proved as we defined stratify, so the case for p is the only interesting one. It goes by induction on the level, using lemma `stratifies_unstratify_more`. Then `strat_unstrat` repackages the result as an identity.

Next we define approximation and show it is a hereditary predicate. This proof is easy, as it simply relies on the fact that aging reduces the level and the hereditary property of the given predicate p.

```
Definition squash (x:nat × F predicate) : knot :=
  match x with (n,f) ⇒ mkKnot n (fmap (strat n) f) end.

Definition unsquash (k:knot) : nat × F predicate :=
  match k with mkKnot n f ⇒ (n, fmap (unstrat n) f) end.

Lemma squash_unsquash : ∀ k, squash (unsquash k) = k.
Proof. ··· Qed.  (* 9 proof lines elided *)

Lemma unsquash_squash : ∀ n f,
  unsquash (squash (n,f)) = (n, fmap (approx n) f).
Proof. ··· Qed.  (* 8 proof lines elided *)

Lemma knot_age1 : ∀ k,
  age1 k =
  match unsquash k with
  | (0,_) ⇒ None
  | (S n,x) ⇒ Some (squash (n,x))
  end.
Proof. ··· Qed.  (* 44 proof lines elided *)

Definition level (x:knot) : nat := fst (unsquash x).

Lemma approx_spec : ∀ n p (k:knot) (o:other),
  appPred (approx n p) (k,o) ↔
  (level k < n ∧ appPred p (k,o)).
Proof. ··· Qed.  (* 5 proof lines elided *)
```

Listing 8. `squash` and `unsquash`

The lemma `unstratify_Q` turns out to be an easy result that follows from a straightforward manipulation of the definitions. Likewise `unstrat_strat` is a relatively easy proof that requires only a few arithmetic facts and `unstratify_Q`.

The final result in this phase, `strat_unstrat_Sn`, shows what happens when unstratification at level $n+1$ is immediately followed by stratification at level n — the result is the same as simply taking the first projection of the stacked sinv. This result is the key lemma that allows us to prove the specification of knot_age1 in the next phase.

Phase 3. At this point, we have all the tools we need to define the main components of the output module type, squash and unsquash, and to prove their properties.

$$\begin{aligned} \mathsf{squash}(n,\, f) &\equiv (n,\, \mathsf{fmap}\ \mathsf{strat}_{\mathsf{n}}\ f) \\ \mathsf{unsquash}(n,\, f) &\equiv (n,\, \mathsf{fmap}\ \mathsf{unstrat}_{\mathsf{n}}\ f) \end{aligned}$$

The two key axioms about squash and unsquash follow immediately from the functor properties of fmap and the lemmas about strat and unstrat from the previous phase.

Once these results are obtained, the only proof of any substance remaining is the proof that knot_age1 satisfies its specification. This proof follows via a straightforward manipulation of the definitions and an application of the `strat_unstrat_Sn` lemma. We finish up by proving that approx has satisfies its stated identity; this proof consists of little more than unfolding the definition. With that, we have completed the entire implementation of the `KNOT_HERED` module interface. The formal definitions and lemmas are found in listing 8.

4 Corollaries and Uniqueness

As with the 1.0 version of indirection theory, a number of simple and useful corollaries follow immediately from the axiomatization. First, we know that unsquash is an injective function, and that unsquash $\circ$ squash is idempotent. Both facts follow directly from axiom `squash_unsquash`.

We also know that when a knot is unsquashed, the contained datastructure has already had its predicates approximated to the level of the knot.

$$\mathsf{unsquash}\ k = (n, f) \ \rightarrow \ f = \mathsf{fmap}\ \mathsf{approx}_n\ f$$

Simply from the definition, we know that approximating multiple times is equal to approximating once at the smaller index.

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

Further, to show that two squashed knots are equal, it suffices to show that the contained datastructures are equivalent up to approximation.

$$\mathsf{fmap}\ \mathsf{approx}_n\ f = \mathsf{fmap}\ \mathsf{approx}_n\ g \ \rightarrow \ \mathsf{squash}(n, f) = \mathsf{squash}(n, g)$$

Just like version 1.0, our new indirection theory also supports building a separation algebra over knots, which allows the construction of separation logics using indirection theory. See [ADH12, `knot_hered_sa.v`] for details.

