A Realizability Model for Impredicative Hoare Type Theory

and Greg Morrisett[2]

[1] IT University of Copenhagen
{rusmus,birkedal}@itu.dk
[2] 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 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.

This can be addressed by restricting dependencies to only effect-free terms (e.g. as in DML [27]). But the goal of our work is to realize the full power of dependent types for specification of effectful programs. 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, 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 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 [23,19], 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 [4,14,28,15] 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 a number of works have been proposed towards integrating Hoare-like reasoning with type checking. Examples include tools and languages like Spec# [2], SPLint [12], ESC/Java [11], 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 [5,6]. 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 [26].

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 / consistency of an extension of Hoare Type Theory with *impredicative* polymorphism. Impredicative polymorphism is important for data abstraction (we show an example below) and for representing certain compiler transformations, such as closure conversion [16], in HTT. It is well-known that the operational methods involving hereditary substitutions mentioned above do not easily scale to impredicative polymorphism. We emphasize that it is highly non-trivial to devise a model of dependent type theory combining an impredicative universe of types with a classical logic and with computation types supporting fixed point induction. We summarize the key challenges involved later on in this introduction.
– The model allows us to use syntax and typing rules that have a more natural reading; in earlier presentations of HTT the operational techniques forced clunkier terms (in order to get the theorems to go through). In particular,

the syntax for computations is fairly close to the one employed in separation logic. Our impredicative HTT is the first model of separation logic for such an expressive language (higher types and impredicative polymorphism).
– We can finally introduce some non-trivial equations on computations. The operational approach we took before largely precluded this.

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., [13] 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. We now give an intuitive overview of the development.

Overview of HTT. 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 (and thus types). 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 (weak) Full Higher-order Dependent Type Theory (FhoDTT) [13].

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 [13]. The extra 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:

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

The contexts before the precondition in the computation types, e.g., $(l : \alpha\,\text{list})$ 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 represent functional stacks (as described by $\alpha\,\text{list}$).
– 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 via new.

Since stacktype is by impredicativity itself a type, we can have stacks of stacks. More generally, we can compose first-class abstract data types (i.e., objects) without needing to artificially stratify them which is necessary in modern programming. Note that in separation logic parlance the types are *tight*. For instance, the precondition for new is simply emp, so new does not rely on the input heap; the frame rule ensures that new can also be used with the following type $(-).\{\text{emp} * R\}s\!:\!\beta\{inv(s, []) * R\}$, 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:

```
$C = \lambda S\!:\!\texttt{stacktype}\ .\ \texttt{do}\ S_{\text{Nat}} \leftarrow \texttt{ret}\ S(\text{Nat})\ \texttt{in}$
        $\texttt{unpack}\ S_{\text{Nat}}\ \texttt{as}\ (\beta, inv, \texttt{new}, \texttt{push}, \texttt{pop}, \texttt{del})\ \texttt{in}$
          $\texttt{do}\ s \leftarrow \texttt{new}\ \texttt{in}\ \texttt{push}(s)(4); \texttt{do}\ n_4 \leftarrow \texttt{pop}(s)\ \texttt{in}\ \texttt{del}(s); \texttt{ret}\ n_4$
```

Then C has type $\Pi S\!:\!\texttt{stacktype}\ .\ (-).\{\text{emp}\}n\!:\!\text{Nat}\{\text{emp} \wedge n =_{\text{Nat}} 4\}$. We often (as in C) abbreviate do $y \leftarrow M$ in N to $M; N$ when y does not occur in N.

Computations are not only needed for accessing the store but also for non-termination 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 $T = \Pi n\!:\!\text{Nat}\ .\ (-).\{\text{emp}\}m\!:\!\text{Nat}\{\text{emp} \wedge m =_{\text{Nat}} n!\}$:

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

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 $T_S = \Pi l\!:\!\text{Nat}\ .\ (n\!:\!\text{Nat}).\{l \mapsto_{\text{Nat}} n\}u\!:\!1\{l \mapsto_{\text{Nat}} n!\}$:

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

Given this we can implement the factorial function as

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

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

Overview of Model. 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.,[13, 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 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. We model our classical propositions using the power set of heaps. Formally, we prove that the standard BI-hyperdoctrine [5] over Set can be extended to one over assemblies, and this guarantees that we get a sound model of the higher-order assertion logic (now for dependent types and kinds).

