Partiality, State and Dependent Types

Kasper Svendsen[1], Lars Birkedal[1], and Aleksandar Nanevski[2]

[1] IT University of Copenhagen {kasv,birkedal}@itu.dk
[2] IMDEA Software aleks.nanevski@imdea.org

Abstract. Partial type theories allow reasoning about recursively-defined computations using fixed-point induction. However, fixed-point induction is only sound for admissible types and not all types are admissible in sufficiently expressive dependent type theories.

Previous solutions have either introduced explicit admissibility conditions on the use of fixed points, or limited the underlying type theory. In this paper we propose a third approach, which supports Hoare-style partial correctness reasoning, without admissibility conditions, but at a tradeoff that one cannot reason equationally about effectful computations. The resulting system is still quite expressive and useful in practice, which we confirm by an implementation as an extension of Coq.

1 Introduction

Dependent type theories such as the Calculus of Inductive Constructions [2] provide powerful languages for integrated programming, specification, and verification. However, to maintain soundness, they typically require all computations to be pure and terminating, severely limiting their use as general purpose programming languages.

Constable and Smith [9] proposed adding partiality by introducing a type $\bigcirc\tau$ of potentially non-terminating computations of type τ, along with the following fixed point principle for typing recursively defined computations:

$$\text{if } M : \bigcirc\tau \to \bigcirc\tau \quad \text{then} \quad \boldsymbol{fix}(M) : \bigcirc\tau$$

Unfortunately, in sufficiently expressive dependent type theories, there exists types τ for which the above fixed point principle is unsound [10]. For instance, in type theories with subset-types, the fixed point principle allows reasoning by a form of fixed point induction, which is only sound for admissible predicates (a predicate is admissible if it holds for the limit whenever it holds for all finite approximations). Previous type theories based on the idea of partial types which admit fixed points have approached the admissibility issue in roughly two different ways:

1. The notion of admissibility is axiomatized in the type theory and explicit admissibility conditions are required in order to use $\boldsymbol{fix}$. This approach has, e.g., been investigated by Crary in the context of Nuprl [10]. The resulting type theory is expressive, but admissibility conditions lead to significant proof obligations, in particular, when using Σ types.

2. The underlying dependent type theory is restricted in such a way that one can only form types that are trivially admissible. This approach has, e.g., been explored in recent work on Hoare Type Theory (HTT) [21]. The restrictions exclude usage of subset types and Σ types, which are often used for expressing properties of computations and for modularity. Another problem with this approach is that since it limits the underlying dependent type theory one cannot easily implement it as a simple extension of existing implementations.

In this paper we explore a third approach, which ensures that all types are admissible, not by limiting the underlying standard dependent type theory, but by limiting only the partial types. The limitation on partial types consists of equating all effectful computations at a given type: if M and N are both of type $\bigcirc\tau$, then they are propositionally equal. Thus, with this approach, the only way to reason about effectful computations is through their type, rather than via equality or predicates. With sufficiently expressive types, the type of an effectful computation can serve as a partial correctness specification of the computation. Our hypothesis is that this approach allows us to restrict attention to a subset of admissible types, which is closed under the standard dependent type formers and which suffices for reasoning about partial correctness.

To demonstrate that this approach scales to expressive type theories and to effects beyond partiality, we extend the Calculus of Inductive Constructions (CIC) [2] with stateful and potentially non-terminating computations. Since reasoning about these effectful computations is limited to their type, our partial types are further refined into a Hoare-style partial correctness specifications, and have the form $\boldsymbol{ST}\,\tau\,(P, Q)$, standing for computations with pre-condition P, post-condition Q, that diverge or terminate with a value of type τ.

The resulting type theory is an impredicative variant of Hoare Type Theory [17], which differs from previous work on Hoare Type Theory in the scope of features considered and the semantic approach. In particular, this paper is the first to clarify semantically the issue of admissibility in Hoare Type Theory.

Impredicative Hoare Type Theory (iHTT) features the universes of propositions (***prop***), small types (***set***), and large types (***type***), with ***prop*** included in ***set***, ***set*** included in ***type***, and axioms ***prop***:***type*** and ***set***:***type***. The ***prop*** and ***set*** universes are impredicative, while ***type*** is predicative. There are two main challenges in building a model to justify the soundness of iHTT: (1) achieving that Hoare types are small ($\boldsymbol{ST}\ \tau\ s : \boldsymbol{set}$), which enables *higher-order store*; that is, storing side-effectful computations into the heap, and (2) supporting arbitrary Σ types, and more generally, inductive types. In this respect iHTT differs from the previous work on Hoare Type Theory, which either lacks higher-order store [19], lacks strong Σ types [21], or whose soundness has been justified using specific syntactic methods that do not scale to fully general inductive definitions [17, 18].

The model is based on a standard realizability model of partial equivalence relations (PERs) and assemblies over a combinatory algebra A. These give rise to a model of the Calculus of Constructions [14], with ***set*** modelled using PERs. Restricting PERs to complete PERs (i.e., PERs closed under limits of ω-chains)

over a suitable universal domain, allows one to model recursion in a simply-typed setting [4], or in a dependently-typed setting, but without strong Σ types [21].

Our contribution is in identifying a set of complete *monotone* PERs that are closed under Σ types and Hoare types. Complete PERs do not model Σ types because, given a chain of dependent pairs, in general, due to dependency, the second components of the chain are elements of *distinct* complete PERs. To apply completeness, we need a fixed single complete PER. Monotonicity will equate the first components of the chain and give us the needed single complete PER for the second components. Monotonicity further forces a trivial equality on Hoare types, equating all effectful computations satisfying a given specification. However, it does not influence the equality on the total, purely functional, fragment of iHTT, ensuring that we still model CIC. This is sufficient for very expressive Hoare-style reasoning, and avoids admissibility conditions on the use of $\boldsymbol{fix}$.

As iHTT is an extension of CIC, we have implemented iHTT as an axiomatic extension of Coq [1], available at: `http://www.itu.dk/people/kasv/ihtt.tgz`. The implementation is carried out in Ssreflect [12] (a recent extension of Coq), based on the previous implementation of predicative Hoare Type Theory [19].

Details and proofs can be found in the accompanying technical report [23].

2 Hoare types by example

To illustrate Hoare types, we sketch a specification of a library for arrays in iHTT. We assume that array indexes range over a finite type $\iota{:}\boldsymbol{set}_{fin}$, that the elements of ι can be enumerated as $\iota_0, \iota_1, \ldots, \iota_n$, and that equality between these elements can be decided by a function $=\!=\colon \iota \to \iota \to \boldsymbol{bool}$.

