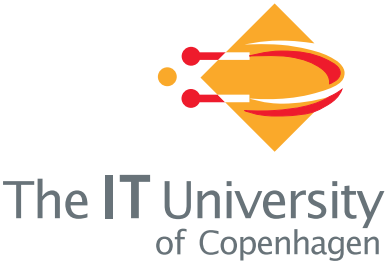


A Realizability Model for Impredicative Hoare Type Theory

Rasmus Lerchedahl Petersen& Lars Birkedal
IT University of Copenhagen, {rusmus | birkedal}@itu.dk
Aleksandar Nanevski& Greg Morrisett
Harvard University, {aleks | greg}@eecs.harvard.edu

IT University Technical Report Series **TR-2007-XX**

ISSN 1600–6100 **09 2007**

Rasmus Lerchedahl Petersen& Lars Birkedal
IT University of Copenhagen, {**rusmus** | **birkedal**}**@itu.dk**
Aleksandar Nanevski& Greg Morrisett
Harvard University, {**aleks** | **greg**}**@eecs.harvard.edu**

ISSN 1600–6100

ISBN ISBN-NO

Copies may be obtained by contacting:

IT University of Copenhagen
Rued Langgaards Vej 7,
DK-2300 København S
Denmark

Telephone: **+45 72 18 50 00**
Telefax: **+45 72 18 50 01**
Web `www.itu.dk`

A Realizability Model for Impredicative Hoare Type Theory

Rasmus Lerchedahl Petersen& Lars Birkedal
IT University of Copenhagen, {rusmus | birkedal}@itu.dk
Aleksandar Nanevski& Greg Morrisett
Harvard University, {aleks | greg}@eecs.harvard.edu

Abstract

We present a denotational model of impredicative Hoare Type Theory, a very expressive dependent type theory in which one can specify and reason about mutable abstract data types.

The model ensures soundness of the extension of Hoare Type Theory with impredicative polymorphism; makes the connections to separation logic clear, and provides a basis for investigation of further sound extensions of the theory, in particular equations between computations and types.

1 Introduction

Dependent types provide a powerful form of specification for higher-order, functional languages. For example, using dependency, one can specify the signature of an array subscript operation as $\texttt{sub} : \forall\alpha.\Pi x{:}\alpha\,\texttt{array}.\Pi y{:}\{i{:}\texttt{nat} \mid i < x.\texttt{size}\}.\alpha$, where the type of the third argument, y, refines the underlying type `nat` using a predicate that ensures that y is a valid index for the array x.

Dependent types have long been used in the development of formal mathematics, but their use in practical programming languages has proven challenging. One of the main reasons is that the presence of any computational effects, including non-termination, exceptions, access to store, or I/O – all of which are indispensable in practical programming – can quickly render a dependent type system unsound.

The problem can be addressed by severely restricting dependencies to only effect-free terms (as in for instance DML [25]). But the goal of our work is to try to realize the full power of dependent types for specification of effectful programs. To that end, we have been developing the foundations of a language that we call *Hoare Type Theory* or HTT [18, 17], which we intend to be an expressive and explicitly annotated internal language, providing a semantic framework for elaborating more practical external languages.

HTT starts with a pure, dependently typed core language and augments it with an indexed monadic type of the form $\{P\}x{:}A\{Q\}$. This type encapsulates and describes effectful computations that may diverge or access a mutable store. The type can be read as a Hoare-like partial correctness specification, asserting that if the computation is run in a heap satisfying the pre-condition P, then if it terminates, it will return a value x of type A and leave a heap described by Q. Through Hoare types, the system can enforce soundness in the presence of effects. The Hoare type admits small footprints as in separation logic [21, 19], where the pre- and postconditions only describe the part of the store that the program actually uses; the unspecified part is automatically assumed invariant.

The most distinguishing feature of HTT in comparison with other recent proposals for Hoare and separation logics for higher-order languages [5, 15, 26, 16] is that specifications in HTT are *integrated with types*. In Hoare logic, it is not possible to abstract over specifications in the source programs, aggregate the logical

invariants of the data structures with the data itself, compute with such invariants, or nest the specifications into larger specifications or types. These features are essential ingredients for data abstraction and information hiding, and, in fact, a number of works have been proposed towards integrating Hoare-like reasoning with type checking. Examples include tools and languages like Spec# [3], SPLint [13], ESC/Java [12], and JML [10].

Our prior work on HTT [18, 17] addresses several of the main challenges for languages for integrated programming and verification [10]: (1) we allow effectful code in specifications by granting such code first-class status, via the monad for Hoare triples; (2) we control pointer aliasing, by employing the small footprint approach of separation logic; and (3) we use higher-order logic to allow for a uniform approach to programming and verification of imperative modules (aka mutable abstract data types), as suggested for separation logic in [6, 7]. In our earlier work on HTT we proved soundness of the type theory via mostly operational methods, by proving progress and type preservation results. The operational proof was combined with a very crude denotational model, which just served to show that the assertion logic of HTT was sound. To deal with dependent types the operational proofs relied heavily on sophisticated techniques involving so-called hereditary substitutions [24].

In this paper we define a realizability model for an extension of Hoare Type Theory with impredicative polymorphism. Apart from the inherent interest in obtaining a denotational model, which provides an alternative more abstract conceptual understanding of the theory, the model serves the following purposes:

- Using the model we can prove soundness of an extension of Hoare Type Theory with *impredicative* polymorphism. Impredicative polymorphism is important for data abstraction (we show an examples below) and for representing certain compiler transformations, such as closure conversion, in HTT. It is well-known that the operational methods involving hereditary substitutions mentioned above do not easily scale to impredicative polymorphism.

- The model makes the connections to separation logic more transparent. Indeed, to bring out the connections very clearly, we have decided to present the type theory using a syntax for computations, which is fairly close to the one employed in separation logic.

- The model can be used to investigate which equality rules for computations the theory can soundly be extended with. We present some simple examples in Section 4.

It is non-trivial to construct sound models of sophisticated dependent type theories such as HTT. Models for various fragments of dependent type theories have been studied intensively in categorical type theory; see, e.g., [14] and the references therein. Thus we shall make use of results from categorical type theory to *prove* that we construct a sound model of impredicative HTT, but we shall always write out the definitions in explicit terms so as to make the paper reasonably self contained. Before proceeding with the technical development proper we now give an intuitive overview of the development.

HTT is a dependent type theory with types and kinds, where types are included in the kinds, and where types and kinds can both depend on kinds. Thus contexts Γ assign kinds to variables and there are judgments $\Gamma \vdash \tau : \mathrm{Type}$ and $\Gamma \vdash A : \mathrm{Kind}$ to conclude that τ is a well-formed type in context Γ and that A is a well-formed kind in context Γ. Type and kind formers include dependent product (Π) and dependent sum (Σ). In the extension with impredicative polymorphism that we consider in this paper, we have that Type is a kind. Thus this part of pure impredicative HTT is what is sometimes called (weak) Full Higher-order Dependent Type Theory (FhoDTT) [14]. In addition to types and kinds, HTT also includes a logic for reasoning about terms in context. Thus there is a judgment $\Gamma \vdash P : \mathrm{Prop}$ for concluding that P is a well-formed proposition and a judgment $\Gamma \mid P_1, \ldots, P_n \vdash P$ for logical entailment. The logic is higher-order, so Prop is a kind. In

Jacobs's terminology we thus have a Higher-order Dependent Predicate Logic over (weak) Full Higher-order Dependent Type Theory [14]. The characteristic feature of HTT is that it includes a type for computations $\Gamma \vdash \{P\}\ x : \tau\ \{Q\} : \mathrm{Type}$. Here P and Q are propositions in context Γ and $\Gamma, x : \tau$, respectively. The intuition is that elements of this type consist of computations, which, given a heap satisfying P either diverges or produces a value of type τ and a heap in Q. Note that computations can diverge; term formers for computations include a fixed point term.

The great benefit of impredicative polymorphism is that for any type τ, $\Pi\alpha : \mathrm{Type}.\tau$ is also a type, even if τ depends on α. Thus terms of this polymorphic type can be returned by computations and stored in memory. Prop is also a kind. So again $\Pi P : \mathrm{Prop}.\tau$ is a type where τ may depend on P. This enables us to abstract over predicates in computation types. Using that $\Sigma P : \mathrm{Prop}.\tau$ is a type, we can pack computations with abstract invariants and hide implementation details. As an illustration of both of these features consider the following type of abstract stacks:

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathsf{stacktype} =} \\
\multicolumn{2}{l}{\quad \Pi\alpha : \mathrm{Type}.\Sigma\beta : \mathrm{Type}.\Sigma inv : \beta \times \alpha\,\mathrm{list} \to \mathrm{Prop}.} \\
/*\,\texttt{new}\,*/ & \quad (-).\{\mathrm{emp}\}s : \beta\{inv(s,[])\} \times \\
/*\,\texttt{push}\,*/ & \quad \Pi s : \beta.\Pi x : \alpha. \\
 & \qquad (l : \alpha\,\mathrm{list}).\{inv(s,l)\}u : 1\{inv(s, x :: l)\} \times \\
/*\,\texttt{pop}\,*/ & \quad \Pi s : \beta. \\
 & \qquad (x : \alpha, l : \alpha\,\mathrm{list}). \\
 & \qquad\quad \{inv(s, x :: l)\}y : \alpha\{inv(s,l) \wedge y =_\alpha x\} \times \\
/*\,\texttt{del}\,*/ & \quad \Pi s : \beta. \\
 & \qquad (l : \alpha\,\mathrm{list}).\{inv(s,l)\}u : 1\{\mathrm{emp}\}
\end{array}
$$

The contexts before the precondition in the computation types, e.g., $(l : \alpha\,\mathrm{list})$ for `push`, universally binds auxiliary / logical variables used in the specifications. A term of type `stacktype` accepts a type α and produces a stack of elements of this type. Such a stack consists of

- β, an abstract type to be thought of as α stack.
- inv, an abstract invariant that expresses that objects of type β represents functional stacks (as described by $\alpha\,\mathrm{list}$).
- Operations `new`, `push`, `pop`, and `del`. Notice, that `push`, `pop`, and `del` require an element of type β, and that the only way to obtain one such is to invoke `new`.

Since `stacktype` is itself a type due to impredicativity, we can have stacks of stacks. Note that in separation logic parlance the types are *tight*. For instance, the precondition for `new` is simply emp, which means that `new` does not rely on the input heap; the frame rule ensures that `new` can also be used with the following type $(-).\{\mathrm{emp} * R\}s : \beta\{inv(s,[]) * R\}$, for any R. Further observe that implementors of the above abstract stack type are free to choose both the representation type β and the representation predicate inv. For example, an implementation using linked lists could take β to be Nat (since we use Nat as the type of locations) and $inv(s,l)$ to be the predicate that holds if s points to a linked list representation of l. A simple example client that creates a new Nat stack, pushes 4, pops it again to return it and deletes the stack would

then look like this:

$$
\begin{array}{l}
C = \lambda S : \texttt{stacktype}. \\
\quad \texttt{do}\ S_{\mathrm{Nat}} \leftarrow \texttt{ret}\ S(\mathrm{Nat})\ \texttt{in} \\
\quad\quad \texttt{unpack}\ S_{\mathrm{Nat}}\ \texttt{as}\ (\beta, inv, \texttt{new}, \texttt{push}, \texttt{pop}, \texttt{del})\ \texttt{in} \\
\quad\quad\quad \texttt{do}\ s \leftarrow \texttt{new}\ \texttt{in} \\
\quad\quad\quad\quad \texttt{do}\ s_4 \leftarrow \texttt{push}(s)(4)\ \texttt{in} \\
\quad\quad\quad\quad\quad \texttt{do}\ n_4 \leftarrow \texttt{pop}(s_4)\ \texttt{in} \\
\quad\quad\quad\quad\quad\quad \texttt{del}(s_4); \texttt{ret}\ n_4
\end{array}
$$

Then C has type $\Pi S : \texttt{stacktype}.(-).\{\mathrm{emp}\}n : \mathrm{Nat}\{\mathrm{emp} \wedge n =_{\mathrm{Nat}} 4\}$.

Computations are not only needed for accessing the store but also for nontermination as the pure fragment does not include fixed points. As an example of a simple fixed point computation (not using the store), consider the factorial function $fac : T$, where $T = \Pi n : \mathrm{Nat}.(-).\{\mathrm{emp}\}m : \mathrm{Nat}\{\mathrm{emp} \wedge m =_{\mathrm{Nat}} n!\}$:

$$
\begin{array}{l}
fac = \texttt{fix}\ f(n)\ \texttt{in}\ \texttt{case}\ n\ \texttt{of} \\
\quad\quad \texttt{zero} \Rightarrow \texttt{ret}\ 1\ \texttt{or} \\
\quad\quad \texttt{succ}\ y \Rightarrow \texttt{do}\ m \leftarrow f(y)\ \texttt{in}\ \texttt{ret}\ m \times \texttt{succ}\ y
\end{array}
$$

We will show in detail why it has the claimed type in Section 2.11 We can implement another version of factorial using the store but with the same type, in the following manner. First we define a term $fac_S : T_S$, where $T_S = \Pi l : \mathrm{Nat}.(n : \mathrm{Nat}).\{l \mapsto_{\mathrm{Nat}} n\}u : 1\{l \mapsto_{\mathrm{Nat}} n!\}$:

$$
\begin{array}{l}
fac_S = \texttt{fix}\ f(l)\ \texttt{in}\ \texttt{let}\ t =!_{\mathrm{Nat}} l\ \texttt{in} \\
\quad\quad \texttt{case}\ t\ \texttt{of} \\
\quad\quad\quad \texttt{zero} \Rightarrow l :=_{\mathrm{Nat}} 1\ \texttt{or} \\
\quad\quad\quad \texttt{succ}\ y \Rightarrow \texttt{do}\ l_y \leftarrow \texttt{alloc}_{\mathrm{Nat}}\ y\ \texttt{in} \\
\quad\quad\quad\quad f(l_y); \texttt{do}\ t_y \leftarrow !_{\mathrm{Nat}} l_y\ \texttt{in} \\
\quad\quad\quad\quad\quad l :=_{\mathrm{Nat}} t_y \times \texttt{succ}\ y; \texttt{dealloc}\ l_y
\end{array}
$$

Given this we can implement the factorial function as

$$
\begin{array}{l}
fac' = \lambda n : \mathrm{Nat}.\texttt{do}\ l \leftarrow\ \texttt{alloc}_{\mathrm{Nat}}\ n\ \texttt{in} \\
\quad fac_S(l); \texttt{do}\ r \leftarrow !_{\mathrm{Nat}} l\ \texttt{in}\ \texttt{dealloc}\ l; \texttt{ret}\ r
\end{array}
$$

Now fac' has the same type T as fac. Using the model, we can prove that $fac =_T fac'$ (see Section 4), so we can use them interchangeably when reasoning in the logic. This could not be done in earlier versions of HTT.