Finally, and most significantly, we can show that the axiomatization from §2.2 is *categorical* in the sense that all models of the axioms are isomorphic. We do this by constructing, when given any two implementations satisfying the axioms, a function f, sending knots of one implementation to knots of the other, and its inverse g such that f and g respect the the operations on knots — namely squash and unsquash. The module type in listing 9 formalizes this notion.

We will not cover the implementation of this module type here,[7] but merely give a sketch of the proof. Roughly, the coercions work by unsquashing a knot of one type into the common form $\mathbb{N} \times F(\mathsf{predicate})$, and then squashing into the other sort of knot. Unfortunately, this simple idea does not quite work, because the type predicate, like knot, is specific to each implementation. Therefore, we need a way to send the predicates of one implementation to the predicates of the other. The functions fPred and gPred from the above module type perform this coercion. Note that fPred is defined using g and vice versa.

Fortunately, we can make the whole construction hang together because the knots are themselves built up via stratification. The predicates inside each knot are simpler than the knot itself. Therefore, we can build up f and g as successive approximations. For each $n \in \mathbb{N}$, there is an f_n and a g_n that have the desired properties when applied to knots of level less than n. To build f, we simply consult the level of its argument and then apply the appropriate f_n. The construction of f and g are intertwined — at each stage of the construction, f_{n+1} is defined by reference to g_n and vice versa.

[7] Interested readers may consult the formal proof scripts.

```
Definition map_pair {A B C D} (f:A → B) (g:C → D) (x:A × C) : B × D :=
  (f (fst x), g (snd x)).

Module Type ISOMORPHIC_KNOTS.
  Declare Module TF : TY_FUNCTOR.
  Declare Module K1 : KNOT_HERED with Module TF := TF.
  Declare Module K2 : KNOT_HERED with Module TF := TF.
  Import TF.

  Parameter f : K1.knot → K2.knot.
  Parameter g : K2.knot → K1.knot.

  Definition fP (p:K1.predicate) : K2.knot × other → Prop :=
    K1.appPred p ∘ map_pair g (@id other).
  Definition gP (p:K2.predicate) : K1.knot × other → Prop :=
    K2.appPred p ∘ map_pair f (@id other).

  Axiom fP_hered : ∀ p,
    ∀ k k' o, K2.age k k' → fP p (k,o) → fP p (k',o).
  Axiom gP_hered : ∀ p,
    ∀ k k' o, K1.age k k' → gP p (k,o) → gP p (k',o).

  Definition fPred (p:K1.predicate) : K2.predicate :=
    K2.mkPred (fP p) (fP_hered p).
  Definition gPred (p:K2.predicate) : K1.predicate :=
    K1.mkPred (gP p) (gP_hered p).

  Definition f' : (nat × F K1.predicate) → (nat × F K2.predicate) :=
    map_pair (@id nat) (fmap fPred).
  Definition g' : (nat × F K2.predicate) → (nat × F K1.predicate) :=
    map_pair (@id nat) (fmap gPred).

  Axiom iso1 : f ∘ g = id K2.knot.
  Axiom iso2 : g ∘ f = id K1.knot.

  Axiom f_squash : ∀ F1,
    f (K1.squash F1) = K2.squash (f' F1).
  Axiom g_squash : ∀ F2,
    g (K2.squash F2) = K1.squash (g' F2).
  Axiom f_unsquash : ∀ k1,
    f' (K1.unsquash k1) = K2.unsquash (f k1).
  Axiom g_unsquash : ∀ k2,
    g' (K2.unsquash k2) = K1.unsquash (g k2).
End ISOMORPHIC_KNOTS.
```

Listing 9. Uniqueness of knots (from `knot_unique.v`)

In the following, squash_1 represents the squash function of the first implementation, $\mathsf{unsquash}_2$ represents the unsquash function of the second implementation, etc. Then the informal definitions of these functions go as follows:

$$
\begin{aligned}
f_0 &\equiv \mathsf{squash}_2 \circ \mathsf{fmap}\ (\lambda P.\ \emptyset) \circ \mathsf{unsquash}_1 \\
f_{n+1} &\equiv \mathsf{squash}_2 \circ \mathsf{fmap}\ (\lambda P.\ \{(k,o) \mid (g_n\ k, o) \in P\}) \circ \mathsf{unsquash}_1
\end{aligned}
$$

$$
\begin{aligned}
g_0 &\equiv \mathsf{squash}_1 \circ \mathsf{fmap}\ (\lambda P.\ \emptyset) \circ \mathsf{unsquash}_2 \\
g_{n+1} &\equiv \mathsf{squash}_1 \circ \mathsf{fmap}\ (\lambda P.\ \{(k,o) \mid (f_n\ k, o) \in P\}) \circ \mathsf{unsquash}_2
\end{aligned}
$$

$$
\begin{aligned}
f(k) &\equiv f_{\mathsf{level}_1(k)+1}(k) \\
g(k) &\equiv g_{\mathsf{level}_2(k)+1}(k)
\end{aligned}
$$