Each array is implemented as a contiguous block of locations, each location storing a value from the range type $\tau{:}\boldsymbol{set}$. The space occupied by the array is uniquely determined by ι, τ, and the pointer to the first element, justifying that the $\boldsymbol{array}$ type be defined as this first pointer.

$$\boldsymbol{array} : \boldsymbol{set}_{fin} \to \boldsymbol{set} \to \boldsymbol{set} = \lambda\iota.\, \lambda\tau.\, \boldsymbol{ptr}.$$

Here, $\boldsymbol{ptr}$ is the type of pointers, which we assume isomorphic to $\boldsymbol{nat}$. Each array is essentially a stateful implementation of some finite function $f{:}\iota{\to}\tau$. To capture this, we define a predicate indexed by f, that describes the layout of an array in the heap.

$$\begin{aligned}&\mathsf{shape} : (\boldsymbol{array}\ \iota\ \tau) \to (\iota \to \tau) \to \boldsymbol{heap} \to \boldsymbol{prop} = \\ &\qquad \lambda a.\, \lambda f.\, \lambda h.\, h = a \mapsto f\,\iota_o \bullet a{+}1 \mapsto f\,\iota_1 \bullet \cdots \bullet a{+}n \mapsto f\,\iota_n.\end{aligned}$$

In other words, h stores an array a, representing a finite function f, if $\mathsf{shape}\ a\ f\ h$ holds, that is, if h consists of $n{+}1$ consecutive locations a, $a{+}1$, …, $a{+}n$, storing $f\,\iota_0, f\,\iota_1, \ldots, f\,\iota_n$, respectively. The property is stated in terms of singleton heaps $a{+}k \mapsto f\,\iota_k$, connected by the operator $\bullet$ for *disjoint* heap union. Later in the text, we will also require a constant empty denoting the empty heap.

The type of arrays comes equipped with several methods for accessing and manipulating the array elements. For example, the method for reading the value at index $k{:}\iota$ can be given the following type.

$$\begin{array}{l}\mathsf{read} : \mathit{\Pi} a{:}\boldsymbol{\mathit{array}}\ \iota\ \tau.\ \mathit{\Pi} k{:}\iota. \\ \qquad \boldsymbol{ST}\ \tau\ (\lambda h.\, \exists f.\, \mathsf{shape}\ a\ f\ h, \\ \qquad\qquad \lambda r.\, \lambda h.\, \lambda m.\, \forall f.\, \mathsf{shape}\ a\ f\ h \to r = f\, k \wedge m = h)\end{array}$$

Informally, $\mathsf{read}\ a\ k$ is specified as a stateful computation whose precondition permits the execution only in a heap h which stores a valid array at address a ($\exists f.\, \mathsf{shape}\ a\ f\ h$). The postcondition, on the other hand, specifies the result of executing $\mathsf{read}\ a\ k$ as a relation between output result $r{:}\tau$, input heap h and output heap m. In particular, the result r is indeed the array value at index k ($r = f\, k$), and the input heap is unchanged ($m = h$).

Unlike in ordinary Hoare logic, but similar to VDM [6], our postcondition is parametrized wrt. both input and the output heaps in order to directly express the relationship between the two. In particular, when this relationship depends on some specification-level value, such as f above, the dependency can be expressed by an ordinary propositional quantification.

Hoare types employ *small footprint* specifications, as in separation logic [20], whereby the specifications only describe the parts of the heap that the computation traverses. The untraversed parts are by default invariant. To illustrate, consider the type for the method new that generates a fresh array, indexed by ι, and populated by the value $x{:}\tau$.

$$\mathsf{new} : \mathit{\Pi} x{:}\tau.\ \boldsymbol{ST}\ (\boldsymbol{\mathit{array}}\ \iota\ \tau)\ (\lambda h.\, h = \mathsf{empty}, \lambda a.\, \lambda h.\, \lambda m.\, \mathsf{shape}\ a\ (\lambda z.\, x)\ m)$$

The type states *not* that $\mathsf{new}\ x$ can only run in an empty heap, but that $\mathsf{new}\ x$ *changes* the empty subheap of the current heap into a heap m containing an array rooted at a and storing all x's. In other words, new is *adding* fresh pointers, and the resulting array a itself is fresh. On the other hand, unlike in separation logic, we allow that the specifications can directly use and quantify over variables of type ***heap***. For completeness, we next simply list without discussion the types of the other methods for arrays.

$$\begin{array}{l}\mathsf{new\_from\_fun} : \mathit{\Pi} f{:}\iota{\to}\tau.\ \boldsymbol{ST}\ (\boldsymbol{\mathit{array}}\ \iota\ \tau)\ (\lambda h.\, h = \mathsf{empty}, \lambda a\, h\, m.\, \mathsf{shape}\ a\ f\ m) \\ \mathsf{free} : \mathit{\Pi} a{:}\boldsymbol{\mathit{array}}\ \iota\ \tau.\ \boldsymbol{ST}\ \boldsymbol{\mathit{unit}}\ (\lambda h.\, \exists f.\, \mathsf{shape}\ a\ f\ h, \lambda r\, h\, m.\, m = \mathsf{empty}) \\ \mathsf{write} : \mathit{\Pi} a{:}\boldsymbol{\mathit{array}}\ \iota\ \tau.\ \mathit{\Pi} k{:}\iota.\ \mathit{\Pi} x{:}\tau. \\ \qquad \boldsymbol{ST}\ \boldsymbol{\mathit{unit}}\ (\lambda h.\, \exists f.\, \mathsf{shape}\ a\ f\ h, \lambda r\, h\, m.\, \forall f.\, \mathsf{shape}\ a\ f\ h \to \\ \qquad\qquad\qquad\qquad \mathsf{shape}\ a\ (\lambda z.\, \mathsf{if}\ z == k\ \mathsf{then}\ x\ \mathsf{else}\ f(z))\ m)\end{array}$$

At this point, we emphasize that various type theoretic abstractions are quite essential for practical work with Hoare types. The usefulness of $\mathit{\Pi}$ types and the propositional quantifiers is apparent from the specification of the array methods. But the ability to structure specification is important too. For example, we can pair pre- and postconditions into a type $\boldsymbol{spec}\ \tau = (\boldsymbol{heap} \to \boldsymbol{prop}) \times (\tau \to \boldsymbol{heap} \to \boldsymbol{heap} \to \boldsymbol{prop})$, which is then used to specify the fixpoint combinator.

$$\begin{array}{l}\mathsf{fix} : \mathit{\Pi} \alpha{:}\boldsymbol{set}.\, \mathit{\Pi} \beta{:}\alpha{\to}\boldsymbol{set}.\, \mathit{\Pi} s{:}\mathit{\Pi} x.\ \boldsymbol{spec}\ (\beta\ x). \\ \qquad (\mathit{\Pi} x.\ \boldsymbol{ST}\ (\beta\ x)\ (s\ x) \to \mathit{\Pi} x.\ \boldsymbol{ST}\ (\beta\ x)\ (s\ x)) \to \mathit{\Pi} x.\ \boldsymbol{ST}\ (\beta\ x)\ (s\ x).\end{array}$$

Structuring proofs and specifications with programs is also necessary, and is achieved using dependent records (i.e., Σ types), which we illustrate next.