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 $V \times$ 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. 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 idea 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 be extended to continuous computations. Note that in a cruder set-theoretic model of the pure fragment of HTT, with types as sets with bounded cardinality and kinds as all sets, we would not be able to extend every pure term (any function, not necessarily continuous) to a continuous computation.

Let us summarize our informal overview of the model by mentioning what the key technical challenges are in constructing a model: First, note that our impredicative HTT combines a *classical* logic with an impredicative universe of types. Consistency, the very existence of a non-trivial model, is therefore highly non-trivial. It hinges on the fact that impredicative HTT does *not* include full subset types or the axiom of unique choice (that every functional relation determines a term). Second, note that we need to model types as some kind of domains in order to accomodate fixed points for the computation types, and, at the same time, types should form an impredicative universe. That is why we use chain-complete pers and not the more standard model of FhoDTT using all pers, and thus we need to prove that we actually do get a model of HTT using such pers. Third, we need to find chain-complete pers for modelling the computation types. Finally, since the logic is over dependent types we need to prove that we can get a model of separation logic over dependent types.

Related Work. In the previous section we have given some pointers to related work on models of separation logic and categorical models of dependent type theory. Other very related work includes the recent step-indexed model by Appel et. al. [1], where they describe a model that can be used for the types of imperative languages. However, their model 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. It is more challenging in our setting, since our types are much more expressive. (Recursive types should exist, though, since admissible pers do accomodate a wide range of recursive types [7].) 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.

Let us also emphasize the relation to the work of Honda, Yoshida, and Berger on Hoare logics for higher-order languages (see [28] and the references therein). One of the differences between the two approaches is that Honda et. al. do not allow for equational reasoning among functions (as we do in dependent type theory). Instead they make use of an evaluation predicate. Intuitively, the evaluation predicate of Honda et. al. can be used to represent in the logic the distinction between pure terms and computations that we instead capture using the monadic language. Honda et. al. have so far focused on total correctness and have thus avoided the need for admissibility, which we have to deal with as we consider partial correctness and have a rule for fixed point induction.

Honda et. al. are able to deal with recursion through the store, but do not cover impredicative polymorphism.

The remainder of the paper is organized as follows: In Section 2 we present the language of impredicative HTT, and in Section 3 the model. In Section 4 we conclude and describe future work. For reasons of space the formal treatment is brief, please see the accompanying technical report [20] for more details.

2 Language

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

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

and there are the following judgments:

$$\Gamma \vdash A{:}\mathrm{Kind} \qquad \Gamma \vdash A = A : \mathrm{Kind} \qquad \Gamma \vdash \tau{:}\mathrm{Type} \qquad \Gamma \vdash P{:}\mathrm{Prop}$$
$$\Gamma \vdash M{:}A \qquad \Gamma \vdash M = M : A \qquad \Gamma \mid \Theta \vdash P$$

The external equality rules include β- and η-equalities and monadic laws for computations.

To express the pre- and post conditions of computations in terms of propositions, we often write $M \mapsto_\tau -$ as a shorthand for $\exists x{:}\tau.M \mapsto_\tau x$. The model that we present in the Section 3 also accommodates coproducts of types and kinds, but we have omitted these from this paper.

Given the explanation in the Introduction, most of the rules are standard except for those for the computation fragment, which we include below. There are two kinds of sums: $\Sigma^T\ x{:}A\ .\ \sigma$ (a type) is used for weak sums over families of types, and $\Sigma^K\ x{:}A\ .\ B$ (a kind) is used for strong sums over families of kinds. Because of the distinction between weak and strong sums, there are two sets of elimination rules for sums (one with `unpack` M `as` (x, y) `in` M and one with `fst` and `snd`), as is standard. 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.

Here are the non-structural rules for computations. Most of them are unsurprising for a tight interpretation of separation logic. The fix rule is used to define recursive functions and captures reasoning via fixed-point induction.