Our model is a realizability model, built over a universal domain V, which is sufficiently rich to model divergent computations. The domain V also includes a subdomain of computations, called $\mathrm{T}(V)$.

The model for the weak FhoDTT part of HTT is mostly standard (see, e.g.,[14, Examples 11.6.5 and 11.6.7]): types are interpreted as chain-complete partial equivalence relations (complete pers) over V and kinds are interpreted as so-called assemblies (aka ω-sets) over V. The category of assemblies is an extension of the category of sets and functions which contains the category of complete pers as a full subcategory. The latter ensures that we soundly model that types are included among the kinds. Moreover, the

collection of all complete pers form a set and hence an assembly, and thus we model that Type is a Kind. Terms with type $\Pi x : \tau.\sigma$ are modeled as set-theoretic functions between the set of equivalence classes for the pers interpreting τ and σ which are *realized* by an element in V. That is, there is a continuous function from V to V that maps related elements in the first per to related elements in the second per. In reality, the model is a bit more complicated since we have to deal with *families* of types and kinds to model that types and kinds depend on kinds. Hence everything is indexed/fibred over the category of assemblies.

The propositions in HTT correspond to what is often called assertions in Hoare and separation logic. Hence we model propositions using the power set of heaps, as is standard in separation logic. It is a model of classical logic. Formally, we prove that the standard BI-hyperdoctrine [6] over Set can be extended to one over assemblies, and this guarantees that we get a sound model of the higher-order assertion logic of separation logic (now for dependent types and kinds). We write out the interpretation in concrete terms.

Finally, computation types are modeled roughly as follows. A computation type $\Gamma \vdash (\Delta).\{P\}x : \tau\{Q\} : \mathrm{Type}$ is modeled as an *admissible* per of continuous functions from Heap to $V \times \mathrm{Heap}$ (or, rather, as a family of such, indexed over the interpretation of Γ). A per is admissible if it relates the bottom element to itself and is complete. Admissibility is needed for interpreting fixed points. An interesting issue is what per one should use on heaps. We have decided to use a per which equates two heaps if they have the same domain. This ensures that allocation of new heap cells, modeled here as taking the least unallocated address, will preserve the partial equivalence relation (see Section 3.5 for a discussion of this choice). This description is a bit rough for the following reasons. First, the interpretation ensures that computations can only access memory that is either described by the precondition P or allocated during the computation. Second, the interpretation uses the chain-complete closure of the post-condition Q. This ensures that the computation type really is interpreted as an admissible per. Taking the admissible closure is an alternative to restricting propositions to a fragment that always generates admissible pers or using test-functions/biorthogonality [9] to force admissibility. Third, the interpretation builds in the frame rule from separation logic, essentially by interpreting $\Gamma \vdash (\Delta).\{P\}x : \tau\{Q\} : \mathrm{Type}$ as $\Gamma \vdash \forall R : \mathrm{Prop}.(\Delta).\{P * R\}x : \tau\{Q * R\} : \mathrm{Type}$, at the modeling level. This idea comes from [8, 9]; type theoretically the ideas was also used in the earlier formulations of HTT [18, 17].

In HTT every pure term can also be viewed as a computation. In the model this holds because pure terms are modeled via *continuously realized* functions, and such can, of course, also be extended to continuous computations. Note that a cruder set-theoretic model of the pure fragment of HTT, with types as sets with bounded cardinality and kinds as all sets, would not work: then we would not be able to extend every pure term (any function, not necessarily continuous) to a continuous computation.

This completes our informal overview of the model. Along the way we have given some pointers to related work on models of separation logic and categorical models of dependent type theory. Other very related work include the recent step-indexed model by Appel et. al. [2]. In *loc. cit.* Appel et. al. describe a model that can be used to model types for imperative languages. The model of Appel et. al. is for a much simpler type system than the one we consider since we deal with dependent types involving pre- and postconditions. Appel et. al. do, however, include a treatment of recursive types; we have left that for future work, since it is more complicated in our setting, exactly because our types are much more expressive qua the use of pre- and postconditions and dependency. In contrast with Appel et. al. we further include a logic to reason about terms; so far it is not well-understood how to model logics in step-indexed models.

The remainder of the paper is organized as follows. In Section 2 we present the language of impredicative HTT. The model is then presented in Section 3. In Section 4 we show how to derive some sound equations from the model, and in Section 5 we conclude and describe future work.

2 Language

In this section we present our formulation of impredicative HTT. As mentioned in the introduction we have adapted the earlier formulations of HTT so as to make the connections with separation logic more transparent. We will explain the changes below along with the presentation of the language. We include some examples in Section 2.11.

2.1 Grammar

The grammar for HTT includes a grammar for types and kinds; in earlier presentations of HTT they were called monotypes and types, respectively. As mentioned in the introduction, there is a kind Type of all types and a kind Prop of all propositions.

On the term level we have pure terms and computations, which can be effectful. Computations are not a separate syntactic category but rather terms of a certain class of types, namely all types of the form $(\Gamma).\{P\}x : \tau\{Q\}$.

The grammar for types, kinds, propositions, terms and computations is as follows:

$$
\begin{array}{lrl}
\text{Types} & \tau, \sigma, \rho ::= & \mathrm{Nat} \mid 1 \mid \Pi^T x : A.\tau \mid \Sigma^T x : A.\tau \mid \\
& & (\Gamma).\{P\}x : \tau\{P\} \\
\text{Kinds} & A, B ::= & \tau \mid \mathrm{Type} \mid \mathrm{Prop} \mid \Pi^K x : A.A \mid \Sigma^K x : A.A \\
\text{Prop's} & P, Q, R ::= & \top \mid \bot \mid M =_A M \mid P \wedge P \mid P \vee P \mid \\
& & P \supset P \mid \neg P \mid \forall x : A.P \mid \exists x : A.P \mid \\
& & \mathrm{emp} \mid M \mapsto_\tau M \mid P \ * \ P \mid P \ \mathrel{-\!\!*} \ P \\
\text{Terms} & M, N ::= & x \mid \mathtt{zero} \mid \mathtt{succ}\ M \mid \mathtt{rec}_{\mathrm{Nat}}(M, M) \mid () \mid \\
& & \lambda^K x : A.M \mid \lambda^T x : A.M \mid M\ M \mid \\
& & (M, M)^K \mid (M, M)^T \mid \mathtt{fst}\ M \mid \\
& & \mathtt{snd}\ M \mid \mathtt{unpack}\ M\ \mathtt{as}\ (x, y)\ \mathtt{in}\ M \\
& & \mathtt{case}\ M\ \mathtt{of}\ \mathtt{zero} \Rightarrow M\ \mathtt{or}\ \mathtt{succ}\ x \Rightarrow M \\
& & \mathtt{fix}\ f(x)\ \mathtt{in}\ M \mid \mathtt{ret}\ M \mid \\
& & !_\tau M \mid M :=_\tau M \mid \mathtt{do}\ x \leftarrow M\ \mathtt{in}\ M \mid \\
& & \mathtt{alloc}_\tau\ M \mid \mathtt{dealloc}\ M
\end{array}
$$

and the language sports the following sequents, which will be explained in the following:

$$
\begin{array}{c}
\Gamma \vdash A : \mathrm{Kind} \qquad \Gamma \vdash A = A : \mathrm{Kind} \\
\Gamma \vdash \tau : \mathrm{Type} \\
\Gamma \vdash P : \mathrm{Prop} \qquad \Gamma \vdash P : \mathrm{PureProp} \\
\Gamma \vdash M : A \qquad \Gamma \vdash M = M : A \\
\Gamma \mid \Theta \vdash P
\end{array}
$$

To express the pre- and post conditions of computations in terms of propositions, we define a convenient macro:

$$M \mapsto_\tau - = \exists x : \tau . M \mapsto_\tau x$$

The model that we present in the Section 3 also accommodates coproducts of types and kinds, but we have omitted these from this paper.

2.2 Structural Rules

Here are the structural rules. We let $\mathcal{J}$ denote anything that can appear on the right side of a turnstile $\vdash$:

$$\frac{\Gamma \vdash A : \mathrm{Kind}}{\Gamma, x : A \vdash x : A}\, proj \qquad \frac{\Gamma \vdash M : A \quad \Gamma, x : A, \Delta \vdash \mathcal{J}}{\Gamma, \Delta[M/x] \vdash \mathcal{J}[M/x]}\, sub$$

$$\frac{\Gamma, x : A, y : A, \Delta \vdash \mathcal{J}}{\Gamma, x : A, \Delta[x/y] \vdash \mathcal{J}[x/y]}\, cont \qquad \frac{\Gamma \vdash A : \mathrm{Kind} \quad \Gamma \vdash \mathcal{J}}{\Gamma, x : A \vdash \mathcal{J}}\, weak$$

$$\frac{\Gamma \vdash B : \mathrm{Kind} \quad \Gamma, x : A, y : B, \Delta \vdash \mathcal{J}}{\Gamma, y : A, x : B, \Delta \vdash \mathcal{J}}\, ex$$

2.3 Contexts

A context $\Gamma = x_1 : A_1, \ldots, x_n : A_n$ is a list of kinded variables. Since any type is also a kind, we do not have a separate type context. We also use Δ to range over contexts. These are the rules for context formation:

$$\frac{}{\emptyset\ \mathrm{Ctx}} \qquad \frac{\Gamma \vdash A : \mathrm{Kind}}{\Gamma, x : A\ \mathrm{Ctx}}\, x \notin \Gamma$$

To use any of the following rules, one must first establish $\Gamma\ \mathrm{Ctx}$.

2.4 Kind Judgments

Rules for establishing $\Gamma \vdash A : \mathrm{Kind}$ are as follows:

$$\frac{\Gamma \vdash \tau : \mathrm{Type}}{\Gamma \vdash \tau : \mathrm{Kind}}\, ext$$

$$\frac{}{\emptyset \vdash \mathrm{Type} : \mathrm{Kind}}\, \mathrm{Type} \quad \frac{}{\emptyset \vdash \mathrm{Prop} : \mathrm{Kind}}\, \mathrm{Prop}$$

$$\frac{\Gamma, x : A \vdash B : \mathrm{Kind}}{\Gamma \vdash \Pi^K x : A.B : \mathrm{Kind}}\, \Pi K \quad \frac{\Gamma, x : A \vdash B : \mathrm{Kind}}{\Gamma \vdash \Sigma^K x : A.B : \mathrm{Kind}}\, \Sigma K$$

$$\frac{\Gamma, x : A \vdash B : \mathrm{Kind} \quad \Gamma \vdash A = A' : \mathrm{Kind}}{\Gamma, x : A' \vdash B : \mathrm{Kind}}\, eqctx$$

Note that types are included into kinds via ext.

2.5 Type Judgments

Rules for establishing $\Gamma \vdash \tau : \mathrm{Type}$ are as follows:

$$\frac{}{\emptyset \vdash \mathrm{Nat} : \mathrm{Type}}\, \mathrm{Nat} \quad \frac{}{\emptyset \vdash 1 : \mathrm{Type}}\, 1$$

$$\frac{\Gamma, x : A \vdash \tau : \mathrm{Type}}{\Gamma \vdash \Pi^T x : A.\tau : \mathrm{Type}}\, \Pi T \quad \frac{\Gamma, x : A \vdash \tau : \mathrm{Type}}{\Gamma \vdash \Sigma^T x : A.\tau : \mathrm{Type}}\, \Sigma T$$

$$\frac{\Gamma, \Delta \vdash \tau : \mathrm{Type} \quad \Gamma, \Delta \vdash P : \mathrm{Prop} \quad \Gamma, \Delta, x : \tau \vdash Q : \mathrm{Prop}}{\Gamma \vdash (\Delta).\{P\}x : \tau\{Q\} : \mathrm{Type}}\, spec$$

Note that the language supports dependent products and sums of families of both kinds A and types τ. The sums for kinds Σ^K are strong, the sums for types are weak, c.f. the discussion regarding the elimination rules below.