The first example of a dependent record is the $\boldsymbol{set}_{fin}$ type. This is an algebraic structure containing the carrier type σ, the operation $==$ for deciding equality on σ, and a list enumerating σ's elements. Additionally, $\boldsymbol{set}_{fin}$ needs proofs that $==$ indeed decides equality, and that the enumeration list contains each element exactly once. Using the record notation $[x_1{:}\tau_1, \ldots x_n{:}\tau_n]$ instead of the more cumbersome $\Sigma x_1{:}\tau_1 \ldots \Sigma x_n{:}\tau_n.\, 1$, the $\boldsymbol{set}_{fin}$ type is defined as follows.

$$\begin{aligned}\boldsymbol{set}_{fin} = [\,&\sigma : \boldsymbol{set}, \mathsf{enum} : \boldsymbol{list}\ \sigma, == : \sigma \to \sigma \to \mathsf{bool},\\ &\mathsf{eqp} : \forall x\ y{:}\sigma.\ x == y = \mathsf{true} \iff x = y,\\ &\mathsf{enump} : \forall x{:}\sigma.\ \mathsf{count}\ x\ \mathsf{enum} = 1]\end{aligned}$$

The above dependent record refines a type. In practice, we will also use records that refine *values*. For example, in programming, arrays are often indexed by the type of bounded integers $I_n = [x : \boldsymbol{nat}, \mathsf{boundp} : x \leq n]$. I_n can be extended with appropriate fields, to satisfy the specification for $\boldsymbol{set}_{fin}$, but the important point here is that the elements of I_n are dependent records containing a number x and a proof that $x \leq n$. Of course, during actual execution, this proof can be ignored (proofs are computationally irrelevant), but it is clearly important statically, during verification.

Finally, the library of arrays itself can be ascribed a signature which will serve as an interface to the client programs. This signature too is a dependent record, providing types for all the array methods. Just as in the case of $\boldsymbol{set}_{fin}$ and I_n, the signature may also include properties, similar to object invariants [13, 15]. For example, we have found it useful in practice to hide from the clients the definitions of the array type and the array shape predicate, but expose that two arrays in stable states; that is, between two method calls, stored in compatible heaps must be equal (i.e., that the shape predicate is "functional"):

$$\begin{aligned}\mathsf{functional} : \Pi\mathsf{shape}.\, \forall a_1\ a_2\ f_1\ f_2\ h_1\ h_2.\, \mathsf{shape}\ a_1\ f_1\ h_1 \to \mathsf{shape}\ a_2\ f_2\ h_2 \to\\ (\exists j_1\ j_2.\, h_1 \bullet j_1 = h_2 \bullet j_2) \to a_1 = a_2 \wedge f_1 = f_2 \wedge h_1 = h_2.\end{aligned}$$

Then the signature for arrays indexed by ι, containing values of type τ, is provided by the following dependent record parametrized by ι and τ.

$$\begin{aligned}\mathit{ArraySig} = \Pi\iota{:}\boldsymbol{set}_{fin}.\, \Pi\tau{:}\boldsymbol{set}.\\ [\,\boldsymbol{array} : \boldsymbol{set},\ \mathsf{shape} : \boldsymbol{array} \to (\iota \to \tau) \to \boldsymbol{heap} \to \boldsymbol{prop},\\ \mathsf{funcp} : \mathsf{functional\ shape},\ \mathsf{read} : \Pi a{:}\boldsymbol{array}.\, \Pi k{:}\iota.\ \ldots].\end{aligned}$$

Therefore, Σ types are central for building verified libraries of programs, specifications and proofs.

3 Semantics

In presenting the model, we first focus on the $\boldsymbol{set}$-universe, and then scale up to cover all of iHTT. Since the purely-function fragment of iHTT is terminating, we

take our universe of realizers to be a universal *pre-domain* with a suitable sub-domain for modelling stateful and potentially non-terminating computations.

Definition 1. *Let $\mathbb{V}$ denote a pre-domain satisfying the following recursive pre-domain equation:*

$$\mathbb{V} \cong 1 + \mathbb{N} + (\mathbb{V} \times \mathbb{V}) + (\mathbb{V} \to_c \mathbb{V}_\perp) + T(\mathbb{V}) + H(\mathbb{V})$$

where $\to_c$ is the space of continuous functions and

$$T(\mathbb{V}) \stackrel{def}{=} H(\mathbb{V}) \to_c ((\mathbb{V} \times H(\mathbb{V})) + 1)_\perp$$
$$H(\mathbb{V}) \stackrel{def}{=} \{h : \mathbf{ptr} \to_c \mathbb{V}_\perp \mid supp(h) \text{ finite} \wedge h(\mathbf{null}) = \perp\}$$
$$supp(h : \mathbf{ptr} \to \mathbb{V}_\perp) \stackrel{def}{=} \{l \in \mathbf{ptr} \mid f(l) \neq \perp\}$$

The first four summands of $\mathbb{V}$ model the underlying dependent type theory, and $T(\mathbb{V})$ and $H(\mathbb{V})$ model computations and heaps, respectively. The ordering on $T(\mathbb{V})$ is the standard pointwise order and the ordering on $H(\mathbb{V})$ is as follows:

$$h_1 \leq h_2 \quad \textit{iff} \quad supp(h_1) = supp(h_2) \wedge \forall n \in supp(h_1).\ h_1(n) \leq h_2(n)$$

Let $in_1, in_{\mathbb{N}}, in_\times, in_\to, in_T$, and in_H denote injections into $\mathbb{V}$ corresponding to each of the above summands.