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

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

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

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

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

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

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

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

The structural rules for computations include the frame rule and the rule of consequence, see [20] for details.

3 Model

Universe of Realizers. Let $Cppo_\perp$ denote the category of chain-complete pointed partial orders and strict continuous functions. Recall that one can solve recursive domain equations in $Cppo_\perp$ for locally continuous bifunctors on $Cppo_\perp$. We take our universe of realizers to be a domain V satisfying the following recursive domain equation in $Cppo_\perp$:

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

where $1_\perp$ is the lift of the one-element set, $\mathbb{N}_\perp$ is lift of the flat natural numbers, $\oplus$ is smash sum, $\times$ is cartesian product, $V \to V$ is the set of continous functions from V to V, and $\mathrm{T}(V)$ is the domain of computations:

$$\mathrm{T}(V) = \mathrm{H}(V)_\perp \multimap \big((V \otimes \mathrm{H}(V)_\perp) \oplus \mathbb{E}\big),$$

in which $\multimap$ denotes strict function space, $\otimes$ is smash product, $\mathbb{E} = \{\mathtt{err}\}_\perp$ and $\mathrm{H}(V)$ is the domain of heaps: $\{h \in \mathrm{Cppo}_\perp(\mathbb{N}_\perp, V) \mid \mathrm{supp}(h) \text{ is finite}\}$, where $\mathrm{supp}(h)$ is the set $\{x \in \mathrm{dom}(h) \mid h(x) \neq \perp\}$, ordered in the following way:

$h \leq h' \Leftrightarrow \mathrm{supp}(h) = \mathrm{supp}(h') \wedge \forall n \in \mathrm{supp}(h).h(n) \leq h'(n)$. Note that H is a locally continous functor whose functorial action is given by composition.

To denote elements in V we use the following injections, mapping elements into the appropriate summand and then, via the above isomorphism, into V.

$$in_1 : 1 \to V \qquad in_{\mathbb{N}} : \mathbb{N} \to V \qquad in_{\times} : (V \times V) \to V$$
$$in_{\to} : (V \to V) \to V \qquad in_{\mathrm{T}} : \mathrm{T}(V) \to V$$

Semantic Operations on Heaps. Elements of $\mathrm{H}(V)$ are total functions 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". This works because two heaps are only related in the partial order if they have the *same* support (and, moreover, are also pointwise ordered). Here we describe some definitions reflecting this interpretation.

Firstly, for $h, h' \in \mathrm{H}(V)$ we define $h \overset{\downarrow}{=} h'$ 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". `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 by updating a cell that was previously unallocated with an element different from $\bot$ and deallocation of cell n in h results in $h[n \mapsto \bot]$.

Types and Kinds. We now describe the FhoDTT structure needed for interpreting types and kinds, beginning with 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 $E \colon X \to \mathsf{P}(V)$, such that for all $x \in X$, $E(x) \neq \emptyset$.
Morphisms: $f \colon (X, E) \to (X', E')$, where $f \colon X \to X'$ 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 $\varGamma : \mathrm{Asm}(V) \to \mathrm{Set}$, defined by $\varGamma(X, E) = X$, 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 interpreted as families of assemblies indexed over assemblies. Formally, the structure is a fibration $\mathrm{UFam}(\mathrm{Asm}(V)) \to \mathrm{Asm}(V)$, defined as in [13]. 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 modelled as families of chain-complete per's indexed over assemblies. We denote the category of chain-complete per's by $\mathrm{CPer}(V)$. The indexing is captured via a fibration $\mathrm{UFam}(\mathrm{CPer}(V)) \to \mathrm{Asm}(V)$, defined similarly to the one for all pers (not only chain-complete pers).