2.6 Proposition Judgments

Rules for establishing $\Gamma \vdash P : \mathrm{Prop}$ are as follows:

$$\frac{}{\Gamma \vdash \top : \mathrm{Prop}} \quad \frac{}{\Gamma \vdash \bot : \mathrm{Prop}}$$

$$\frac{\Gamma \vdash M = N : A}{\Gamma \vdash M =_A N : \mathrm{Prop}}\, exteq \qquad \frac{\Gamma \vdash M : A \quad \Gamma \vdash N : A}{\Gamma \vdash M =_A N : \mathrm{Prop}}\, eq$$

$$\frac{}{\Gamma \vdash \mathrm{emp} : \mathrm{Prop}}\, \mathrm{emp}$$

$$\frac{\Gamma \vdash M : \mathrm{Nat} \quad \Gamma \vdash \tau : \mathrm{Type} \quad \Gamma \vdash N : \tau}{\Gamma \vdash M \mapsto_\tau N : \mathrm{Prop}}\, \mapsto$$

$$\frac{\Gamma \vdash P : \mathrm{Prop} \quad \Gamma \vdash Q : \mathrm{Prop}}{\Gamma \vdash P \ * \ Q : \mathrm{Prop}}\, * \qquad \frac{\Gamma \vdash P : \mathrm{Prop} \quad \Gamma \vdash Q : \mathrm{Prop}}{\Gamma \vdash P \ -\!\!* \ Q : \mathrm{Prop}}\, -\!\!*$$

$$\frac{\Gamma \vdash P : \mathrm{Prop} \quad \Gamma \vdash Q : \mathrm{Prop} \quad \mathit{op} \in \{\wedge, \vee, \supset\}}{\Gamma \vdash P \ \mathit{op}\ Q : \mathrm{Prop}}$$

$$\frac{\Gamma \vdash P : \mathrm{Prop}}{\Gamma \vdash \neg P : \mathrm{Prop}} \qquad \frac{\Gamma, x : A \vdash P : \mathrm{Prop} \quad Q \in \{\forall, \exists\}}{\Gamma \vdash Qx : A.P : \mathrm{Prop}}$$

The judgment $\Gamma \vdash P : \mathrm{PureProp}$ holds if $\Gamma \vdash P : \mathrm{Prop}$ and, moreover, P does not contain emp, $\mapsto$, $*$, $-\!\!*$.

2.7 Logical Entailment Rules

Entailment is formulated in a judgment of the form $\Gamma \mid \Theta \vdash P$ where Θ is a list of propositions $P_1, \ldots, P_n$ such that, for all i, $\Gamma \vdash P_i : \mathrm{Prop}$ and, moreover, $\Gamma \vdash P : \mathrm{Prop}$.

We write $\Gamma \vdash P$ for $\Gamma \mid \top \vdash P$.

We have the standard rules for classical higher-order predicate logic together with the standard rules for separation logic. In particular, we have extensionality of entailment for propositions (two propositions are equal at kind Prop iff they entail each other) and extensionality for functions.

2.8 Typing Rules

Rules for establishing $\Gamma \vdash M : A$ are as follows:

Pure Terms

$$\frac{}{\Gamma \vdash \mathtt{zero} : \mathrm{Nat}} \qquad \frac{\Gamma \vdash M : \mathrm{Nat}}{\Gamma \vdash \mathtt{succ}\ M : \mathrm{Nat}}$$

$$\frac{\Gamma \vdash M : A \quad \Gamma, x : A \vdash N : A}{\Gamma \vdash \mathtt{rec}_{\mathrm{Nat}}(M, N) : \Pi n : \mathrm{Nat}.A}\, rec \qquad \frac{}{\Gamma \vdash () : 1}$$

$$\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda^K x : A.M : \Pi^K x : A.B} \qquad \frac{\Gamma, x : A \vdash \tau : \mathrm{Type} \quad \Gamma, x : A \vdash M : \tau}{\Gamma \vdash \lambda^T x : A.M : \Pi^T x : A.\tau}$$

$$\frac{\Gamma \vdash M : \Pi^{K,T} x : B.A \quad \Gamma \vdash N : B}{\Gamma \vdash M\ N : A[B/x]}$$

$$\frac{\Gamma \vdash M : A \quad \Gamma \vdash N : B[M/x]}{\Gamma \vdash (M, N)^K : \Sigma^K x : A.B}$$

$$\frac{\Gamma \vdash M : A \quad \Gamma, x : A \vdash \tau : \mathrm{Type} \quad \Gamma \vdash N : \tau[M/x]}{\Gamma \vdash (M, N)^T : \Sigma^T x : A.\tau}$$

$$\frac{\Gamma, x : A \vdash \sigma : \mathrm{Type} \quad \Gamma, x : A, y : \sigma \vdash M : B}{\Gamma, z : \Sigma^T x : A.\sigma \vdash \mathtt{unpack}\ z\ \mathtt{as}\ (x, y)\ \mathtt{in}\ M : B}$$

$$\frac{\Gamma \vdash M : \Sigma^K x : A.B}{\Gamma \vdash \mathtt{fst}\ M : A} \qquad \frac{\Gamma \vdash M : \Sigma^K x : A.B}{\Gamma \vdash \mathtt{snd}\ M : B[\mathtt{fst}\ M/x]}$$

Note that there are two sets of elimination rules for sums (one with `unpack` z `as` (x, y) `in` M and one with `fst` and `snd`). The first one is used for the weak sums over families of types, i.e., for sums $\Sigma^T x : A.\sigma$ (a type) and and the second one is used for the strong sums over families of kinds, i.e., for sums $\Sigma^K x : A.B$ (a kind). In the following section describing the model we explain why we get these different kinds of elimination rules when we show the concrete interpretation of sums. We sometimes leave out the superscript ^{K} and ^{T}, when there is no risk of confusion.

Computations

$$\frac{\Gamma \vdash M : (\Delta).\{P\}y : \sigma\{S\} \quad \Gamma, \Delta, x : \tau \vdash Q : \mathrm{Prop} \quad \Gamma, y : \sigma \vdash N : (\Delta).\{S\}x : \tau\{Q\}}{\Gamma \vdash \mathtt{do}\ y \leftarrow M\ \mathtt{in}\ N : (\Delta).\{P\}x : \tau\{Q\}}\ seq$$

$$\frac{\Gamma, \Delta \vdash \tau : \mathrm{Type} \quad \Gamma \vdash M : \tau}{\Gamma \vdash \mathtt{ret}\ M : (\Delta).\{\mathrm{emp}\}x : \tau\{\mathrm{emp} \wedge x =_\tau M\}}\ dia$$

$$\frac{\Gamma \vdash \tau : \mathrm{Type} \quad \Gamma \vdash M : \mathrm{Nat}}{\Gamma \vdash !_\tau M : (y : \tau).\{M \mapsto_\tau y\}x : \tau\{M \mapsto_\tau y \wedge x =_\tau y\}}\ lookup$$

$$\frac{\Gamma \vdash \tau : \mathrm{Type} \quad \Gamma \vdash M : \mathrm{Nat} \quad \Gamma \vdash N : \tau}{\Gamma \vdash M :=_\tau N : (-).\{M \mapsto_\sigma -\}x : 1\{M \mapsto_\tau N\}}\ update$$

$$\frac{\Gamma \vdash \tau : \mathrm{Type} \quad \Gamma \vdash M : \tau}{\Gamma \vdash \mathtt{alloc}_\tau\ M : (-).\{\mathrm{emp}\}x : \mathrm{Nat}\{x \mapsto_\tau M\}}\ alloc$$

$$\frac{\Gamma \vdash \tau : \mathrm{Type} \quad \Gamma \vdash M : \mathrm{Nat}}{\Gamma \vdash \mathtt{dealloc}\ M : (-).\{M \mapsto_\tau -\}x : 1\{\mathrm{emp}\}}\ dealloc$$

$$\frac{\Gamma \vdash M_1 : (\Delta).\{P \wedge M =_{\mathrm{Nat}} \mathtt{zero}\}x : \tau\{Q\} \quad \Gamma \vdash M : \mathrm{Nat} \quad \Gamma, y : \mathrm{Nat} \vdash M_2 : (\Delta).\{P \wedge M =_{\mathrm{Nat}} \mathtt{succ}\ y\}x : \tau\{Q\}}{\Gamma \vdash \mathtt{case}\ M\ \mathtt{of}\ \mathtt{zero} \Rightarrow M_1\ \mathtt{or}\ \mathtt{succ}\ y \Rightarrow M_2 : (\Delta).\{P\}x : \tau\{Q\}}\ case$$

$$\frac{\Gamma, f : \Pi^T y : A.(\Delta).\{P\}x : \tau\{Q\}, y : A \vdash M : (\Delta).\{P\}x : \tau\{Q\}}{\Gamma \vdash \mathtt{fix}\ f(x)\ \mathtt{in}\ M : \Pi^T y : A.(\Delta).\{P\}x : \tau\{Q\}}\ fix$$

Since we have recursion over the natural numbers in the pure fragment, one might ask why the factorial example in the introduction is coded via a combination of fix and $case$ and perhaps even question the need for a case rule for computations. It has been included for the following reason: when reasoning about each branch of a $case$, the pre-condition will contain information about which branch it is. This allows us to conclude that fac indeed does compute the factorial in its post-condition (se Section 2.11 for details).

We often abbreviate $\mathtt{do}\ y \leftarrow M\ \mathtt{in}\ N$ to $M; N$ when y does not occur in N.

2.9 Structural Rules for Computations

$$\frac{\Gamma \vdash M : (\Delta).\{R\}x : \tau\{S\} \quad \Gamma, \Delta \vdash P \supset R \quad \Gamma, \Delta \vdash S \supset Q}{\Gamma \vdash M : (\Delta).\{P\}x : \tau\{Q\}}\ consequent$$

$$\frac{\Gamma \vdash M : (\Delta).\{P\}x : \tau\{Q\} \quad \Gamma, \Delta \vdash R : \mathrm{Prop}}{\Gamma \vdash M : (\Delta).\{P * R\}x : \tau\{Q * R\}}\ frame$$

$$\frac{\Gamma \vdash M : (\Delta).\{P\}x : \tau\{Q\} \quad \Gamma, \Delta \vdash R : \mathrm{PureProp}}{\Gamma \vdash M : (\Delta).\{P \wedge R\}x : \tau\{Q \wedge R\}}\ \wedge - frame$$

$$\frac{\Gamma \vdash M : (\Delta).\{P\}x : \tau\{Q\} \quad \Gamma, \Delta \vdash \sigma : \mathrm{Kind}}{\Gamma \vdash M : (\Delta, y : \sigma).\{P\}x : \tau\{Q\}}\ weakening$$

$$\frac{\begin{array}{l}\Gamma, \Delta \vdash P : \mathrm{Prop} \quad \Gamma, \Delta, x : \tau \vdash Q : \mathrm{Prop} \\ \Gamma \vdash M : (\Delta, y : \sigma).\{P\}x : \tau\{Q\}\end{array}}{\Gamma \vdash M : (\Delta).\{P\}x : \tau\{Q\}}\ strengthening$$

The $\wedge$-frame rule is not standard in separation logic and indeed is not valid in standard models of separation logic, where term variables can be mutable. The intuitive reason for why the rule holds in our model is that the proposition R is assumed to be pure (not involving the heap) and, moreover, that all our term variables are immutable. In other words R can only speak about immutable variables so if it holds before a computation is executed, then it also holds after the computation is executed since the variables are not mutated.

2.10 External Equality

The external equality rules for terms, types, kinds, and propositions correspond to the least congruence relation derived from the following equations.

$$(\lambda^{K,T} x : A.M)N = M[N/x] : \Pi^{K,T} x : A.B$$
$$M = \lambda^{K,T} x : A.Mx : \Pi^{K,T} x : A.B$$
$$\mathtt{fst}\ (M, N) = M : A$$
$$\mathtt{snd}\ (M, N) = N : A[M/x]$$
$$\mathtt{unpack}\ z\ \mathtt{as}\ (x, y)\ \mathtt{in}\ (x, y) = z : \tau$$
$$\mathtt{unpack}\ (M, N)\ \mathtt{as}\ (x, y)\ \mathtt{in}\ K = K[M/x, N/y] : \tau$$
$$M = (\mathtt{fst}\ M, \mathtt{snd}\ M) : \Sigma^K x : A.B$$
$$\mathtt{rec}_{\mathrm{Nat}}(M, N)(\mathtt{zero}) = M : A$$
$$\mathtt{rec}_{\mathrm{Nat}}(M, N)(\mathtt{succ}\ (n)) = N(\mathtt{rec}_{\mathrm{Nat}}(M, N)(n)) : A$$
$$\mathtt{do}\ x \leftarrow M\ \mathtt{in}\ \mathtt{ret}\ x = M : (\Delta).\{P\}x : \tau\{Q\}$$
$$\mathtt{do}\ x \leftarrow \mathtt{ret}\ M\ \mathtt{in}\ N = N[M/x] : (\Delta).\{P\}y : \tau\{Q\}$$
$$\mathtt{do}\ x \leftarrow (\mathtt{do}\ y \leftarrow M\ \mathtt{in}\ N)\ \mathtt{in}\ K =$$
$$\mathtt{do}\ y \leftarrow M\ \mathtt{in}\ (\mathtt{do}\ x \leftarrow N\ \mathtt{in}\ K) : (\Delta).\{P\}x : \tau\{Q\}$$

2.11 Examples

As mentioned in Section 1, we can code the factorial function $fac : \Pi n : \mathrm{Nat}.(-).\{\mathrm{emp}\}m : \mathrm{Nat}\{\mathrm{emp} \wedge m =_{\mathrm{Nat}} n!\}$ as

$$\begin{array}{l} fac = \mathtt{fix}\ f(n)\ \mathtt{in}\ \mathtt{case}\ n\ \mathtt{of} \\ \qquad \mathtt{zero} \Rightarrow \mathtt{ret}\ 1\ \mathtt{or} \\ \qquad \mathtt{succ}\ y \Rightarrow \mathtt{do}\ m \leftarrow f(y)\ \mathtt{in}\ \mathtt{ret}\ m \times \mathtt{succ}\ y \end{array}$$

Let us discuss why it has the claimed type. For convenience let T denote $\Pi n : \mathrm{Nat}.(-).\{\mathrm{emp}\}m : \mathrm{Nat}\{\mathrm{emp} \wedge m =_{\mathrm{Nat}} n!\}$. We begin with the two branches, starting with the `zero` case. Since $n : \mathrm{Nat} \vdash 1 : \mathrm{Nat}$ we get

$$n : \mathrm{Nat} \vdash \mathtt{ret}\ 1 : (-).\{\mathrm{emp}\}x : \mathrm{Nat}\{\mathrm{emp} \wedge x =_{\mathrm{Nat}} 1\}.$$

Using $\wedge\text{-}frame$ we obtain

$$\begin{array}{l} n : \mathrm{Nat} \vdash \mathtt{ret}\ 1 : (-). \\ \quad \{\mathrm{emp} \wedge n =_{\mathrm{Nat}} 0\}x : \mathrm{Nat}\{\mathrm{emp} \wedge x =_{\mathrm{Nat}} 1 \wedge n =_{\mathrm{Nat}} 0\} \end{array}$$

By consequence we can weaken the post-condition slightly to obtain

$$n : \mathrm{Nat} \vdash \mathtt{ret}\ 1 : (-).\{\mathrm{emp} \wedge n =_{\mathrm{Nat}} 0\}x : \mathrm{Nat}\{\mathrm{emp} \wedge x =_{\mathrm{Nat}} n!\}.$$

We can then weaken the context to $f : T, n : \mathrm{Nat}$.