In the end, we get functions f and g that apply to knots of any level. It is then a relatively straightforward exercise to show that these functions are inverses and that they commute properly with the operations of the axiomatizations.

5 Related work

Version 1.0 of indirection theory was presented previously by Hobor et al. [HDA10b] — this work is a direct continuation. Like the previous version of indirection theory, this version is designed to dovetail with semantic techniques for building program logics and type systems [AMRV07, HDA10a] and specifically with techniques for building separation logics [DHA09]. However, nothing prevents the theory from being used in a stand-alone setting. Note that while the construction from this paper appears in the MSL [ADH12], there are some minor differences, as discussed in Appendix B.

The general idea of step-indexing is by now rather well-known; here we list just a few of the papers that use step-indexing techniques [AAV03, HS08, AFM05, HAZ08, ADR09, DAB09, DNRB10].

An alternative approach to using indirection theory is to directly solve the desired domain equation in a category of ultrametric spaces [BSS10]. The solution thus obtained is then an isomorphism of nonexpansive functions between certain kinds of metric spaces. Birkedal et al. demonstrate how to use these kinds of solutions to reason about a capability calculus and also compare their metric space approach to indirection theory 1.0 [BRS$^+$11]. Benton et al. [BBKV10] have shown how to mechanize (in Coq) the core concepts of the ultrametric space approach and use them to reason about a simple λ-calculus.

6 Conclusion

In this paper we have presented an improved axiomatization of indirection theory, and proved the axiomatization sound and categorical. In addition, we have examined the formal Coq proof of these facts in some detail, explaining some of the unusual techniques we were forced to adopt to complete the construction.

References

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.

ADH12. Andrew Appel, Robert Dockins, and Aquinas Hobor. Mechanized Semantic Library. Available at http://msl.cs.princeton.edu, 2009–2012.

ADR09. Amal Ahmed, Derek Dreyer, and Andreas Rossberg. State-dependent representation independence. In *Proc. ACM Symposium on Principles of Programming Languages (POPL)*, pages 340–353, 2009.

AFM05. Amal Ahmed, Matthew Fluet, and Greg Morrisett. A step-indexed model of substructural state. In *Proc. ACM International Conference on Functional programming (ICFP)*, pages 78–91, 2005.

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. ACM Symposium on Principles of Programming Languages (POPL)*, pages 109–122, 2007.

App11. Andrew W. Appel. Verified software toolchain. In *Proc. European Symposium on Programming Languages and Systems (ESOP)*, volume 6602 of *LNCS*, pages 1–17. Springer, 2011.

BBKV10. Nick Benton, Lars Birkedal, Andrew Kennedy, and Carsten Varming. Formalizing domains, ultrametric spaces and semantics of programming languages. Submitted for publication, 2010.

BRS+11. Lars Birkedal, Bernhard Reus, Jan Schwinghammer, Kristian Storving, Jacob Thamsborg, and Hongsoek Yang. Step-indexed kripke models over recursive worlds. In *Proc. ACM Symposium on Principles of Programming Languages (POPL)*, 2011.

BSS10. Lars Birkedal, Jan Schwinghammer, and Kristian Storving. The catehgor-theoretic solution of recursive metric-space equations. *Theoretical Computer Science*, 411:4102–4122, 2010.

DAB09. Derek Dreyer, Amal Ahmed, and Lars Birkedal. Logical step-indexed logical relations. In *Proc. IEEE Symposium on Logic in Computer Science (LICS)*, 2009.

DH12. Robert Dockins and Aquinas Hobor. Verifying time bounds for general function pointers. In *Mathematical Foundations of Programming Semantics (MFPS XXVIII)*, pages 132–148. Springer ENTCS, June 2012.