Any complete per R can be seen as an assembly $(V/R, E)$, where V/R is the set of equivalence classes of R and E is the identity function. This will be used to model that types are included among the kinds. This inclusion of complete pers into assemblies extends to families and the extension has a left adjoint:

Lemma 1. *The fibred inclusion of* $\mathrm{UFam}(\mathrm{CPer}(V))$ *into* $\mathrm{UFam}(\mathrm{Asm}(V))$ *has a fibred left adjoint given by chain completion.*

We now present the formal statement which ensures that we can 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*

$$\begin{tikzcd}
\mathrm{UFam}(\mathrm{CPer}(V)) \arrow[r, hook, bend right] \arrow[dr] & \mathrm{UFam}(\mathrm{Asm}(V)) \arrow[l, bend right] \arrow[r, "\simeq"] \arrow[d] & \mathrm{Asm}(V)^{\to} \arrow[dl] \\
 & \mathrm{Asm}(V) &
\end{tikzcd}$$

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

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

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 $[\![\Gamma \vdash A : \mathrm{Kind}]\!]^{\mathrm{Kinds}} = ((A_x, E_{A_x}))_{x \in X}$ (a family of assemblies indexed over the assembly (X, E)), 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 parts of the interpretation of kinds:

- the inclusion of types into kinds is modeled via the inclusion from complete pers into assemblies
- 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 $\nabla(\mathit{Obj}(\mathrm{CPer}(V))$, where $\mathit{Obj}(\mathrm{CPer}(V))$ is the set of all chain-complete pers over V.
- Prop is modeled by $\nabla \mathsf{P}(\mathrm{H}(V))$ (see the next subsection).
- Π^K 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

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

– Σ^K is modeled by dependent sum.
– External equality of kinds is interpreted by equality in the model.

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 $(\{(in_1(*), in_1(*))\})$.
– Π^T is modeled by dependent product.
– Σ^T 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 $(B_x)_{x \in X}$, where B_x is

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

Here $\mathcal{CC}(R)$ denotes the chain completion of R (the reflection into UFam (CPer(V)), cf. Lemma 1). 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. This is why these sums are only weak. 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.

The following 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.

Lemma 2. *For any type $\Gamma \vdash \sigma : \mathrm{Type}$, no per in the family $[\![\Gamma \vdash \sigma : \mathrm{Type}]\!]^{\mathrm{Types}}$ relates $\bot$ to itself.*

We omit the description of the interpretation of pure terms. Suffice it to say that lambda abstractions in the calculus really are interpreted via continuous functions (realizers from $V \to V$).

We say that an external equality judgment of terms $\Gamma \vdash M = N : A$ *holds* if M and N are interpreted as the same morphism. The soundness corollary 1 means that any derivable external equality judgment of terms holds.

Logic. As in separation logic, we really have a logic of heaps and hence propositions will be modeled as subsets of $\mathrm{H}(V)$. We obtain the structure needed for interpreting the logic as follows. The power set of heaps $\mathsf{P}(\mathrm{H}(V))$ ordered by inclusion is a BI-algebra [21] in Set. We embed it into $\mathrm{Asm}(V)$ via the functor ∇ to get $\nabla(\mathsf{P}(\mathrm{H}(V)))$. One can show that the object is an internally complete BI-algebra in $\mathrm{Asm}(V)$. Hence, as explained in [5], 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. Hence, a proposition in context $\Gamma \vdash P : \mathrm{Prop}$ is interpreted as follows: Suppose Γ is interpreted as the assembly (X, E). Then P is interpreted as a function from X to $\mathsf{P}(\mathrm{H}(V))$. The propositional connectives are all interpreted in the standard way from separation logic. For instance, $[\![\Gamma \vdash P * Q : \mathrm{Prop}]\!]_x^{\mathrm{Props}}$ is $\{h \mid \exists h_1 \in [\![\Gamma \vdash P : \mathrm{Prop}]\!]_x^{\mathrm{Props}}, h_2 \in [\![\Gamma \vdash Q : \mathrm{Prop}]\!]_x^{\mathrm{Props}} \ . \ h = h_1 * h_2\}$. The quantifiers are also interpreted in the standard way. For instance,

$$[\![\Gamma \vdash \forall y : A.P : \mathrm{Prop}]\!]_x^{\mathrm{Props}} = \{h \mid \forall y \in [\![\Gamma \vdash A : \mathrm{Kind}]\!]_x^{\mathrm{Kinds}} \ . \ h \in [\![\Gamma, y : A \vdash P]\!]_{(x,y)}^{\mathrm{Props}}\}$$

In the display above, note that $[\![\Gamma \vdash A : \mathrm{Kind}]\!]^{\mathrm{Kinds}}$ is a uniform family of assemblies over (X, E), so $[\![\Gamma \vdash A : \mathrm{Kind}]\!]_x^{\mathrm{Kinds}}$ 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 *depend* on x (we have a separation logic for a *dependent* type theory).