For the `succ` case we can similarly conclude that

$$\begin{array}{l} y : \mathrm{Nat}, m : \mathrm{Nat} \vdash \mathtt{ret}\ m \times \mathtt{succ}\ y : (-). \\ \quad \{\mathrm{emp} \wedge m =_{\mathrm{Nat}} y!\}x : \mathrm{Nat}\{\mathrm{emp} \wedge x =_{\mathrm{Nat}} (\mathtt{succ}\ y)!\} \end{array}$$

which by weakening, $\wedge\text{-}frame$ and consequent gives

$$\begin{array}{l} f : T, n : \mathrm{Nat}, y : \mathrm{Nat}, m : \mathrm{Nat} \vdash \mathtt{ret}\ m \times \mathtt{succ}\ y : (-). \\ \quad \{\mathrm{emp} \wedge m =_{\mathrm{Nat}} y! \wedge n =_{\mathrm{Nat}} \mathtt{succ}\ y\}x : \mathrm{Nat}\{\mathrm{emp} \wedge x =_{\mathrm{Nat}} n!\} \end{array}$$

Now, using $\wedge\text{-}frame$, we can show

$$\begin{array}{l} f : T, n : \mathrm{Nat}, y : \mathrm{Nat} \vdash f(y) : (-). \\ \quad \{\mathrm{emp} \wedge n =_{\mathrm{Nat}} \mathtt{succ}\ y\}m : \mathrm{Nat} \\ \quad \{\mathrm{emp} \wedge m =_{\mathrm{Nat}} y! \wedge n =_{\mathrm{Nat}} \mathtt{succ}\ y\} \end{array}$$

Thus we can run the two in sequence to obtain

$$\begin{array}{l} f : T, n : \mathrm{Nat}, y : \mathrm{Nat} \vdash \mathtt{do}\ m \leftarrow f(y)\ \mathtt{in}\ \mathtt{ret}\ m \times \mathtt{succ}\ y : (-). \\ \quad \{\mathrm{emp} \wedge n =_{\mathrm{Nat}} \mathtt{succ}\ y\}x : \mathrm{Nat}\{\mathrm{emp} \wedge x =_{\mathrm{Nat}} n!\} \end{array}$$

The case rule now gives that

$$\begin{array}{l} \mathtt{case}\ n\ \mathtt{of} \\ \quad \mathtt{zero} \Rightarrow \mathtt{ret}\ 1\ \mathtt{or} \\ \quad \mathtt{succ}\ y \Rightarrow \mathtt{do}\ m \leftarrow f(y)\ \mathtt{in}\ \mathtt{ret}\ m \times \mathtt{succ}\ y \end{array}$$

has type T in context $f : T, n : \mathrm{Nat}$. Thus, by the fix rule, also fac has type T.

3 Model

In this section we define our realizability model. We begin by describing the universe of realizers; next we define a structure for modeling the pure type theory (kinds and types); a structure for modeling the logic (props); and finally we explain how computations are modeled.

3.1 Universe of Realizers

Let $Cppo_\bot$ denote the category of chain-complete pointed partial orders and strict continuous functions. Recall that one can solve recursive domain equations in $Cppo_\bot$ for locally continuous bifunctors on $Cppo_\bot$. We take our universe of realizers to be a domain V satisfying a recursive domain equation in $Cppo_\bot$. To define V we first recall a number of objects in $Cppo_\bot$ and locally continuous (bi)functors on $Cppo_\bot$.

In $Cppo_\bot$, we find the following objects:

$1_\bot$**:** $\{\bot, *\}$ with $\bot < *$.

$\mathbb{N}_\bot$**:** The flat naturals. The set of natural numbers, all related to themselves, none related to any other, augmented with a bottom element.

$\mathbb{E}$**:** $\{\bot, \mathtt{err}\}$ with $\bot < \mathtt{err}$. The lifted error value. Isomorphic to $1_\bot$.

We further find the following functors:

$\multimap$**:** strict continuous function space.

$\oplus$**:** smash sum.

$\otimes$**:** smash product.

H**:** $V \mapsto \{h \in \mathrm{Cppo}_\bot(\mathbb{N}_\bot, V) \mid \mathrm{supp}(h) \text{ is finite}\}$, where $\mathrm{supp}(h)$ is the set $\{x \in \mathrm{dom}(h) \mid h(x) \neq \bot\}$, ordered in the following way: $h < h' \Leftrightarrow \mathrm{supp}(h) = \mathrm{supp}(h') \wedge \forall n \in \mathrm{supp}(h).h(n) < h'(n)$. The functorial action is by composition.

T**:** $V \mapsto \mathrm{H}(V)_\bot \multimap V \otimes \mathrm{H}(V)_\bot \oplus \mathbb{E}$.

The domain of realizers is a domain V satisfying the following recursive domain equation in $Cppo_\bot$:

$$V \cong 1_\bot \oplus \mathbb{N}_\bot \oplus (V \times V)_\bot \oplus (V \to V)_\bot \oplus \mathrm{T}(V)_\bot$$

where $V \to V$ is the set of continuous functions from V to V. Note that $(V \times V)_\bot \simeq V_\bot \otimes V_\bot$ and $V \to V \simeq V_\bot \multimap V$, so the above recursive domain equation really can be solved in $Cppo_\bot$.

To denote elements in V we shall make use of the following injection maps, mapping elements into the appropriate summand and then, via the above isomorphism, into V.

$$\begin{aligned} in_1 &: 1 \to V \\ in_\mathbb{N} &: \mathbb{N} \to V \\ in_\times &: (V \times V) \to V \\ in_\to &: (V \to V) \to V \\ in_\mathrm{T} &: \mathrm{T}(V) \to V \end{aligned}$$

We use these maps so that whenever we write $in_\to(f)$, say, we know that f is a function and not the added bottom element of $(V \to V)_\bot$.

Lemma 1 *V is a total combinatory algebra.*

Proof.
This follows since $V \to V$ is a retract of V. $\square$

3.2 Semantic Operations on Heaps

Elements of $\mathrm{H}(V)$ are total functions albeit with finite support. We wish to think of them as partial functions in order to model separation logic. This is accomplished by interpreting $h(n) = \bot$ as "n is not allocated in h". Here we describe some definitions reflecting this interpretation.

Firstly we can express that two heaps are "equally defined". For $h, h' \in \mathrm{H}(V)$ we define the relation $h \stackrel{\downarrow}{=} h'$ as h and h' having the same support.

We can then define the $*$-operator on "disjoint" heaps. For heaps $h_1, h_2 \in \mathrm{H}(V)$ such that $\mathrm{supp}(h_1) \cap \mathrm{supp}(h_2) = \emptyset$, we define $h_1 * h_2$ as the heap with support $\mathrm{supp}(h_1) \cup \mathrm{supp}(h_2)$ satisfying $(h_1 * h_2)|_{\mathrm{supp}(h_1)} = h_1 \wedge (h_1 * h_2)|_{\mathrm{supp}(h_2)} = h_2$. In other words, $h_1 * h_2$ is the (disjoint) amalgamation of h_1 and h_2.

For $h \in \mathrm{H}(V)$, it makes sense to ask for "the least unallocated cell of h". $\mathtt{leastfree}(h)$ is defined as $\min\{n \in \mathbb{N} \mid h(n) = \bot\}$.

Updating the heap cell n is by redefining the value at n. For $h \in \mathrm{H}(V)$, $n \in \mathbb{N}$ and $d \in V$, we define the heap $h[n \mapsto d]$ by

$$\lambda m \in \mathbb{N}.\mathtt{if}\ m = n\ \mathtt{then}\ d\ \mathtt{else}\ h(m).$$

Allocation is then given by updating a cell that was previously unallocated with an element different from $\bot$ and deallocation of cell n in h results in $h[n \mapsto \bot]$.

3.3 Types and Kinds

In this section we describe the FhoDTT structure needed for interpreting types and kinds. As mentioned in the introduction, the structure is reasonably standard; it is a variation over the one described in, e.g., [14, Examples 11.6.7]; we use another universe of realizers and we use complete pers instead of extensional pers.

First we describe the category $\mathrm{Asm}(V)$ of assemblies over V, which will be used for modeling contexts:

Definition ($\mathrm{Asm}(V)$):

Objects: (X, E), where X is a set, and $E : X \to \mathsf{P}(V)$, such that for all $x \in X$, $E(x) \neq \emptyset$.

Morphisms: $f : (X, E) \to (X', E')$, where $f : X \to X'$ is a set-theoretic function, such that there exists a realizer α for it, i.e

$$\exists \alpha : V \to V. \forall x \in X. \forall d \in E(x). \alpha(d) \in E(f(x))$$

Note that $\mathrm{Asm}(V)$ is an extension of the category of sets and functions: there is a full and faithful functor $\nabla : \mathrm{Set} \to \mathrm{Asm}(V)$, which maps a set X to (X, E) with $E(x) = V$. Functor ∇ is right adjoint to $\Gamma : \mathrm{Asm}(V) \to \mathrm{Set}$, defined by $\Gamma(X, E) = X$, that is, there is a one-to-one correspondence between morphisms $(X, E) \to \nabla(Y)$ in $\mathrm{Asm}(V)$ and functions $X \to Y$ in Set.

Kinds in context are modeled as objects in a category $\mathrm{UFam}(\mathrm{Asm}(V))$ which is the total category of the fibration

$$\begin{array}{c} \mathrm{UFam}(\mathrm{Asm}(V)) \\ \downarrow \\ \mathrm{Asm}(V) \end{array} \tag{1}$$

Let us explain what this means concretely. For every object (X, E) in $\mathrm{Asm}(V)$, there is a category, called the fibre over (X, E) and denoted by $\mathrm{UFam}(\mathrm{Asm}(V))_{(X,E)}$. It is defined as follows:

Definition ($\mathrm{UFam}(\mathrm{Asm}(V))_{(X,E)}$):

Objects: $((A_x, E_{A_x}))_{x\in X}$ families of assemblies over V indexed by X, i.e. for all $x \in X$, (A_x, E_{A_x}) is an object of $\mathrm{Asm}(V)$.

Morphisms: $(f_x)_{x\in X}$:$((A_x, E_{A_x}))_{x\in X} \to ((B_x, E_{B_x}))_{x\in X}$, where for all $x \in X$, $f_x : A_x \to B_x$ and there exists a *uniform* realizer $\alpha \in V \to V \to V$, i.e.

$$\begin{array}{l} \forall x \in X.\forall e \in E(x).\forall a \in A_x.d \in E_{A_x}(a) \Rightarrow \\ \quad \alpha(e)(d) \in E_{B_x}(f(a)). \end{array}$$

Thus objects in the fibre over (X, E) are families of assemblies indexed over X and morphisms between two such families are uniformly realized morphisms between members of the family. The functor in (1) maps an indexed family to its indexing object (X, E). The fact that the functor is a fibration means in particular that whenever we have a family $((A_y, E_{A_y}))_{y\in Y}$ in the fibre over (Y, E_Y) and a morphism $u : (X, E_X) \to (Y, E_Y)$ in $\mathrm{Asm}(V)$, then we can *reindex* the family to be a family $((A_{f(x)}, E_{A_{f(x)}}))_{x\in X}$ over (X, E_X).

This indexed structure is needed to interpret kinds in context: if context Γ is interpreted as object (X, E) in $\mathrm{Asm}(V)$, then kind $\Gamma \vdash A : \mathrm{Kind}$ is interpreted as an object in the fibre over (X, E), i.e., as a family of assemblies indexed over X.

The fibration of uniform families of assemblies is equivalent to the standard codomain fibration over assemblies, denoted $\mathrm{Asm}(V)^{\to} \to \mathrm{Asm}(V)$.

Types in context are modeled as objects of the total category of the fibration

$$\begin{array}{c} \mathrm{UFam}(\mathrm{CPer}(V)) \\ \downarrow \\ \mathrm{Asm}(V) \end{array} \tag{2}$$

of which the fibre over (X, E) is given by

Definition ($\mathrm{UFam}(\mathrm{CPer}(V))_{(X,E)}$):

Objects: $(R_x)_{x\in X}$ families of chain-complete partial equivalence relations over V indexed by X, i.e. for all $x \in X$, R_x is a chain-complete per over V.