$\mathbb{V}$ defines a partial combinatory algebra with the following partial application operator:

Definition 2. *Let $\cdot : \mathbb{V} \times \mathbb{V} \rightharpoonup \mathbb{V}$ denote the function,*

$$a \cdot b = \begin{cases} f(b) & \text{if } a = in_\to(f) \wedge f(b) \neq \perp \\ \mathbf{undef} & \text{otherwise} \end{cases}$$

We recall some notation and definitions. If $R \subseteq A \times A$ is a PER then its domain, denoted $|R|$, is $\{x \in A \mid (x, x) \in R\}$. If $R, S \subseteq A \times A$ are PERs, then $R \to S$ is the PER $\{(\alpha, \beta) \in A \times A \mid \forall x, y \in A.\ (x, y) \in R \Rightarrow (\alpha \cdot x, \beta \cdot y) \in S\}$. If $R \subseteq A \times A$ is a PER and $f : A \to B$ then $f(R)$ denotes the PER $\{(f(x), f(y)) \mid (x, y) \in R\} \subseteq B \times B$. For a subset $X \subseteq A$, we use $\Delta(X)$ to denote the PER $\{(x, y) \mid x \in X \wedge y \in X\}$. Lastly, if $R \subseteq A \times A$ is a PER, we use $[R]$ to denote the set of R equivalence classes.

Definition 3 (Per(A)). *The category of PERs, Per(A), over a partial combinatory algebra $(A, \cdot)$, has PERs over A as objects. Morphisms from R to S are set-theoretic functions $f : [R] \to [S]$, such that there exists a realizer $\alpha \in A$ such that,*

$$\forall e \in |R|.\ [\alpha \cdot e]_S = f([e]_R)$$

Per($\mathbb{V}$) is cartesian closed and thus models simple type theory. To model recursion, note that a realized set-theoretic function is completely determined by its realizers (i.e., $\mathrm{Per}(\mathbb{V})(R, S) \cong [R \to S]$) and that we have the standard least fixed-point operator on the sub-domain of computations of $\mathbb{V}$. This lifts to a least fixed-point operator on those PERs that are admissible on the sub-domain of computations:

Definition 4.

1. *A PER $R \subseteq A \times A$ on a pre-domain A is* complete *if, for all chains $(c_i)_{i\in\mathbb{N}}$ and $(d_i)_{i\in\mathbb{N}}$ such that $(c_i, d_i) \in R$ for all i, also $(\sqcup_i c_i, \sqcup_i d_i) \in R$.*
2. *A PER $R \subseteq A \times A$ on a domain A is* admissible *if it is complete and $\bot \in |R|$.*

Let $\mathrm{CPer}(A)$ *and* $\mathrm{AdmPer}(A)$ *denote the full sub-categories of* $\mathrm{Per}(A)$ *consisting of complete PERs and admissible PERs, respectively.*

Definition 5. *Define $u : \mathbb{V} \to_c (T(\mathbb{V}) \to_c T(\mathbb{V}))$ as follows,*

$$u(x)(y) \stackrel{def}{=} \begin{cases} z & \text{if } x \cdot in_T(y) = in_T(z) \\ \bot & \text{otherwise} \end{cases}$$

and let lfp denote the realizer $in_\to(\lambda x.\ [in_T(\sqcup_n (u(x))^n)])$.

Lemma 1. *Let $R \in \mathrm{AdmPer}(T(\mathbb{V}))$, then $lfp \in |(in_T(R) \to in_T(R)) \to in_T(R)|$ and for all $\alpha \in |in_T(R) \to in_T(R)|$, $\alpha \cdot (lfp \cdot \alpha) = lfp \cdot \alpha$.*

The above development is standard and suffices to model fixed points over partial types in a non-stateful, simply-typed setting. However, it does not extend directly to a stateful dependently-typed setting: Assume $\vdash \tau : \textbf{\textit{type}}$ and $x : \tau \vdash \sigma : \textbf{\textit{type}}$. Then τ is interpreted as a PER $R \in \mathrm{Per}(\mathbb{V})$, and σ as an $[R]$-indexed family of PERs $S \in [R] \to \mathrm{Per}(\mathbb{V})$, and $\vdash \Sigma x : \tau.\sigma : \textbf{\textit{type}}$ as the PER $\Sigma_R(S)$:

$$\Sigma_R(S) = \{(in_\times(a_1, b_1), in_\times(a_2, b_2)) \mid a_1\ R\ a_2 \wedge b_1\ S([a_1]_R)\ b_2\}.$$

In general, this PER is not chain-complete even if R and each S_x is: given a chain $(a_i, b_i)_{i\in\mathbb{N}}$, we do not know in general that $[a_i]_R = [a_j]_R$ and hence cannot apply the completeness of S_x to the chain $(b_i)_{i\in\mathbb{N}}$ for any $x \in [R]$.

To rectify this problem we impose the following monotonicity condition on the PERs, which ensures exactly that $a_i\ R\ a_j$, for all $i, j \in \mathbb{N}$, and hence that $\Sigma_R(S)$ is chain-complete.

Definition 6 ($\mathrm{CMPer}(A)$). *A PER $R \subseteq A \times A$ on a pre-domain A is* monotone *if, for all $x, y \in |R|$ such that $x \leq y$, we have $(x, y) \in R$. Let* $\mathrm{CMPer}(A)$ *denote the full sub-category of* $\mathrm{Per}(A)$ *consisting of complete monotone PERs.*

Restricting to complete monotone PERs forces a trivial equality on any particular Hoare type, as all of the elements of the type have to be equal to the diverging computation. However, it does not trivialize the totality of Hoare types, as we can still interpret each distinct Hoare type, $\boldsymbol{ST}\ \tau\ s$, as a distinct PER R, containing the computations that satisfy the specification s.

Restricting to complete monotone PERs does not collapse the equality on types in the purely functional fragment of iHTT, as these types are all interpreted as PERs without a bottom element in their domain. In particular, Π types, which are modelled as elements of $\mathbb{V} \to \mathbb{V}_\bot$, picks out only those elements that map to non-bottom on their domain.

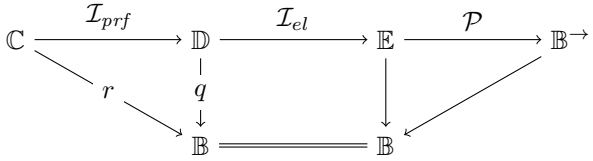
We shall see later that the monotonicity condition is also used to interpret partial types with a post-condition, which induces a dependency similar to that of Σ types, in the semantics of partial types.

3.1 iHTT

So far, we have informally introduced a PER model of a single dependent type universe extended with partial types. The next step is to scale the ideas to a model of all of iHTT, and to prove that we do indeed get a model of iHTT. We start by showing that CMPERs and assemblies form a model of the underlying dependent type theory. Next, we sketch the interpretation of iHTT specific features such as heaps and computations in the model. Lastly, we show that the model features W-types at both the ***set*** and ***type*** universe.