DHA09. Robert Dockins, Aquinas Hobor, and Andrew W. Appel. A fresh look at separation algebras and share accounting. In *Proc. Asian Symposium on Programming Languages and Systems (APLAS)*, volume 5904 of *LNCS*, pages 161–177. Springer, 2009.

DNRB10. Derek Dreyer, Georg Neis, Andreas Rossberg, and Lars Birkedal. A relational modal logic for higher-order stateful adts. In *Proc. ACM Symposium on Principles of Programming Languages (POPL)*, pages 185–198, 2010.

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.

HD10. Aquinas Hobor and Robert Dockins. Developing and mechanizing semantic models for program logics, November–December 2010. http://www.comp.nus.edu.sg/~hobor/Publications/2010/aplas10_examples_tutorial.pdf.

HDA10a. Aquinas Hobor, Robert Dockins, and Andrew W. Appel. A logical mix of approximation and separation. In *The 8th Asian Symposium on Programming Languages and Systems*, pages 439–454. Springer ENTCS, 2010.

HDA10b. Aquinas Hobor, Robert Dockins, and Andrew W. Appel. A theory of indirection via approximation. In *Proc. ACM Symposium on Principles of Programming Languages (POPL)*, pages 171–185, January 2010.

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

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

Ler09. Xavier Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4):363–446, 2009.

A Coq Notation

In type theory it is conventional to present functions in "Curried" form (with multiple distinct arguments rather than as single tupled argument), and to use juxtaposition for function application. Thus, while an application of a two-argument function is traditionally written as $f(x, y)$, in type theory one writes instead $f\ x\ y$. Likewise, the type

of f is traditionally given as $A \times B \to C$, whereas the type-theory convention is to instead write $A \to B \to C$, which should be parsed $A \to (B \to C)$.

The symbol $\forall$ is everywhere used to denote the dependent function space. It is uniformly used for parametric polymorphism (types depending on types), dependent types (types depending on "data") and for universal quantification (propositions depending on types or data). Quantified variables may either appear with a specific type ascription (as `(f : B →C)`) or without, in which case Coq infers the appropriate type. The arrow symbol $\to$ is likewise used uniformly for both the ordinary (nondependent) function space and as the logical operator for implication. The keyword **`Type`** is the builtin type of types, and **`Prop`** is the builtin type of logical propositions. The `Record` keyword declares a dependent record and the appropriate associated functions (*e.g.*, constructors and projections). The symbol $\circ$ refers to standard function composition. The symbol $\oplus$ refers to disjoint union type. The `Implicit Arguments` directive tells Coq to automatically infer parameters at call sites.

In module types, the `Axiom` and `Parameter` keywords are used to indicate opaque constants — when a client imports an implementation of the module type, he will know the type of these items, but not their definitions.

B Comparison with MSL version

The development presented here closely follows the one present in the Mechanized Semantic Library [ADH12]. Readers who are interested in actually using indirection theory in Coq are encouraged to use the MSL, which has additional support for a variety of interesting features. However, the reader should be aware that the development presented here differs in some minor respects from the MSL version. These differences are largely related to integrating the knot construction with other parts of the library.

In the input module type of the MSL version, the function fmap and the facts regarding it are packaged together into a record — this eases construction of specific functor instances. Otherwise, the input module is the same. Likewise, the output module refers to an auxiliary concept, ageable, which is a record generalizing age1 and level and which asserts certain facts about their relationship. In the MSL, the concept of hereditary predicates is defined with respect to these ageable structures. Thus hereditary predicates with respect to knot $\times$ other pairs are just one sort of hereditary predicate. This engineering decision allows large-scale proof reuse in various different applications. In addition, the MSL version of hereditary predicates are defined as an anonymous Σ-type because this facilitates the use of the `Program Definition` extension. This extralogical extension makes it easier to separate the definition of hereditary predicates from their proof obligations. We also make use of Coq's implicit coercion system to provide nicer syntax for applying hereditary predicates.

All these auxiliary records have been "flattened" in the presentation we prepared for this paper in order to simplify the exposition and to avoid dragging in any of the rest of the MSL. In a similar way, the construction of §3 differs slightly in the MSL version — mostly in inconsequential ways. The main visible difference is a preference for anonymous Σ-types in the MSL version over the named dependent record types presented here. Additionally, some care was taken with the proof prepared for this paper to ensure that nice notation is used everywhere so that the typographical layout is pleasing. No such care has been taken with the MSL version.