Morphisms: $(f_x)_{x\in X}$:$(R_x)_{x\in X} \to (S_x)_{x\in X}$, where for all $x \in X$, $f_x : V/R_x \to V/S_x$ in Set (here V/R_x is the set of equivalence classes of R_x) and there exists a *uniform* realizer $\alpha : V \to V \to V$, i.e.

$$\forall x \in X.\forall e \in E(x).\forall v \in |R_x|.\alpha(e)(v) \in f_x([v]_{R_x}).$$

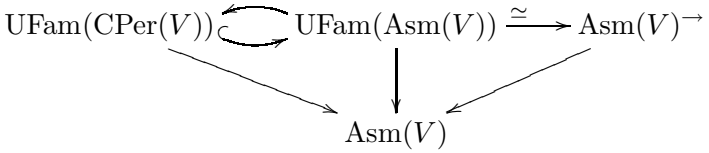
There is a full and faithful fibred inclusion from $\mathrm{UFam}(\mathrm{CPer}(V))$ to $\mathrm{UFam}(\mathrm{Asm}(V))$, which maps a family of pers to a family of assemblies. It works simply by viewing every per as an assembly, which can be done as follows. Suppose R is a per; then the corresponding assembly is $(V/R, E)$, where V/R is the set of equivalence classes of R and E is the identity function. In fact, the inclusion has a fibred left adjoint:

Lemma 2 *The fibred inclusion of* $\mathrm{UFam}(\mathrm{CPer}(V))$ *into* $\mathrm{UFam}(\mathrm{Asm}(V))$ *has a fibred left adjoint.*

Proof.
The proof proceeds by showing that $\mathrm{UFam}(\mathrm{CPer}(V))$ forms a fibred reflective subcategory of $\mathrm{UFam}(\mathrm{Per}(V))$ and then composing with the well-known fibred reflection between $\mathrm{UFam}(\mathrm{Per}(V))$ and $\mathrm{UFam}(\mathrm{Asm}(V))$ (see, e.g., [14, 1.2.6 and 1.8.7]). For the former, we just present the proof that the inclusion of $\mathrm{CPer}(V)$ into $\mathrm{Per}(V)$ has a left adjoint; it lifts easily to uniform families. The left adjoint maps a per R to $\overline{R}$, the least complete per containing R. For adjointness, we need to show that morphisms from R to S, with S a complete per, are in one-to-one correspondence with morphisms from $\overline{R} \to S$. To this end, one gives a construction of $\overline{R}$ by transfinite recursion and then shows the correspondence by transfinite induction, as in [1, Section 15.7]. □

We now present the formal statement which ensures that the structures defined above can be used to model soundly the pure type and kind fragment of HTT. After that, we explain how types and kinds are modeled concretely.

Theorem 1 *The categories and functors in the diagram*



constitute a split weak FhoDTT [14, Ch. 11] with a fibred natural numbers object in $\mathrm{UFam}(\mathrm{CPer}(V))$*, which is also a fibred natural numbers object in* $\mathrm{UFam}(\mathrm{Asm}(V))$.

Proof.
As in [14, Example 11.6.4], using Lemmas 1 and 2. In the fibre over (X, E), the natural numbers object is the constant family of pers $(R_x)_{x \in X}$ given by $R_x = \{(in_{\mathbb{N}}(n), in_{\mathbb{N}}(n)) \mid n \in \mathbb{N}\}$. □

Corollary 1 *The pure type and kind fragment (excluding computation types) of HTT is sound wrt. the interpretation in the above FhoDTT.*

We now explain in concrete terms how pure types, kinds, and terms are interpreted in the FhoDTT. For a context Γ, let Kind_Γ denotes the set of syntactic kinds A such that $\Gamma \vdash A : \mathrm{Kind}$ and Types_Γ denotes the set of syntactic types τ such that $\Gamma \vdash \tau : \mathrm{Type}$. For a kind A in Kinds_Γ, let Terms_A denote the set of syntactic terms M such that $\Gamma \vdash M : A$. The interpretation of contexts, types, kinds, and pure terms is

given by functions:

$$\begin{aligned}
[\![-]\!]^{\mathrm{Ctxs}} :&\quad \mathrm{Ctxs} \to \mathrm{Asm},\\
[\![-]\!]^{\mathrm{Kinds}} :&\quad \Sigma_{\Gamma\in\mathrm{Ctxs}}\mathrm{Kinds}_\Gamma \to \mathrm{UFam}(\mathrm{Asm}(V))_{[\![\Gamma]\!]^{\mathrm{Ctxs}}},\\
[\![-]\!]^{\mathrm{Types}} :&\quad \Sigma_{\Gamma\in\mathrm{Ctxs}}\mathrm{Types}_\Gamma \to \mathrm{UFam}(\mathrm{CPer}(V))_{[\![\Gamma]\!]^{\mathrm{Ctxs}}},\\
[\![-]\!]^{\mathrm{Terms}} :&\quad \Sigma_{\Gamma\in\mathrm{Ctxs}}\Sigma_{A\in\mathrm{Kinds}_\Gamma}\mathrm{Terms}_A \to\\
&\qquad \mathrm{UFam}(\mathrm{Asm}(V))_{[\![\Gamma]\!]^{\mathrm{Ctxs}}}(1, [\![A]\!]^{\mathrm{Kinds}}).
\end{aligned}$$

Note that $[\![-]\!]^{\mathrm{Ctxs}}$ appears in the domain of the other functions. This is feasible since the functions are defined by simultaneous induction on the derivation and in order to derive that, say, a certain type is a type, one first has to derive that the appropriate context is a context. Thus $[\![-]\!]^{\mathrm{Ctxs}}$ will be defined per the induction hypothesis. For the same reason $[\![-]\!]^{\mathrm{Kinds}}$ can be used in the domain of $[\![-]\!]^{\mathrm{Terms}}$.

The empty context is interpreted as the terminal object in $\mathrm{Asm}(V)$:

$$[\![\emptyset]\!]^{\mathrm{Ctxs}} = 1 = (\{*\}, * \mapsto V)$$

and if $[\![\Gamma]\!]^{\mathrm{Ctxs}} = (X, E)$ and $[\![\Gamma \vdash A : \mathrm{Kind}]\!]^{\mathrm{Kinds}} = ((A_x, E_{A_x}))_{x\in X}$ then $[\![\Gamma, x : A]\!]^{\mathrm{Ctxs}}$ is

$$(\Sigma_{x\in X}A_x, (x, a) \mapsto \{(d, d') \in V \times V \mid d \in E(x) \wedge d' \in E_{A_x}(a)\})$$

Thus context formation is modeled by dependent sum.

We now describe the interpretation of kinds.

ext is modeled by the inclusion from $\mathrm{UFam}(\mathrm{CPer}(V))$ to $\mathrm{UFam}(\mathrm{Asm}(V))$.

Type is modeled as an object in the fibre $\mathrm{UFam}(\mathrm{Asm}(V))_1$ over the terminal object 1 in $\mathrm{Asm}(V)$, i.e., as an object in $\mathrm{Asm}(V)$, namely the object $\nabla(Obj(\mathrm{CPer}(V))$, where $Obj(\mathrm{CPer}(V))$ is the set of all chain-complete pers over V.

Prop is modeled by $\nabla\mathsf{P}(\mathrm{H}(V))$ (a full explanation will be given in the next subsection).

ΠK is modeled by dependent product: If $[\![\Gamma \vdash A : \mathrm{Kind}]\!]^{\mathrm{Kinds}} = ((A_x, E_{A_x}))_{x\in X}$ and $[\![\Gamma, x : A \vdash B : \mathrm{Kind}]\!]^{\mathrm{Kinds}} = ((B_{(x,a)}, E_{B_{(x,a)}}))_{(x,a)\in\Sigma x:X.A_x}$ then $[\![\Gamma \vdash \Pi^K x : A.B : \mathrm{Kind}]\!]^{\mathrm{Kinds}}$ is given by

$$(\{f \in \Pi_{a\in A_x}B_{(x,a)} \mid E_{\Pi_x}(f) \neq \emptyset\}, E_{\Pi_x})_{x\in X},$$

where E_{Π_x} is given by

$$f \mapsto \{in_{\to}(g) \mid \forall a \in A_x.e \in E_{A_x}(a) \Rightarrow g\ e \in E_{B_{(x,a)}}(f(a))\}.$$

ΣK is modeled by dependent sum: If $[\![\Gamma \vdash A : \mathrm{Kind}]\!]^{\mathrm{Kinds}} = ((A_x, E_{A_x}))_{x\in X}$ and $[\![\Gamma, x : A \vdash B : \mathrm{Kind}]\!]^{\mathrm{Kinds}} = ((B_{(x,a)}, E_{B_{(x,a)}}))_{(x,a)\in\Sigma x:X.A_x}$ then $[\![\Gamma \vdash \Sigma^K x : A.B : \mathrm{Kind}]\!]^{\mathrm{Kinds}}$ is given by

$$\begin{aligned}
&(\Sigma_{a\in A_x}B_{(x,a)}, (a, b) \mapsto\\
&\quad \{in_{\times}(d, e) \mid d \in E_{A_x}(a) \wedge e \in E_{B_{(x,a)}}(b)\})_{x\in X}.
\end{aligned}$$

eqctx External equality of kinds is interpreted by equality in the model, so the interpretation of $\Gamma, x : A' \vdash B : \mathrm{Kind}$ is the same as the interpretation of $\Gamma, x : A \vdash B : \mathrm{Kind}$.

We now describe the interpretation of the pure types:

Nat is modeled by the flat naturals, i.e $(\{(in_{\mathbb{N}}(n), in_{\mathbb{N}}(n)) \mid n \in \mathbb{N}\})$

1 is modeled by the terminal object in $\mathrm{CPer}(V)$, i.e., as $(\{(in_1(*), in_1(*))\})$.

ΠT is modeled by dependent product: If $[\![\Gamma \vdash A : \mathrm{Kind}]\!]^{\mathrm{Kinds}} = ((A_x, E_{A_x}))_{x \in X}$ and $[\![\Gamma, x : A \vdash \tau : \mathrm{Type}]\!]^{\mathrm{Types}} = (R_{(x,a)})_{(x,a) \in \Sigma x : X. A_x}$ then $[\![\Gamma \vdash \Pi^T x : A.\tau : \mathrm{Type}]\!]^{\mathrm{Types}}$ is given by

$$\{(in_{\to}(f), in_{\to}(g)) \mid \forall a \in A_x. e \in E_{A_x}(a) \Rightarrow f(e)\ R_{(x,a)}\ g(e)\}$$

ΣT is modeled by dependent sum: If $[\![\Gamma \vdash A : \mathrm{Kind}]\!]^{\mathrm{Kinds}} = ((A_x, E_{A_x}))_{x \in X}$ and $[\![\Gamma, x : A \vdash \tau : \mathrm{Type}]\!]^{\mathrm{Types}} = (R_{(x,a)})_{(x,a) \in \Sigma x : X. A_x}$ then $[\![\Gamma \vdash \Sigma^T x : A.\tau : \mathrm{Type}]\!]^{\mathrm{Types}}$ is given by

$$\overline{\{(in_{\times}(d, e), in_{\times}(d', e')) \mid \exists a \in A_x. d, d' \in E_{A_x}(a) \wedge e\ R_{(x,a)} e'\}}.$$

Note the use of the chain completion (the reflection into $\mathrm{UFam}(\mathrm{CPer}(V))$). We need to use the chain-completion to get a chain-complete per and the elements in the chain-completion are not necessarily pairs of realizers for the constituent types. It is because of the use of this chain-completion that these sums are only weak, i.e., that the associated elimination rule is the rule for `unpack`, rather than rules involving `fst` and `snd`. Indeed, if we try to apply the first-projection realizer to a realizer for an element of the above sum, then we will not be sure to end up with a realizer for A (we only know that we'll get something in the chain-completion of A).

An external equality judgment of kinds $\Gamma \vdash A = B : \mathrm{Kind}$ *holds* if A and B are interpreted as the same objects in the fibre over the interpretation of Γ. Likewise for external equality of types $\Gamma \vdash \tau = \sigma : \mathrm{Type}$. The soundness corollary 1 means that any external equality judgment that can be derived holds.

Lemma 3 *For any type $\Gamma \vdash \sigma : \mathrm{Type}$, no per in the family $[\![\Gamma \vdash \sigma : \mathrm{Type}]\!]^{\mathrm{Types}}$ relates $\perp$ to itself.*

Proof.
By induction on the derivation of $\Gamma \vdash \sigma : \mathrm{Type}$. □
The above lemma shows that any well-typed term corresponds to a proper value in the model, even the diverging computation. The computation types relate the least element of $\mathrm{T}(V)$ to itself.

We now describe the concrete interpretation of terms. For $\Gamma \vdash M : A$, $[\![\Gamma \vdash M : A]\!]^{\mathrm{Terms}}$ gives a morphism in the fibre category $\mathrm{UFam}(\mathrm{Asm}(V))_{[\![\Gamma]\!]^{\mathrm{Ctxs}}}$ from the terminal object 1 to $[\![A]\!]^{\mathrm{Kinds}}$. Such a morphism can be described by a continuous function that maps realizers for the context to realizers for A: Assume that $[\![\Gamma]\!]^{\mathrm{Ctxs}} = (X, E_X)$ and that $[\![A]\!]^{\mathrm{Kinds}} = (Y_x, E_{Y_x})_{x \in X}$. Then a morphism

$$(f_x)_{x \in X} : (\{*\}, * \mapsto \{in_1(*)\})_{x \in X} \to (Y_x, E_{Y_x})_{x \in X}$$

has a uniform realizer α, such that

$$\forall x \in X. \forall e \in E_X(x). \forall a \in \{*\}. \forall d \in \{in_1(*)\}. \alpha(e)(d) \in E_{Y_x}(f(a))$$

Since a is always $*$ in the above equation we can simplify it a bit. If we denote $f(*)$ by $[\![M]\!]$ and write $\alpha(e)(in_1(*))$ as just $\alpha(e)$, the requirement becomes

$$\forall x \in X. \forall e \in E_X(x). \alpha(e) \in E_{Y_x}([\![M]\!])$$

In the case that A is a type, $[\![M]\!]$ will be the equivalence class of the complete per $[\![A]\!]^{\mathrm{Types}}$. In this case f is uniquely defined by α. Otherwise $[\![M]\!]$ will just be an element of Y_x, and the subset of V, $E_{Y_x}([\![M]\!])$, may overlap with other subsets returned by E_{Y_x}. Thus, we must provide both $[\![M]\!]$ and α, unless we know A to be a type.