Underlying DTT. We begin by defining a general class of models for the dependent type theory underlying iHTT, and then present a concrete instance based on complete monotone PERs. To simplify the presentation and exploit existing categorical descriptions of models of the Calculus of Constructions, the model will be presented using the fibred approach of [14]. To simplify the definition of the interpretation function, we consider a split presentation of the model (i.e., with canonical choice of all fibred structure, preserved on-the-nose).

Definition 7 (Split iHTT structure). *A split iHTT structure is a structure*



such that (1) $\mathcal{P}$ is a split closed comprehension category, (2) $\mathcal{I}_{el}$ and $\mathcal{I}_{prf}$ are split fibred reflections, (3) the coproducts induced by the $\mathcal{I}_{el}$ reflection are strong (i.e., $\mathcal{P} \circ \mathcal{I}_{el}$ is a split closed comprehension category), and (4) there exists objects $\Omega_{el}, \Omega_{prf} \in \mathbb{E}_1$ such that $\{\Omega_{el}\}$ is a split generic object for the q fibration and $\{\Omega_{prf}\}$ is a split generic object for the r fibration, where $\{-\} = \mathrm{Dom} \circ \mathcal{P} : \mathbb{E} \to \mathbb{B}$.

The idea is to model contexts in $\mathbb{B}$, and the three universes, ***prop***, ***set***, and ***type*** in fibres of $\mathbb{C}$, $\mathbb{D}$ and $\mathbb{E}$, respectively. The split closed comprehension category structure models unit, Π and Σ types in the ***type*** universe. The split fibred reflections models the inclusion of ***prop*** into ***set*** and ***set*** into ***type*** and induces unit, Π and weak Σ types in ***prop*** and ***set***. Lastly, the split generic objects models the axioms ***prop*** : ***type*** and ***set*** : ***type***, respectively.

The concrete model we have in mind is mostly standard: the contexts and the ***type*** universe will be modelled with assemblies, the ***set*** universe with complete monotone PERs, and the ***prop*** universe with regular subobjects of assemblies. We begin by defining a category of uniform families of complete monotone PERs. Uniformity refers to the fact that each morphism is realized by a single $\alpha \in \mathbb{V}$.

Definition 8 ($\mathrm{UFam}(\mathrm{CMPer}(A))$**).**

- *Objects are pairs $(I, (S_i)_{i \in |I|})$ where $I \in \mathrm{Asm}(\mathbb{V})$ and each $S_i \in \mathrm{CMPer}(\mathbb{V})$.*

– *Morphisms from* (I, S_i) *to* (J, T_j) *are pairs* $(u, (f_i)_{i\in|I|})$ *where*

$$u : I \to J \in \mathrm{Asm}(\mathbb{V}) \qquad \textit{and} \qquad f_i : \mathbb{V}/S_i \to \mathbb{V}/T_{u(i)}$$

such that there exists an $\alpha \in \mathbb{V}$ *satisfying*

$$\forall i \in |I|.\ \forall e_i \in E_I(i).\ \forall e_v \in |S_i|.\ \alpha \cdot e_i \cdot e_v \in f_i([e_v]_{S_i})$$

Recall [14] that the standard category $\mathrm{UFam}(\mathrm{Per}(A))$ is defined in the same manner, using all PERs instead of only the complete monotone ones.

Let $\mathrm{RegSub}(\mathrm{Asm}(\mathbb{V}))$ denote the standard category of regular subobjects of assemblies and $\mathrm{UFam}(\mathrm{Asm}(\mathbb{V}))$ the standard category of uniform families of assemblies (see [14] for a definition). There is a standard split fibred reflection of $\mathrm{UFam}(\mathrm{Per}(A))$ into $\mathrm{UFam}(\mathrm{Asm}(A))$: the inclusion views a PER R as the assembly $([R], id_{[R]})$ [14]. This extends to a split fibred reflection of $\mathrm{UFam}(\mathrm{CMPer}(A))$ into $\mathrm{UFam}(\mathrm{Asm}(V))$ by composing with the following reflection of $\mathrm{UFam}(\mathrm{CMPer}(A))$ into $\mathrm{UFam}(\mathrm{Per}(A))$.

Lemma 2. *The inclusion* $\mathcal{I} : \mathrm{UFam}(\mathrm{CMPer}(A)) \to \mathrm{UFam}(\mathrm{Per}(A))$ *is a split fibred reflection.*

Proof (Sketch). We show that $\mathrm{CMPer}(A)$ is a reflective sub-category of $\mathrm{Per}(A)$; the same construction applies to uniform families, by a point-wise lifting. The left-ajoint, $\mathcal{R} : \mathrm{Per}(A) \to \mathrm{CMPer}(A)$ is given by monotone completion:

$$\mathcal{R}(S) = \overline{S} \qquad\qquad \mathcal{R}([\alpha]_{R\to S}) = [\alpha]_{\overline{R}\to\overline{S}}$$

where $\overline{R} \stackrel{\text{def}}{=} \bigcap\{S \in \mathrm{CMPer}(\mathbb{V}) \mid R \subseteq S\}$ for a PER $R \in \mathrm{Per}(\mathbb{V})$. Since the underlying realizers are continuous functions, for all $R, S \in \mathrm{Per}(A)$, $\overline{R} \to \overline{S} = R \to \overline{S}$, which induces the adjoint-isomorphism:

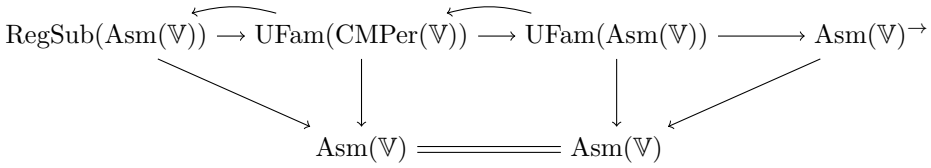
$$\mathrm{CMPer}(A)(\mathcal{R}(S), T) \cong [\overline{S} \to T] = [S \to T] \cong \mathrm{Per}(A)(S, \mathcal{I}(T))$$

for $S \in \mathrm{Per}(A)$ and $T \in \mathrm{CMPer}(A)$.

Lemma 3. *The coproducts induced by the* $\mathcal{I}_{el} : \mathrm{UFam}(\mathrm{CMPer}(\mathbb{V})) \to \mathrm{UFam}(\mathrm{Asm}(\mathbb{V}))$ *reflection are strong.*

Proof. The coproducts induced by $\mathcal{I}_{el}$ coincide with those induced by the $\mathrm{UFam}(\mathrm{Per}(\mathbb{V})) \to \mathrm{UFam}(\mathrm{Asm}(\mathbb{V}))$ reflection, which are strong [14, Section 10.5.8].