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

Computations. As mentioned in Section 1, a computation type $(\Delta).\{P\}x : \tau\{Q\}$ is modeled as an admissible per of realizers in $\mathrm{T}(V)$, which 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

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

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. This does not support the law of conjunctivity. 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

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

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 3. *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 with its field inside* $\mathrm{T}(V)$*, relating* $in_{\mathrm{T}}(\lambda h \ . \ \bot)$ *to itself. As such it is an admissible per over* $\mathrm{T}(V)$*.*

As mentioned in the introduction, we require that computations should produce heaps with equal support (given suitable heaps with equal support) so that allocation can 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 [25,24,3], 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 $[\![\Gamma]\!]^{\text{Ctxs}} = (X, E)$ and that $[\![\Gamma \vdash M : (\Delta).\{P\}x : \tau\{Q\}]\!]^{\text{Terms}}$ is realized by α. Then $[\![\Gamma \vdash M : (\Delta).\{P * R\}x : \tau\{Q * R\}]\!]^{\text{Terms}}$ is also realized by α since, for all $x \in X$, the field of $[\![\Gamma \vdash (\Delta).\{P\}x : \tau\{Q\} : \text{Type}]\!]^{\text{Types}}_x$ is included in the field of $[\![\Gamma \vdash (\Delta).\{P * R\}x : \tau\{Q * R\} : \text{Type}]\!]^{\text{Types}}_x$ (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 $[\![\Gamma]\!]^{\text{Ctxs}} = (X, E)$ 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

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

Note that the realizers for computations are as one would hope. Consider, for example, lookup $!M$, whose realizer is $\lambda e\ .\ \lambda h\ .\ \texttt{if } h(m(e)) = \bot \texttt{ then err else } (h(m(e)), h)$. 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 $[\![\Gamma, f\!:\!\Pi^T\ y\!:\!\sigma\ .\ (\Delta).\{P\}x\!:\!\tau\{Q\}, y\!:\!\sigma \vdash M\!:\!(\Delta).\{P\}x\!:\!\tau\{Q\}]\!]^{\text{Terms}}$.

Theorem 2. *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 $1 \to [\![\Gamma \vdash (\Delta).\{P\}x : \tau\{Q\} : \text{Type}]\!]^{\text{Types}}$ in the fibre over $[\![\Gamma]\!]^{\text{Ctxs}}$. Moreover, the external equality rules for computations hold.*

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

4 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.

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 [5]. Future work includes investigating how to model recursive types, as needed for the specification of programs that recurse through the store [22]. 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. 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. Appel, A., Melliìees, P.-A., Richards, C., Vouillon, J.: A very modal model of a modern, major, general type system. In: POPL 2007 (2007)
2. Barnett, M., Leino, K.R.M., Schulte, W.: The Spec# programming system: An overview. In: Barthe, G., Burdy, L., Huisman, M., Lanet, J.-L., Muntean, T. (eds.) CASSIS 2004. LNCS, vol. 3362, Springer, Heidelberg (2005)
3. Benton, N., Leperchey, B.: Relational reasoning in a nominal semantics for storage. In: Urzyczyn, P. (ed.) TLCA 2005. LNCS, vol. 3461, pp. 88–101. Springer, Heidelberg (2005)