The interpretation of terms of pure types and kinds $[\![\Gamma \vdash M : A]\!]^{\mathrm{Terms}}$ is as follows:

$$\begin{aligned}
&[\![\Gamma \vdash \mathtt{zero} : \mathrm{Nat}]\!]^{\mathrm{Terms}} = \lambda e. in_{\mathbb{N}}(0)\\
&[\![\Gamma \vdash \mathtt{succ}\ M : \mathrm{Nat}]\!]^{\mathrm{Terms}}\\
&\quad = \lambda e. \mathtt{case}\ [\![\Gamma \vdash M : \mathrm{Nat}]\!]^{\mathrm{Terms}}(e)\ \mathtt{of}\\
&\qquad\qquad in_{\mathbb{N}}(m) \Rightarrow in_{\mathbb{N}}(m+1)\\
&\qquad\qquad \mathtt{otherwise} \Rightarrow \bot\\
&[\![\Gamma \vdash \mathtt{rec}_{\mathrm{Nat}}(M, N)]\!]^{\mathrm{Terms}}\\
&\quad = \lambda e. in_{\rightarrow}(\lambda v. \mathtt{case}\ v\ \mathtt{of}\ in_{\mathbb{N}}(x)\ \Rightarrow f(x)\\
&\qquad\qquad\qquad\qquad \mathtt{otherwise}\ \Rightarrow \bot),\\
&\qquad \text{where } f(0) = [\![\Gamma \vdash M]\!]^{\mathrm{Terms}}(e)\\
&\qquad \text{and } f(n+1) = [\![\Gamma \vdash N]\!]^{\mathrm{Terms}}(e)(f(n))\\
&[\![\Gamma \vdash () : 1]\!]^{\mathrm{Terms}} = \lambda e. in_1(*)\\
&[\![\Gamma \vdash \lambda^{T,K} x : A. M : \Pi^{T,K} x : A. B]\!]^{\mathrm{Terms}}\\
&\quad = \lambda e. in_{\rightarrow}(\lambda x. [\![\Gamma, x : A \vdash M : B]\!]^{\mathrm{Terms}}(e, x))\\
&[\![\Gamma \vdash M\ N : A[B/x]]\!]^{\mathrm{Terms}}\\
&\quad = \lambda e. \mathtt{case}\ [\![\Gamma \vdash M : \mathrm{Nat}]\!]^{\mathrm{Terms}}(e)\ \mathtt{of}\\
&\qquad\qquad in_{\rightarrow}(f) \Rightarrow f([\![\Gamma \vdash N : \mathrm{Nat}]\!]^{\mathrm{Terms}}(e))\\
&\qquad\qquad \mathtt{otherwise} \Rightarrow \bot\\
&[\![\Gamma \vdash (M, N)^{T,K} : \Sigma^{T,K} x : A. B]\!]^{\mathrm{Terms}}\\
&\quad = \lambda e. \mathtt{let}\ d = [\![\Gamma \vdash M : A]\!]^{\mathrm{Terms}}(e)\ \mathtt{in}\\
&\qquad\quad in_{\times}(d, [\![\Gamma, x : A \vdash N : B]\!]^{\mathrm{Terms}}(e, d))\\
&[\![\Gamma, z : \Sigma^T x : A. \sigma \vdash \mathtt{unpack}\ z\ \mathtt{as}\ (x, y)\ \mathtt{in}\ M : \tau]\!]^{\mathrm{Terms}}\\
&\quad = \lambda(e, (x, y)). [\![\Gamma, x : A, y : \sigma \vdash M : \tau]\!]^{\mathrm{Terms}}((e, x), y)\\
&[\![\Gamma \vdash \mathtt{fst}\ M : A]\!]^{\mathrm{Terms}}\\
&\quad = \lambda e. \mathtt{case}\ [\![\Gamma \vdash M : \Sigma^K x : A. B]\!]^{\mathrm{Terms}}(e)\ \mathtt{of}\\
&\qquad\qquad in_{\times}(m, n) \Rightarrow m\\
&\qquad\qquad \mathtt{otherwise} \Rightarrow \bot\\
&[\![\Gamma \vdash \mathtt{snd}\ M : A]\!]^{\mathrm{Terms}}\\
&\quad = \lambda e. \mathtt{case}\ [\![\Gamma \vdash M : \Sigma^K x : A. B]\!]^{\mathrm{Terms}}(e)\ \mathtt{of}\\
&\qquad\qquad in_{\times}(m, n) \Rightarrow n\\
&\qquad\qquad \mathtt{otherwise} \Rightarrow \bot
\end{aligned}$$

Most of the interpretations are straightforward. The interpretation of `unpack` works because of the following. The realizers x and y come from the interpretation of the weak sum, hence from the chain-completion (c.f., the explicit description of the interpretation of sums). But since $[\![\Gamma, x : A, y : \sigma \vdash M : \tau]\!]^{\mathrm{Terms}}$ is a *continuous* realizer, it also works for elements x and y possibly added via the chain-completion process. Formally this is exactly what Lemma 2 guarantees (see the proof sketch of the lemma).

This completes the description of the interpretation of the non-structural rules for pure types and kinds. The structural rules are interpreted using the basic fibrational structure of an FhoDTT. Here we just discuss a simple example; see [14] for details. Suppose that Γ is interpreted by (X, E_X), that $\Gamma \vdash A : \mathrm{Kind}$ is interpreted by the family $(Y_x, E_{Y_x})_{x \in X}$ over (X, E_X), that $\Gamma \vdash M : A$ is interpreted by $m : 1 \to (Y_x, E_{Y_x})_{x \in X}$ over (X, E_X), and $\Gamma, x : A \vdash B : \mathrm{Kind}$ is interpreted by $(Z_{(x,y)}, F_{(x,y)})_{(x,y) \in \Sigma_{x:X} Y_x}$. Then $\Gamma \vdash B[M/x] : \mathrm{Kind}$ is interpreted by the family $(Z_{(x,m(x))}, F_{(x,m(x))})_{x \in X}$.

We say that an external judgment of kinds $\Gamma \vdash M = N : A$ *holds* if M and N are interpreted as the same morphism. The soundness corollary 1 means that any external equality judgment of terms that can be derived using the rules in Sections 2.10 holds.

3.4 Logic

As in separation logic, the logic is really a logic of heaps and hence propositions will be modeled as subsets of $\mathrm{H}(V)$. Again, we begin with the abstract description of the structure needed and then follow it by a concrete description of the interpretation.

We obtain the structure needed for interpreting the logic as follows. First, the power set of heaps $\mathsf{P}(\mathrm{H}(V))$ ordered by inclusion is a BI-algebra in Set. We now embed it into $\mathrm{Asm}(V)$ via the functor ∇ to get $\nabla(\mathsf{P}(\mathrm{H}(V)))$. One can now show that the object is an internally complete BI-algebra in $\mathrm{Asm}(V)$. Hence, as explained in [6], there is a canonical BI-hyperdoctrine $P = \mathrm{Asm}(\_, \nabla(\mathsf{P}(\mathrm{H}(V))))$, which soundly models classical higher-order separation logic. Note that the fibre over an object (X, E) in P is the set of morphisms in $\mathrm{Asm}(V)$ from (X, E) to $\nabla(\mathsf{P}(\mathrm{H}(V)))$, which, as mentioned earlier, is in one-to-one correspondence with functions from X to $\mathsf{P}(\mathrm{H}(V))$ in Set. In line with the earlier presentations of structures for types and kinds, we turn P into a fibration via the Grothendieck construction to get the fibration

$$\begin{array}{c} \mathrm{Fam}(\mathsf{P}(\mathrm{H}(V))) \\ \Big\downarrow \\ \mathrm{Asm}(V) \end{array} \tag{3}$$

of which the fibre over (X, E) is the poset given by

Definition ($\mathrm{Fam}(\mathsf{P}(\mathrm{H}(V)))_{(X,E)}$):

Objects: $\phi : X \to \mathsf{P}(\mathrm{H}(V))$ functions from X to $\mathsf{P}(\mathrm{H}(V))$.

Morphisms: $\phi \leq \psi$ iff for all $x \in X$, $\phi(x) \subseteq \psi(x)$.

Theorem 2 *The fibration in* (3) *is a BI-hyperdoctrine with quantification along all maps in the base category.*[1]

[1]The phrase "with quantification along all maps in the base category" means that there are left and right adjoint along all reindexing functor u^*, for all morphisms u in the base. This ensures that the logic quantifiers can be interpreted over the dependent type theory (c.f. [14, Section 11.2]). In the definition of BI-hyperdoctrine given in [6] we only asked for quantifiers with respect to simple projections since that suffices for interpreting logic over a non-dependent type theory.

Proof.
Each fibre is a BI-algebra since it is defined pointwise over the BI-algebra $\mathsf{P}(\mathrm{H}(V))$. Clearly, $\nabla(\mathsf{P}(\mathrm{H}(V)))$ is a generic object. For every $u : (X, E_X) \to (Y, E_Y)$ in $\mathrm{Asm}(V)$, there is a right adjoint $\forall_u$ to reindexing along u, given by $\forall_u(\phi)(y) = \{h \in \mathrm{H}(V) \mid \forall x \in X.u(x) = y \supset h \in \phi(x)\}$. The Beck-Chevalley condition is easily seen to hold. Existential quantifiers are given similarly. □

Corollary 2 *The interpretation of the logic in the above BI-hyperdoctrine is sound.*

We now describe the interpretation of the proposition judgments concretely. For a context Γ, let Props_Γ denote the set of syntactic predicates P such that $\Gamma \vdash P : \mathrm{Prop}$. The interpretation function has the type

$$[\![-]\!]^{\mathrm{Props}} : \Sigma_{\Gamma \in \mathrm{Ctxs}} \mathrm{Props}_\Gamma \to \mathrm{Asm}([\![\Gamma]\!]^{\mathrm{Ctxs}}, \nabla\mathsf{P}(\mathrm{H}(V))).$$

Assume that $[\![\Gamma]\!]^{\mathrm{Ctxs}} = (X, E)$. Then the proposition judgments are interpreted as follows:

$$
\begin{array}{l}
[\![\Gamma \vdash \mathrm{emp} : \mathrm{Prop}]\!]^{\mathrm{Props}}_x = \{\lambda n.\bot\} \\
[\![\Gamma \vdash M \mapsto_\tau N : \mathrm{Prop}]\!]^{\mathrm{Props}}_x = \{h \mid h(m)\ ([\![\tau]\!]^{\mathrm{Types}}_x) = n\}, \\
\quad \text{where } [\![\Gamma \vdash N : \tau]\!]^{\mathrm{Terms}}_x = n \text{ and } [\![\Gamma \vdash M : \mathrm{Nat}]\!]^{\mathrm{Terms}}_x = m \\
[\![\Gamma \vdash P * Q : \mathrm{Prop}]\!]^{\mathrm{Props}}_x = \\
\quad \{h \mid \exists h_1 \in [\![\Gamma \vdash P : \mathrm{Prop}]\!]^{\mathrm{Props}}_x, h_2 \in [\![\Gamma \vdash Q : \mathrm{Prop}]\!]^{\mathrm{Props}}_x. \\
\qquad h = h_1 * h_2\} \\
[\![\Gamma \vdash P \mathrel{-\!\!*} Q : \mathrm{Prop}]\!]^{\mathrm{Props}}_x = \\
\quad \{h \mid \forall h_P \in [\![\Gamma \vdash P : \mathrm{Prop}]\!]^{\mathrm{Props}}_x. h * h_P \in [\![\Gamma \vdash Q : \mathrm{Prop}]\!]^{\mathrm{Props}}_x\} \\
[\![\Gamma \vdash \top : \mathrm{Prop}]\!]^{\mathrm{Props}}_x = \mathrm{H}(V) \\
[\![\Gamma \vdash \bot : \mathrm{Prop}]\!]^{\mathrm{Props}}_x = \emptyset \\
[\![\Gamma \vdash M =_A N : \mathrm{Prop}]\!]^{\mathrm{Props}}_x = \\
\quad \{h \mid [\![\Gamma \vdash M]\!]^{\mathrm{Terms}}_x = [\![\Gamma \vdash N]\!]^{\mathrm{Terms}}_x\} \\
[\![\Gamma \vdash P \wedge Q : \mathrm{Prop}]\!]^{\mathrm{Props}}_x = \\
\quad [\![\Gamma \vdash P : \mathrm{Prop}]\!]^{\mathrm{Props}}_x \cap [\![\Gamma \vdash Q : \mathrm{Prop}]\!]^{\mathrm{Props}}_x \\
\\
\quad [\![\Gamma \vdash P \vee Q : \mathrm{Prop}]\!]^{\mathrm{Props}}_x = \\
\qquad [\![\Gamma \vdash P : \mathrm{Prop}]\!]^{\mathrm{Props}}_x \cup [\![\Gamma \vdash Q : \mathrm{Prop}]\!]^{\mathrm{Props}}_x \\
\quad [\![\Gamma \vdash P \supset Q : \mathrm{Prop}]\!]^{\mathrm{Props}}_x = \\
\qquad \{h \mid h \in [\![\Gamma \vdash P : \mathrm{Prop}]\!]^{\mathrm{Props}}_x \Rightarrow h \in [\![\Gamma \vdash Q : \mathrm{Prop}]\!]^{\mathrm{Props}}_x\} \\
\quad [\![\Gamma \vdash \neg P : \mathrm{Prop}]\!]^{\mathrm{Props}}_x = \mathrm{H}(V) \setminus [\![\Gamma \vdash P]\!]^{\mathrm{Props}}_x \\
\quad [\![\Gamma \vdash \forall y : A.P : \mathrm{Prop}]\!]^{\mathrm{Props}}_x \\
\qquad = \{h \mid \forall y \in [\![\Gamma \vdash A : \mathrm{Kind}]\!]^{\mathrm{Kinds}}_x.\ h \in [\![\Gamma, y : A \vdash P]\!]^{\mathrm{Props}}_{(x,y)}\} \\
\quad [\![\Gamma \vdash \exists x : A.P : \mathrm{Prop}]\!]^{\mathrm{Props}}_x \\
\qquad = \{h \mid \exists y \in [\![\Gamma \vdash A : \mathrm{Kind}]\!]^{\mathrm{Kinds}}_x.\ h \in [\![\Gamma, y : A \vdash P]\!]^{\mathrm{Props}}_{(x,y)}\}
\end{array}
$$

In the display above, we use the following convention when presenting the semantics for the quantifiers. Note that $[\![\Gamma \vdash A : \mathrm{Kind}]\!]^{\mathrm{Kinds}}$ is a uniform family of assemblies over (X, E), so $[\![\Gamma \vdash A : \mathrm{Kind}]\!]^{\mathrm{Kinds}}_x$ is

an assembly (Y, E_Y). When we write $y \in [\![\Gamma \vdash A : \mathrm{Kind}]\!]_x^{\mathrm{Kinds}}$, we mean that $y \in Y$. Note that y may, of course, depend on x.