Theorem 1. *The diagram below forms a split iHTT structure.*



Heaps. Pre- and post-conditions in iHTT are expressed as predicates over a type ***heap***, of heaps, which supports basic operations such as update, which stores a value at a given location, and peek, which reads the value at a given location. ***heap*** is a large type (i.e., ***heap*** : ***type***) and the types of update and peek are as follows.

$$\begin{aligned} \textbf{\textit{upd}} &: \Pi\alpha : \textbf{\textit{set}}.\ \textbf{\textit{heap}} \to \textbf{\textit{ptr}} \to \alpha \to \textbf{\textit{heap}} \\ \textbf{\textit{peek}} &: \textbf{\textit{heap}} \to \textbf{\textit{ptr}} \to 1 + \Sigma\alpha : \textbf{\textit{set}}.\ \alpha \end{aligned}$$

We use $h[m \mapsto_\tau v]$ as shorthand for $\textbf{\textit{upd}}\ \tau\ h\ m\ v$. These operations can be used to define the points-to ($\mapsto$) and disjoint union ($\bullet$) operations used in Section 2.

We would like to model the values of ***heap*** as elements of $H(\mathbb{V})$. However, with this interpretation of ***heap*** we cannot interpret the update operation, $h[m \mapsto_\tau v]$, as the definition would not be independent of the choice of realizer for v. Rather, we introduce a notion of a world, which gives a notion of heap equivalence and interpret a heap as a pair consisting of a world and an equivalence class of heaps in the given world. A world is a finite map from locations to small semantic types:

$$\mathbb{W} \stackrel{\text{def}}{=} \textbf{\textit{ptr}} \stackrel{\text{fin}}{\rightharpoonup} \mathrm{CMPer}(\mathbb{V})$$

and two heaps $h_1, h_2 \in H(\mathbb{V})$ are considered equivalent in a world $w \in \mathbb{W}$ iff their support equals the domain of the world and their values are point-wise related by the world:

$$h_1 \sim_w h_2 \quad \text{iff} \quad supp(h_1) = supp(h_2) = \mathrm{Dom}(w) \wedge \forall l \in \mathrm{Dom}(w).\ h_1(l)\ w(l)\ h_2(l)$$

A typed heap is then a pair consisting of a world and an equivalence class of domain-theoretic heaps,

$$\mathbb{H}_t \stackrel{\text{def}}{=} \coprod_{w \in \mathbb{W}} [\sim_w]$$

and ***heap*** is interpreted as the set of typed heaps with the underlying domain-theoretic heaps as realizers:

$$[\![\Gamma \vdash \textbf{\textit{heap}}]\!] = (\mathbb{H}_t, (w, U) \mapsto in_H(U))_{i \in |I|}$$

for $I = [\![\Gamma]\!]$. That is, for each i, we have the assembly with underlying set $\mathbb{H}_t$ and with realizability map $\mathbb{H}_t \to P(\mathbb{V})$ given by $(w, U) \mapsto in_H(U)$. The realizers themselves do not have to contain any typing information, as we interpret small types with trivial realizability information (i.e., $[\![\Gamma \vdash \textbf{\textit{set}}]\!] = \nabla(\mathrm{CMPer}(\mathbb{V}))$). We now sketch the interpretation of update and peek in the empty context. Let $(w, [h]) = [\![\vdash M]\!]$, $in_{\mathrm{N}}(n) \in [\![\vdash N_1]\!]$, $R = [\![\vdash \tau]\!]$, and $v \in [\![\vdash N_2]\!]$. Then,

$$\begin{aligned} [\![\vdash M[N_1 \mapsto_\tau N_2]]\!] &= (w[n \mapsto R], [h[n \mapsto v]]_{w[n \mapsto R]}) \\ [\![\vdash \textbf{\textit{peek}}\ M\ N_1]\!] &= \begin{cases} inr(w(n), [h(n)]_{w(n)}) & \text{if } w(n) \text{ defined} \\ inl(*) & \text{otherwise} \end{cases} \end{aligned}$$

In the interpretation of update, the world is used to ensure that the interpretation is independent of the choice of realizer v for N_2. Since the underlying domain-theoretic heaps do not contain any typing information, the world is also used in the interpretation of peek, to determine the type of the value stored at the given location.

Note that iHTT has "strong" update and that the world is modified to contain the new type (semantically, the per R) upon update. Thus our notion and use of worlds is different from the use of worlds in models of "weak" ML-like reference types, e.g. [5]; in particular, note that we do not index every type by a world, but only use worlds to interpret the type of heaps and the operations thereon (see also the next subsection for further discussion).

Computations. We are now ready to sketch the interpretation of Hoare-types. The idea is to interpret Hoare-types as PERs on elements of $T(\mathbb{V})$ that satisfy the given specification, but with a trivial equality. Specifically, given a partial correctness specification, we define an admissible subset $X \subseteq T(\mathbb{V})$ of computations satisfying the specification, and interpret the associated Hoare type as the PER,

$$R = in_T(\Delta(X)) = \{(in_T(f), in_T(g)) \mid f \in X \wedge g \in X\}$$

The trivial equality ensures that R is trivially monotone and admissibility on the sub-domain of computations follows from admissibility of X.

Assume a semantic pre-condition $P \in \mathbb{H}_t \to 2$, a small semantic type $R \in \mathrm{CMPer}(\mathbb{V})$, and a semantic post-condition $Q \in [R] \times \mathbb{H}_t \times \mathbb{H}_t \to 2$. As explained in the previous section, the pre- and post-condition is expressed in terms of a typed heaps, $\mathbb{H}_t$, instead of the underlying domain theoretic heaps. The subset of computations satisfying the specification is thus defined using the usual "forall initial worlds, there exists a terminal world"-formulation, known from models of ML-like references [5]. However, as iHTT supports strong update and deallocation, the terminal world is not required to be an extension of the initial world. Specifically, define $hoare(R, P, Q)$ as the following subset of $T(\mathbb{V})$:

$$\begin{aligned}\{f \in T(\mathbb{V}) \mid \forall w \in \mathbb{W}.\, \forall h \in |\sim_w|.\, P([h]_w) = \top \Rightarrow \\ (f(h) = \bot \ \vee \exists v', h'.\, f(h) = (v', h') \wedge v' \in |R| \wedge \\ h' \in \overline{\{h' \in H(\mathbb{V}) \mid \exists w' \in \mathbb{W}.\, Q([v']_R)([h]_w)([h']_{w'}) = \top\}})\}\end{aligned}$$

where $\overline{(-)}$ denotes the chain-completion operator on $T(\mathbb{V})$. The explicit chain-completion of the post-condition is required because of the existential quantification over worlds. Furthermore, since the post-condition is indexed by the return value $[v']_R$, monotonicity is used to collapse a chain of domain-theoretic values $v_1 \leq v_2 \leq \cdots$ into a single type-theoretic value $[v_1]_R$, when proving that $hoare(R, P, Q)$ is an admissible subset of $T(\mathbb{V})$.