4. Berger, M., Honda, K., Yoshida, N.: A logical analysis of aliasing in imperative higher-order functions. In: Danvy, O., Pierce, B.C. (eds.) ICFP 2005, Tallinn, Estonia, September 2005, pp. 280–293 (2005)
5. Biering, B., Birkedal, L., Torp-Smith, N.: Bi hyperdoctrines and higher-order separation logic. In: Sagiv, M. (ed.) ESOP 2005. LNCS, vol. 3444, pp. 233–247. Springer, Heidelberg (2005)
6. Biering, B., Birkedal, L., Torp-Smith, N.: BI hyperdoctrines, Higher-Order Separation Logic, and Abstraction. In: TOPLAS 2007 (to appear, 2007)
7. Birkedal, L., Møgelberg, R., Petersen, R.: Domain-theoretic models of parametric polymorphism. In: TCS (to appear, 2007)
8. Birkedal, L., Torp-Smith, N., Yang, H.: Semantics of separation-logic typing and higher-order frame rules for algol-like languages. LMCS 2(5:1), 1–33 (2006)
9. Birkedal, L., Yang, H.: Relational parametricity and separation logic. In: Seidl, H. (ed.) FOSSACS 2007. LNCS, vol. 4423, Springer, Heidelberg (2007)
10. Burdy, L., Cheon, Y., Cok, D., Ernst, M., Kiniry, J., Leavens, G.T., Leino, K.R.M., Poll, E.: An overview of JML tools and applications. International Journal on Software Tools for Technology Transfer 7(3), 212–232 (2005)
11. Detlefs, D.L., Leino, K.R.M., Nelson, G., Saxe, J.B.: Extended static checking. Compaq Systems Research Center, Research Report 159 (December 1998)
12. Evans, D., Larochelle, D.: Improving security using extensible lightweight static analysis. IEEE Software 19(1), 42–51 (2002)
13. Jacobs, B.: Categorical Logic and Type Theory. Studies in Logic and the Foundations of Mathematics, vol. 141. Elsevier, Amsterdam (1999)
14. Krishnaswami, N.: Separation logic for a higher-order typed language. In: SPACE 2006, pp. 73–82 (2006)
15. Krishnaswami, N., Aldrich, J., Birkedal, L.: Modular verification of the subject-observer pattern via higher-order separation logic. In: FTfJP (2007)
16. Morrisett, G., Walker, D., Crary, K., Glew, N.: From System F to typed assembly language. ACM TPLS 21(3), 527–568 (1999)
17. Nanevski, A., Ahmed, A., Morrisett, G., Birkedal, L.: Abstract Predicates and Mutable ADTs in Hoare Type Theory. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 189–204. Springer, Heidelberg (2007)
18. Nanevski, A., Morrisett, G., Birkedal, L.: Polymorphism and separation in Hoare Type Theory. In: ICFP 2006, Portland, Oregon, pp. 62–73 (2006)
19. O'Hearn, P.W., Yang, H., Reynolds, J.C.: Separation and information hiding. In: POPL 2004, pp. 268–280 (2004)
20. Petersen, R., Birkedal, L., Nanevski, A., Morrisett, G.: A realizability model of impredicative hoare type theory. Technical report, IT University of Copenhagen (2007), `http://www.itu.dk/people/birkedal/papers/httmodel-tr.pdf`
21. Pym, D.: The Semantics and Proof Theory of the Logic of Bunched Implications. Applied Logics Series, vol. 26. Kluwer, Dordrecht (2002)
22. Reus, B., Schwinghammer, J.: Separation Logic for Higher-Order Store. In: Ésik, Z. (ed.) CSL 2006. LNCS, vol. 4207, pp. 575–590. Springer, Heidelberg (2006)
23. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: LICS 2002, pp. 55–74 (2002)
24. Shinwell, M.: The Fresh Approach: Functional Programming with Names and Binders. PhD thesis, Computer Laboratory, Cambridge University (December 2004)
25. Shinwell, M.R., Pitts, A.M.: On a monadic semantics for freshness. TCS 342, 28–55 (2005)

26. Watkins, K., Cervesato, I., Pfenning, F., Walker, D.: A concurrent logical framework: The propositional fragment. In: Filliâtre, J.-C., Paulin-Mohring, C., Werner, B. (eds.) TYPES 2004. LNCS, vol. 3839, pp. 355–377. Springer, Heidelberg (2006)
27. Xi, H., Pfenning, F.: Dependent types in practical programming. In: POPL 1999, San Antonio, pp. 214–227 (1999)
28. Yoshida, N., Honda, K., Berger, M.: Local state in hoare logic for imperative higher-order functions. In: Seidl, H. (ed.) FOSSACS 2007. LNCS, vol. 4423, Springer, Heidelberg (2007)