Let Γ be a context and suppose that $[\![\Gamma]\!]^{\mathrm{Ctxs}} = (X, E)$. We say that a logical entailment $\Gamma \mid P_1, \ldots, P_n \vdash P$ *holds* if, for all $x \in X$, $[\![\Gamma \vdash P_1 \wedge \cdots P_n]\!]^{\mathrm{Props}}(x) \subseteq [\![\Gamma \vdash P]\!]^{\mathrm{Props}}(x)$. The soundness corollary (Corollary 2) means that any logical entailment that can be derived using the rules for logical entailment in Section 2.7 holds.

Now it should also be clear why the kind Prop was interpreted as $\nabla(\mathsf{P}(\mathrm{H}(V))$ earlier.

Lemma 4 *Let $\Gamma \vdash P : \mathrm{PureProp}$, and suppose that $[\![\Gamma]\!]^{\mathrm{Ctxs}} = (X, E)$. Then the interpretation of P is a function ϕ from X to $\mathsf{P}(\mathrm{H}(V))$ satisfying that $\phi(x)$ is either the empty set of the set of all heaps, for all x in X.*

3.5 Computations

We now describe how computations are interpreted in the model. As mentioned in Section 1, a computation type $(\Delta).\{P\}x : \tau\{Q\}$ is modeled as an admissible per of realizers in $\mathrm{T}(V)$, that given heaps satisfying the precondition P do not produce error and upon termination leaves a heap satisfying the postcondition Q. The context Δ is implicitly quantified, so that this behaviour should be adhered to for all instantiations of Δ. Formally it looks like this. Assume $[\![\Gamma]\!]^{\mathrm{Ctxs}} = (X, E)$ and $[\![\Gamma, \Delta]\!]^{\mathrm{Ctxs}} = (\Sigma_{x \in X} Y_x, F)$. Then $[\![\Gamma \vdash (\Delta).\{P\}x : \tau\{Q\} : \mathrm{Type}]\!]^{\mathrm{Types}}$ is the family of pers $(S_x)_{x \in X}$ with fields given by $d \in |S_x|$ iff $d = in_{\mathrm{T}}(f)$ and

$$
\begin{array}{l}
\forall y \in Y_x. \forall E \in \mathrm{Prop}_{\Gamma,\Delta}. \forall h \in [\![\Gamma, \Delta \vdash (P \ * \ E)]\!]^{\mathrm{Props}}_{(x,y)}. \\
(f(h) \neq \mathtt{err}) \wedge \Big(f(h) = (v_f, h_f) \Rightarrow \\
\quad v_f \in |[\![\Gamma, \Delta \vdash \tau : \mathrm{Type}]\!]^{\mathrm{Types}}_{(x,y)}| \wedge \\
\quad h_f \in \overline{[\![\Gamma, \Delta, x : \tau \vdash (Q \ * \ E)]\!]^{\mathrm{Props}}_{(x,y,v_f)}} \Big)
\end{array}
$$

So suitable realizers are elements of $\mathrm{T}(V)$ that for any extension $P \ * \ E$ of P takes heaps satisfying $P \ * \ E$ to heaps satisfying the chain completion of $Q \ * \ E$ and do not produce error. Thus the frame rule is baked into the interpretation of computations. The actual per is then given by $in_{\mathrm{T}}(f) \ S_x \ in_{\mathrm{T}}(g)$ iff $in_{\mathrm{T}}(f), in_{\mathrm{T}}(g) \in |S_x|$ and

$$
\begin{array}{l}
\forall y \in Y_x. \forall E \in \mathrm{Prop}_{\Gamma,\Delta}. \forall h, h' \in [\![\Gamma, \Delta \vdash (P \ * \ E)]\!]^{\mathrm{Props}}_{(x,y)}. \\
h \stackrel{\downarrow}{=} h' \Rightarrow \\
\quad f(h) \downarrow \Leftrightarrow g(h') \downarrow \wedge \Big(f(h) = (v_f, h_f) \wedge g(h') = (v_g, h_g) \Rightarrow \\
\qquad v_f \ [\![\Gamma, \Delta \vdash \tau : \mathrm{Type}]\!]^{\mathrm{Types}}_{(x,y)} \ v_g \ \wedge \ h_f \stackrel{\downarrow}{=} h_g \Big)
\end{array}
$$

So two realizers denote the same computation if they both fulfill the specification and on heaps with equal support gives results related in the interpretation of the return type and heaps with equal support.

Lemma 5 *Let $[\![\Gamma]\!]^{\mathrm{Ctxs}} = (X, E)$ and $[\![\Gamma \vdash (\Delta).\{P\}x : \tau\{Q\} : \mathrm{Type}]\!]^{\mathrm{Types}} = (S_x)_{x \in X}$. Then for all $x \in X$ S_x is a chain-complete per relating $in_{\mathrm{T}}(\lambda h.\bot)$ to itself and relating only elements in (the image of) $\mathrm{T}(V)$. As such it is an admissible per over $\mathrm{T}(V)$.*

Proof.
S_x is easily a per. It trivially relates only elements in $\mathrm{T}(V)$ and $in_{\mathrm{T}}(\lambda h.\bot)$ to itself and it is chain complete

because the interpretation of τ is chain complete as is the chain completion of the interpretation of $Q * E$ and the relation $\overset{\downarrow}{=}$. $\square$

As mentioned in the introduction, the reason we require that computations should produce heaps with equal support (given suitable heaps with equal support) is that then allocation can simply be modeled by taking the least unallocated address (see the semantics of `alloc` below). An unfortunate consequence of this choice is that two computations that intuitively behave in the same way but allocate cells in different order may *not* be equated by the model. We believe that the model can be refined by using realizers in FM-domains [23, 22, 4], such that support would then be up to a permutation of the locations in the heap. (Indeed, FM-domains have already been applied in a recent parametric model for separation logic [9].) We leave this refinement for future work, however.

We now describe how terms of computation types are interpreted in the model. Recall that for a computation type $(\Delta).\{P\}x : \tau\{Q\}$, we can give the interpretation of $\Gamma \vdash M : (\Delta).\{P\}x : \tau\{Q\}$ by giving the realizer α.

We first consider the structural rules for computations. We begin with the frame rule. Assume $[\![\Gamma]\!]^{\mathrm{Ctxs}} = (X, E)$ and that $[\![\Gamma \vdash M : (\Delta).\{P\}x : \tau\{Q\}]\!]^{\mathrm{Terms}}$ is realized by α. Then $[\![\Gamma \vdash M : (\Delta).\{P * R\}x : \tau\{Q * R\}]\!]^{\mathrm{Terms}}$ is also realized by α since, for all $x \in X$, the field of $[\![\Gamma \vdash (\Delta).\{P\}x : \tau\{Q\} : \mathrm{Type}]\!]^{\mathrm{Types}}_x$ is included in the field of $[\![\Gamma \vdash (\Delta).\{P * R\}x : \tau\{Q * R\} : \mathrm{Type}]\!]^{\mathrm{Types}}_x$ (here we use that the frame rule is baked into the interpretation of computation types). The remaining structural rules are also interpreted by using the same realizer (for the consequence rule we use that the chain-completion operation is monotone).

Now for the non-structural rules. Assume $[\![\Gamma]\!]^{\mathrm{Ctxs}} = (X, E)$ and that $[\![M]\!]$ is given by α and $[\![N]\!]$ is given by β when they are of computation types and m and n otherwise. Then

$$
\begin{aligned}
&[\![\Gamma \vdash \mathtt{do}\ y \leftarrow M\ \mathtt{in}\ N : (\Delta).\{P\}x : \tau\{Q\}]\!]^{\mathrm{Terms}} \\
&\quad = \lambda e.\lambda h.\mathtt{case}\ \alpha(e)(h)\ \mathtt{of} \\
&\qquad (v_M, h_M) \ \Rightarrow\ \beta(e, v_M)(h_M) \\
&\qquad\quad \mathtt{err} \ \Rightarrow\ \mathtt{err} \\
&\qquad\quad \bot \ \Rightarrow\ \bot \\
&[\![\Gamma \vdash \mathtt{ret}\ M : (\Delta).\{\mathrm{emp}\}x : \tau\{\mathrm{emp} \wedge x =_\tau M\}]\!]^{\mathrm{Terms}} \\
&\quad = \lambda e.\lambda h.(m(e), h) \\
&[\![\Gamma \vdash\ !_\tau M : (y : \tau).\{M \mapsto_\tau y\}x : \tau\{M \mapsto_\tau y \wedge x =_\tau y\}]\!]^{\mathrm{Terms}} \\
&\quad = \lambda e.\lambda h.\mathtt{if}\ h(m(e)) = \bot\ \mathtt{then}\ \mathtt{err}\ \mathtt{else}\ (h(m(e)), h) \\
&[\![\Gamma \vdash M :=_\tau N : (-).\{M \mapsto -\}x : 1\{M \mapsto_\tau N\}]\!]^{\mathrm{Terms}} \\
&\quad = \lambda e.\lambda h.(*, h[m \mapsto n]) \\
&[\![\Gamma \vdash \mathtt{alloc}_\tau\ M : (-).\{\mathrm{emp}\}x : \mathrm{Nat}\{x \mapsto_\tau M\}]\!]^{\mathrm{Terms}} \\
&\quad = \lambda e.\lambda h.\mathtt{let}\ l = \mathtt{leastfree}(h)\ \mathtt{in}\ (l, h[l \mapsto m]) \\
&[\![\Gamma \vdash \mathtt{dealloc}\ M : (-).\{M \mapsto_\tau -\}x : 1\{\mathrm{emp}\}]\!]^{\mathrm{Terms}} \\
&\quad = \lambda e.\lambda h.\mathtt{if}\ h(m) = \bot\ \mathtt{then}\ \mathtt{err}\ \mathtt{else}\ (*, h[m \mapsto \bot]) \\
&[\![\Gamma \vdash \mathtt{case}\ M\ \mathtt{of}\ \mathtt{zero} \Rightarrow M_1\ \mathtt{or}\ \mathtt{succ}\ y \Rightarrow M_2 : \\
&\quad (\Delta).\{P\}x : \tau\{Q\}]\!]^{\mathrm{Terms}} = \lambda e.\lambda h. \\
&\qquad \mathtt{if}\ m(e) = in_{\mathbb{N}}(0)\ \mathtt{then}\ \alpha_1(e)(h)\ \mathtt{else}\ \alpha_2(e, m-1)(h) \\
&[\![\Gamma \vdash \mathtt{fix}\ f(x)\ \mathtt{in}\ M : \Pi^T y : \sigma.(\Delta).\{P\}x : \tau\{Q\}]\!]^{\mathrm{Terms}} \\
&\quad = \lambda e.\mathtt{fixedpointof}\ \lambda f.\lambda y.\alpha(e, f, y))
\end{aligned}
$$

Note that the realizers for computations are as should be expected. Consider, for example, lookup $!M$, whose realizer is $\lambda e.\lambda h.\mathtt{if}\ h(m(e)) = \bot\ \mathtt{then}\ \mathtt{err}\ \mathtt{else}\ (h(m(e)), h)$. Given a realizer e in $E_X(x)$ (intuitively, a realizer for Γ), it produces a computation that when given a heap h yields error if the location $m(e)$ is not allocated in h and otherwise the value stored in h at $m(e)$, along with h. The realizer e is needed, as always, because the type theory is dependent.

For fixed points, the realizer is obtained by the usual least fixed point construction, which applies since $\lambda f.\lambda y.\alpha(e, f, y)$ is indeed an endofunction of the pointed domain $V \to T(V)$, when α is the realizer for $[\![\Gamma, f : \Pi^T y : \sigma.(\Delta).\{P\}x : \tau\{Q\}, y : \sigma \vdash M : (\Delta).\{P\}x : \tau\{Q\}]\!]^{\mathrm{Terms}}$.

Theorem 3 *The interpretation of computations is well-defined, i.e., any well-typed computation term* $\Gamma \vdash M : (\Delta).\{P\}x : \tau\{Q\}$ *is interpreted as a morphism* $1 \to [\![\Gamma \vdash (\Delta).\{P\}x : \tau\{Q\} : \mathrm{Type}]\!]^{\mathrm{Types}}$ *in the fibre over* $[\![\Gamma]\!]^{\mathrm{Ctxs}}$. *Moreover, the external equality rules for computations hold.*

Proof.
Well-definedness is proved by induction. The monadic external equality rules for computations (the last four rules in Subsection 2.10) hold since the interpretation of computation types really does involve a categorical monad. The essential point is that there is an adjunction between the category of admissible pers over $\mathrm{T}(V)$ and $\mathrm{CPer}(V)$; this adjunction gives rise to a monad on $\mathrm{CPer}(V)$. □

Notice that the above theorem expresses that *well-typed programs do not produce error*: If $[\![\Gamma]\!]^{\mathrm{Ctxs}} = (X, E_X)$ and $[\![\Gamma \vdash M : (\Delta).\{P\}x : \tau\{Q\}]\!]^{\mathrm{Terms}} = m$ then, for all $x \in X$, all $e \in E_X(x)$, m(e) is in $[\![\Gamma \vdash (\Delta).\{P\}x : \tau\{Q\} : \mathrm{Type}]\!]^{\mathrm{Types}}_x$. Thus $m(e)$ is a realizer in $\mathrm{T}(V)$, which given a heap satisfying P does not produces `err`. If $m(e)$ then terminates (does not give $\bot$), it returns a value and a heap in the chain-completion of Q.