A Hoare-type in the empty context is now interpreted as follows:

$$[\![\vdash \boldsymbol{st}_\tau\ (\mathrm{P}, \mathrm{Q})]\!] = in_T(\Delta(hoare([\![\vdash \tau]\!], [\![\vdash \mathrm{P}]\!], [\![\vdash \mathrm{Q}]\!])))$$

The previous model of iHTT [21] featured a non-trivial computational equality. However, the previous model lacked the worlds introduced in the previous section and with it a useful notion of heap equivalence. As a result, the computational equality in the previous model was very strict, rendering the structural rule for existentials in Hoare logic unsound. The new model validates all the usual structural rules of Hoare-logic.

Theorem 2. *The interpretation of computations is sound, i.e., well-typed computations satisfy their specifications.*

W-types. The presentation of (co)-inductive types in CIC is based on an intricate syntactic scheme of inductive families. So far, this presentation of (co)-inductive types has eluded a categorical semantics. Martin-Löf type theory features an alternative presentation, based on W-types (a type-theoretic formalization of well-founded trees), which is strong enough to represent a wide range of predicative inductive types in extensional models (such as ours) [11, 3]. Since W-types in addition have a simple categorical semantics, we have chosen to show that iHTT models inductive types by showing that it models W-types. Specifically, we show that it models W-types at both the ***type*** and ***set*** universe, and that the W-types at the ***set*** universe supports elimination over large types.

Semantically, in the setting of locally cartesian closed categories, W-types are modelled as initial algebras of polynomial functors [16]. In the setting of split closed comprehension categories we define:

Definition 9. *A split closed comprehension category $\mathcal{P} : \mathbb{E} \to \mathbb{B}^{\to}$ has split W-types, if for every $I \in \mathbb{B}$, $X \in \mathbb{E}_I$ and $Y \in \mathbb{E}_{\{X\}}$ the endo-functor,*

$$P_{X,Y} = \Sigma_X \circ \Pi_Y \circ (\pi_X \circ \pi_Y)^* : \mathbb{E}_I \to \mathbb{E}_I$$

has a chosen initial algebra $\alpha_{X,Y} : P_{X,Y}(W_{X,Y}) \to W_{X,Y} \in \mathbb{E}_I$, which is preserved on-the-nose by re-indexing functors.

As is well-known, $\mathrm{Asm}(\mathbb{V})$ is locally cartesian closed and all polynomial functors on $\mathrm{Asm}(\mathbb{V})$ have initial algebras. This yields initial algebras for functors $P_{X,Y} : \mathrm{UFam}(\mathrm{Asm}(\mathbb{V}))_1 \to \mathrm{UFam}(\mathrm{Asm}(\mathbb{V}))_1$ for $X \in \mathbb{E}_1$ and $Y \in \mathbb{E}_{\{X\}}$. This lifts to an arbitrary context $I \in \mathrm{Asm}(\mathbb{V})$ by a point-wise construction, which yields split W-types, as re-indexing in $\mathrm{UFam}(\mathrm{Asm}(\mathbb{V}))$ is by composition:

Lemma 4. *The split ccomp $\mathcal{P} : \mathrm{UFam}(\mathrm{Asm}(\mathbb{V})) \to \mathrm{Asm}(\mathbb{V})^{\to}$ has split W-types.*

This models W-types in the ***type***-universe. In addition, iHTT features small W-types over small types, with elimination over large types. The idea is to model these small W-types by forming the W-type in $\mathrm{UFam}(\mathrm{Asm}(\mathbb{V}))$ and mapping it back into $\mathrm{UFam}(\mathrm{CMPer}(\mathbb{V}))$ using the reflection. This yields a small type, and by the following lemma, this small type is still a W-type in $\mathrm{UFam}(\mathrm{Asm}(\mathbb{V}))$, and thus models elimination over large types:

Lemma 5. *For every $I \in \mathbb{B}$, $X \in \mathrm{UFam}(\mathrm{CMPer}(\mathbb{V}))_I$, and $Y \in \mathrm{UFam}(\mathrm{CMPer}(\mathbb{V}))_{\{X\}}$, there is a chosen isomorphism, $W_{\mathcal{I}_{el}(X),\mathcal{I}_{el}(Y)} \cong \mathcal{I}_{el}(\mathcal{R}_{el}(W_{\mathcal{I}_{el}(X),\mathcal{I}_{el}(Y)}))$, which is preserved on-the-nose by reindexing functors.*

Proof (Sketch). The reflection $\mathcal{R}_{el}$ collapses an assembly into a PER by equating values with overlapping sets of realizers and into a complete monotone PER by taking the monotone completion of said PER. The result follows by proving that $W_{X,Y}$ is a modest set (i.e., has no overlapping sets of realizers), and that the induced PER is already monotone complete, for X and Y in the image of $\mathcal{I}_{el}$.

4 Implementation

One of the advantages of weakening Hoare types instead of the underlying dependent type theory – as in the case of [21] – is that it can simplify implementation of the resulting type theory. In our case, presenting iHTT as an extension of CIC allows for easy implementation as an axiomatic extension of the Coq proof assistant.

Our implementation is based on the Coq infrastructure developed by Nanevski et. al. [19], to support efficient reasoning about stateful computations in Coq. This infrastructure defines a new Hoare type on top of the built-in Hoare type with support for more efficient reasoning, based on ideas from separation logic. Specifically, as illustrated in Section 2, this new Hoare type features (1) small footprint specifications and (2) efficient reasoning about heaps. Efficient reasoning about heaps is achieved by reasoning using the partial commutative monoid $(1+\boldsymbol{heap}, \bullet)$ instead of the total non-commutative monoid $(\boldsymbol{heap}, \bullet)$, where $\boldsymbol{heap}$ is the actual type of heaps and $\bullet$ is heap union [19].

Compared to Nanevski et. al.'s implementation of predicative Hoare Type Theory (pHTT), this new implementation features higher-order store and impredicative quantification, but the predicative hierarchy lacks Hoare-types. The new implementation is almost source compatible with verifications in pHTT that do not exploit the predicative hierarchy.

Compared to the Ynot implementation of pHTT [18], in addition to impredicativity the main difference lies in the treatment of ghost variables. Post conditions in Ynot are unary and thus employ ghost-variables to relate the pre- and post-condition. Ynot expresses ghost variables of a specification as computationally irrelevant arguments to the computation. As Coq lacks support for computationally irrelevant variables, Ynot extends Coq with an injectivity axiom, which gives an embedding of $\boldsymbol{set}$ into the computationally irrelevant $\boldsymbol{prop}$-universe. This axiom is inconsistent with a proof irrelevant $\boldsymbol{prop}$-universe and thus in particular unsound in our model. Additionally, this limits Ynot's ghost variables to small types, whereas iHTT supports ghost variables of large types.

5 Related work

Our approach to partiality is based on the idea of partial types, as introduced by Constable and Smith [9]. We have already discussed its relation to the work on admissibility by Crary [10] in the introduction. Below we first discuss related work on partiality, followed by related work on partial correctness reasoning.

Bove and Capretta [7] proposed representing a partial function $f : A \rightharpoonup B$ as a total function $\overline{f} : \Pi a : A.\ P(a) \to B$, defined by recursion over an inductively defined predicative $P : A \to \textbf{\textit{prop}}$, expressing the domain of the partial function. This allows the definition of partial computations by general recursion, but does not model non-termination, as $\overline{f}$ can only be applied to arguments on which it terminates. Capretta [8] proposed an alternative co-inductive representation, which does model non-termination, representing a partial function $f : A \rightharpoonup B$ as a total function $\overline{f} : A \to B^v$, where B^v is co-inductive type of partial elements of type B. This representation yields a least fixed point operator on finitary (continuous) endo-functions on $A \to B^v$. Capretta does not provide a fixed point induction principle, but we believe such a principle would require admissibility proofs.

Another alternative approach to partiality is to give a model of a language featuring general recursion inside a dependent type theory. This allows one to model and reason about partial computations inside the type theory, but does not extend the type theory itself with partial computations. This approach has for instance been studied and implemented by Reus [22], who formalized Synthetic Domain Theory in the Lego proof checker. The resulting type theory can be seen as a very expressive version of LCF. The synthetic approach alleviates the need for continuity proofs, but still requires admissibility proofs when reasoning by fixed point induction.

Hoare-style specification logics is another line of closely related work. With a collapsed computational equality, reasoning about a partial computation in iHTT is limited to Hoare-style partial correctness reasoning, as in a specification logic. With the modularity features provided by the underlying dependent type theory (i.e., Σ types), iHTT can thus be seen as a modular specification logic for a higher-order programming language.

6 Conclusion

We have presented a new approach for extending dependent type theory with potentially non-terminating computations, without weakening the underlying dependent type theory or adding restrictions on the use of fixed points in defining partial computations. We have shown that it scales to very expressive dependent type theories and effects beyond partiality, by extending the Calculus of Inductive Constructions with stateful and potentially non-terminating computations. We have further demonstrated that this approach is practical, by implementing this extension of CIC as an axiomatic extension of the Coq proof assistant.

To justify the soundness of our extension of CIC, we have presented a realizability model of the theory. For lack of space, we have limited the presentation to a single predicative universe, but the model can be extended to the whole predicative hierarchy $\textbf{\textit{type}} \subseteq \textbf{\textit{type}}_1 \subseteq \ldots$ of CIC.

[†] This research has been partially supported by MICINN Project TIN2010-20639 Paran10; AMAROUT grant PCOFUND-GA-2008-229599; and Ramon y Cajal grant RYC-2010-0743.

References

1. The Coq Proof Assistant. http://coq.inria.fr/.
2. *Coq Reference Manual, Version 8.3*.
3. M. Abbott, T. Altenkirch, and N. Ghani. Representing nested inductive types using W-types. In *Proc. of ICALP*, 2004.
4. R. M. Amadio. Recursion over Realizability Structures. *Information and Computation*, 91:55–85, 1991.
5. L. Birkedal, K. Støvring, and J. Thamsborg. Realisability semantics of parametric polymorphism, general references and recursive types. *Math. Struct. Comp. Sci.*, 20(4):655–703, 2010.
6. D. Bjørner and C. B. Jones, editors. *The Vienna Development Method: The Meta-Language*, volume 61 of *Lecture Notes in Computer Science*. Springer, 1978.
7. A. Bove. Simple General Recursion in Type Theory. *Nordic Journal of Computing*, 8, 2000.
8. V. Capretta. General Recursion via Coinductive Types. *Logical Methods in Computer Science*, 1(2):1–28, 2005.
9. R. L. Constable and S. F. Smith. Partial Objects in Constructive Type Theory. In *Proceedings of Second IEEE Symposium on Logic in Computer Science*, 1987.
10. K. Crary. Admissibility of Fixpoint Induction over Partial Types. In *Automated Deduction - CADE-15*, 1998.
11. P. Dybjer. Representing inductively defined sets by wellorderings in Martin-Löf's type theory. *Theor. Comput. Sci.*, 176(1-2):329–335, 1997.
12. G. Gonthier and A. Mahboubi. A Small Scale Reflection Extension for the Coq system. Technical report, INRIA, 2007.
13. C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1:271–281, 1972.
14. B. Jacobs. *Categorical Logic and Type Theory*. Elsevier Science, 1999.
15. B. Meyer. *Object-oriented software construction*. Prentice Hall, 1997.
16. I. Moerdijk and E. Palmgren. Wellfounded trees in categories. *Annals of Pure and Applied Logic*, 104(1–3):189–218, 2000.
17. A. Nanevski, G. Morrisett, and L. Birkedal. Hoare Type Theory, Polymorphism and Separation. *Journal of Functional Programming*, 18(5–6):865–911, 2008.
18. A. Nanevski, G. Morrisett, A. Shinnar, P. Govereau, and L. Birkedal. Ynot: Dependent Types for Imperative Programs. In *Proceedings of ICFP 2008*, pages 229–240, 2008.
19. A. Nanevski, V. Vafeiadis, and J. Berdine. Structuring the Verification of Heap-Manipulating Programs. In *Proceedings of POPL 2010*, 2010.
20. P. O'Hearn, J. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *CSL'01*, pages 1–19, 2001.
21. R. Petersen, L. Birkedal, A. Nanevski, and G. Morrisett. A Realizability Model of Impredicative Hoare Type Theory. In *Proceedings of ESOP 2008*, 2008.
22. B. Reus. Synthetic Domain Theory in Type Theory: Another Logic of Computable Functions. In *Proceedings of TPHOL 1996*, 1996.
23. K. Svendsen, L. Birkedal, and A. Nanevski. Partiality, State and Dependent Types. Technical report, IT University of Copenhagen, 2011. Available at: `http://www.itu.dk/people/kasv/ihtt-adm-tr.pdf`.