Let us discuss the use of the chain-completion briefly.

When computations are typed with a post condition Q that is not chain-complete, the interpretation can be counterintuitive because the resulting heap may not be in Q (but, of course, it will be in Q's chain-completion). As an example we consider the function from [11, Section 5.1]. It takes some recoding due to the monadic presentation of HTT.

Let $\mathrm{Nat}_\bot = (-).\{\mathrm{emp}\}x : \mathrm{Nat}\{\mathrm{emp}\}$. We then have the following terms

$$\begin{aligned}
&\Omega : \mathrm{Nat}_\bot = \text{diverging} \\
&\mathtt{zero}_\bot : \mathrm{Nat}_\bot = \mathtt{ret}\ \mathtt{zero} \\
&\mathtt{succ}\ _\bot : \Pi n : \mathrm{Nat}_\bot.\mathrm{Nat}_\bot \\
&\quad = \lambda n : \mathrm{Nat}_\bot.\mathtt{do}\ m \leftarrow n\ \mathtt{in}\ \mathtt{ret}\ \mathtt{succ}\ m
\end{aligned}$$

We can now describe partial functions. Let $T = \Pi n : \mathrm{Nat}.\mathrm{Nat}_\bot$ then non-totality can be described by the predicate

$$nt = \exists n : \mathrm{Nat}.f(n) =_{\mathrm{Nat}_\bot} \Omega$$

with the free variable $f : T$. From this we can build a type of non-total functions

$$nontotal = (-).\{\mathrm{emp}\}f : T\{\mathrm{emp} \wedge nt(f)\}.$$

Given a partial function we can produce one that is defined in more places via the term $exp : \Pi f : T.T$ given by:

$$\begin{aligned}
\lambda f : T.\lambda n : \mathrm{Nat}.\ &\mathtt{case}\ n\ \mathtt{of}\ \mathtt{zero} \Rightarrow \mathtt{zero}_\bot \\
&\mathtt{or}\ \mathtt{succ}\ y \Rightarrow \mathtt{do}\ t \leftarrow f(y)\ \mathtt{in}\ \mathtt{succ}\ _\bot(t)
\end{aligned}$$

If f is defined at n then $exp(f)$ is defined at $n+1$ as well as on 0. However, if f is non-total then so is $exp(f)$, so we can write a term $expand : \Pi p : nontotal.nontotal$:

$$expand = \lambda p : nontotal.\mathtt{do}\ f \leftarrow p\ \mathtt{in}\ \mathtt{ret}\ exp(f)$$

To type this function we use that $nt(f) \supset nt(exp(f))$. Now taking a fixed point of $expand$ should leave us with the identity function, which is certainly a total function. The fixed point does leave us with the identity function, but we can give it the type $\Pi u : 1.nontotal$:

$$\mathtt{fix}\ f(u)\ \mathtt{in}\ \mathtt{do}\ p \leftarrow f(u)\ \mathtt{in}\ expand(p) : \Pi u : 1.nontotal$$

where f is of type $\Pi u : 1.nontotal$.

This may seem disturbing and highlights the fact that for a post-condition Q the chain-completion of Q may be difficult to guess. The reason that Q is not chain-complete already is that we used existential quantification over an infinite set in the predicate nt, and it is well-known from domain theory that chain-complete predicates are not closed under such existentials. Of course, there are two ways to deal with this issue. One is simply to ban postconditions that are not chain-complete (and there are well-known ways to do so by restricting the grammar for propositions; see, e.g., [11]). Another, which we have chosen

here, is to formally chain-complete the post-condition in the model. This has the following advantage: If the post-condition Q is already chain-complete then chain completing it does not change it, of course. Thus in this case the heap possibly resulting from the computation will in fact satisfy Q. This holds, *even if* subcomputations have been specified using pre- and postconditions that are not chain-complete, qua soundness. For example, since the interpretation of emp is chain complete, we can conclude that any computation of type $(-).\{\mathrm{emp}\}x : \tau\{\mathrm{emp}\}$ must deallocate any memory that it allocates, just as we would expect. Moreover, if you prefer a more standard interpretation, then you can just restrict yourself to using only chain-complete predicates; then our interpretation will indeed be the standard one.

4 Reasoning via the model

We now exemplify reasoning via the model. For reasons of space, we only include two simple examples.

The model can be used to show that terms are propositionally equal. As a simple example, we now argue that the two implementations of factorial fac and fac' presented in the introduction are propositionally equal at their type $T = \Pi n : \mathrm{Nat}.(-).\{\mathrm{emp}\}m : \mathrm{Nat}\{\mathrm{emp} \wedge m =_{\mathrm{Nat}} n!\}$. Looking at the interpretations in the model they both, when applied to a number n, give a computation that takes the empty heap $\lambda m.\bot$ to $(n!, \lambda m.\bot)$, even though fac' allocates and deallocates local storage during its computation. Thus they are related in the interpretation of T. This means that they are propositionally equal, i.e $- \vdash fac =_T fac'$.

We can also use the model to show types propositionally equal. Consider the two computation types

$$\begin{aligned} T_1 &= (x : \sigma, y : \tau).\{P\}r : \rho\{Q\} \quad \text{and} \\ T_2 &= (z : \Sigma x : \sigma.\tau).\{\mathtt{unpack}\ z\ \mathtt{as}\ (x, y)\ \mathtt{in}\ P\} \\ &\qquad\qquad r : \mathtt{unpack}\ z\ \mathtt{as}\ (x, y)\ \mathtt{in}\ \rho \\ &\qquad\qquad \{\mathtt{unpack}\ z\ \mathtt{as}\ (x, y)\ \mathtt{in}\ Q\} \end{aligned}$$

Intuitively they contain the same terms, since computations do not depend on the logic variables. This intuition is reflected in the model as the two types are interpreted as the same family of pers. This can be seen by comparing their fields. Assume $\Gamma \vdash T_1 : \mathrm{Type}$ and $\Gamma \vdash T_2 : \mathrm{Type}$ and that $[\![\Gamma]\!]^{\mathrm{Types}} = (X, E)$. Consider then $in_{\mathrm{H}}(f) \in [\![T_1]\!]_x$. For all realizers d, d' with $((x, d), d')$ in $[\![\Gamma, x : \sigma, y : \tau]\!]^{\mathrm{Ctxs}}$ and r in $[\![\Gamma, x : \sigma, y : \tau \vdash \rho]\!]^{\mathrm{Types}}_{((x,d),d')}$, f must satisfy the computational requirements imposed by $[\![P]\!]^{\mathrm{Props}}_{((x,d),d')}$, $[\![Q]\!]^{\mathrm{Props}}_{(((x,d),d'),r)}$ and $[\![\tau]\!]^{\mathrm{Types}}_{((x,d),d')}$. This happens iff for all realizers d'' with (x, d'') in $[\![\Gamma, z : \Pi x : \sigma.\tau]\!]^{\mathrm{Ctxs}}$ and r in $[\![\Gamma, z : \Pi x : \sigma.\tau \vdash \mathtt{unpack}\ z\ \mathtt{as}\ (x, y)\ \mathtt{in}\ \rho]\!]^{\mathrm{Types}}_{((x,d'')}$, f satisfies the computational requirements imposed by
$[\![\mathtt{unpack}\ z\ \mathtt{as}\ (x, y)\ \mathtt{in}\ P]\!]^{\mathrm{Props}}_{((x,d'')}$, $[\![\mathtt{unpack}\ z\ \mathtt{as}\ (x, y)\ \mathtt{in}\ Q]\!]^{\mathrm{Props}}_{(((x,d''),r)}$ and $[\![\mathtt{unpack}\ z\ \mathtt{as}\ (x, y)\ \mathtt{in}\ \tau]\!]^{\mathrm{Types}}_{((x,d'')}$.

Thus we can conclude that the two types are propositionally equal, i.e., $T_1 =_{\mathrm{Type}} T_2$. Their interpretations contain the same realizers, so we could add two rules to the calculus stating that $\Gamma \vdash M : T_1$ iff $\Gamma \vdash M : T_2$.

5 Conclusion and Future Work

We have developed a realizability model for impredicative Hoare Type Theory, a very expressive dependent type theory in which one can specify and reason about mutable abstract data types. The model is used to establish the soundness of the type theory. Moreover, the model can be used to discover new equations between terms and types; we have presented a few simple examples.

We are presently working on a simple implementation of a tool for HTT, which will make it easier to experiment with larger examples.

Our model also accommodates certain kinds of subset kinds and types. For a kind A we can model the subset kind $\{x : A \mid P\}$, for all propositions P. For a type τ we can model the subset kind $\{x : \tau \mid P\}$, for all *chain-complete* propositions P; it also seems possible to model subset types $\{x : \tau \mid P\}$, for all propositions P by using the chain-completion. The subset kinds / types will not be *full* subset kinds / types, however, for the same reason that we do not have full subset types for the standard separation logic BI-hyperdoctrine over Set [6]. Future work includes investigating how to model recursive types, as needed for the specification of programs that recurse through the store [20]. It would also be interesting to refine the model using, e.g., FM-domains to get a more abstract model of allocation leading to more equalities among terms, c.f. the discussion in Section 3.5. Another avenue for future work is to explore the soundness of higher-order frame rules [8]. This seems to involve a further level of indexing over a Kripke structure similar to the one in [8]. Finally, it would also be interesting to investigate relational parametricity for the impredicative polymorphism.

References

[1] R. Amadio and P.-L. Curien. *Domains and Lambda Calculi*, volume 46 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1998.

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

[3] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In *CASSIS 2004*, Lecture Notes in Computer Science. Springer, 2004.

[4] N. Benton and B. Leperchey. Relational reasoning in a nominal semantics for storage. In *Proc. of TLCA'05*, pages 88–101, Nara, Japan, 2005.

[5] M. Berger, K. Honda, and N. Yoshida. A logical analysis of aliasing in imperative higher-order functions. In O. Danvy and B. C. Pierce, editors, *International Conference on Functional Programming, ICFP'05*, pages 280–293, Tallinn, Estonia, September 2005.

[6] B. Biering, L. Birkedal, and N. Torp-Smith. Bi hyperdoctrines and higher-order separation logic. In *In Proceedings of European Symposium on Programming 2005*, volume 3444 of *Lecture Notes in Computer Science*, pages 233–247, 2005.

[7] B. Biering, L. Birkedal, and N. Torp-Smith. BI hyperdoctrines, Higher-Order Separation Logic, and Abstraction. *TOPLAS*, 2007. To Appear.

[8] L. Birkedal, N. Torp-Smith, and H. Yang. Semantics of separation-logic typing and higher-order frame rules for algol-like languages. *Logical Methods in Computer Science*, 2(5:1):1–33, 2006.

[9] L. Birkedal and H. Yang. Relational parametricity and separation logic. In *Proceedings of 10th International Conference on Foundations of Software Science and Computation Structures, FOSSACS 2007*, number 4423 in LNCS. Spring, 2007.

[10] L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer*, 7(3):212–232, June 2005.

[11] K. Crary. *Type-Theoretic Methodology for Practical Programming Languages*. PhD thesis, Cornell University, 1998.

[12] D. L. Detlefs, K. R. M. Leino, G. Nelson, and J. B. Saxe. Extended static checking. Compaq Systems Research Center, Research Report 159, December 1998.

[13] D. Evans and D. Larochelle. Improving security using extensible lightweight static analysis. *IEEE Software*, 19(1):42–51, 2002.

[14] B. Jacobs. *Categorical Logic and Type Theory*, volume 141 of *Studies in Logic and the Foundations of Mathematics*. Elsevier, 1999.

[15] N. Krishnaswami. Separation logic for a higher-order typed language. In *Workshop on Semantics, Program Analysis and Computing Environments for Memory Management, SPACE'06*, pages 73–82, 2006.

[16] N. Krishnaswami, J. Aldrich, and L. Birkedal. Modular verification of the subject-observer pattern via higher-order separation logic. In *9th Workshop on Formal Techniques for Java-like Programs, FTfJP 2007*, 2007.

[17] A. Nanevski, A. Ahmed, G. Morrisett, and L. Birkedal. Abstract Predicates and Mutable ADTs in Hoare Type Theory. In *European Symposium on Programming, ESOP'07*, volume 4421 of *Lecture Notes in Computer Science*, pages 189–204, 2007.

[18] A. Nanevski, G. Morrisett, and L. Birkedal. Polymorphism and separation in Hoare Type Theory. In *International Conference on Functional Programming, ICFP'06*, pages 62–73, Portland, Oregon, 2006.

[19] P. W. O'Hearn, H. Yang, and J. C. Reynolds. Separation and information hiding. In *Symposium on Principles of Programming Languages, POPL'04*, pages 268–280, 2004.

[20] B. Reus and J. Schwinghammer. Separation logic for higher-order store. In *International Workshop on Computer Science Logic, CSL'06*, 2006.

[21] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Symposium on Logic in Computer Science, LICS'02*, pages 55–74, 2002.

[22] M. Shinwell. *The Fresh Approach: Functional Programming with Names and Binders*. PhD thesis, Computer Laboratory, Cambridge University, Dec. 2004.

[23] M. R. Shinwell and A. M. Pitts. On a monadic semantics for freshness. *Theoretical Computer Science*, 342:28–55, 2005.

[24] K. Watkins, I. Cervesato, F. Pfenning, and D. Walker. A concurrent logical framework: The propositional fragment. In S. Berardi, M. Coppo, and F. Damiani, editors, *Types for Proofs and Programs*, volume 3085 of *Lecture Notes in Computer Science*, pages 355–377. Springer, 2004.

[25] H. Xi and F. Pfenning. Dependent types in practical programming. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 214–227, San Antonio, January 1999.

[26] N. Yoshida, K. Honda, and M. Berger. Local state in hoare logic for imperative higher-order functions. In *Proceedings of 10th International Conference on Foundations of Software Science and Computation Structures, FOSSACS 2007*, number 4423 in LNCS. Spring, 2007.