Abstract Predicates and Mutable ADTs in Hoare Type Theory

Aleksandar Nanevski[1], Amal Ahmed[2], Greg Morrisett[1], and Lars Birkedal[3]

[1] Harvard University {aleks,greg}@eecs.harvard.edu
[2] Toyota Technological Institute at Chicago amal@tti-c.org
[3] IT University of Copenhagen birkedal@itu.dk

Abstract. *Hoare Type Theory* (HTT) combines a dependently typed, higher-order language with monadically-encapsulated, stateful computations. The type system incorporates pre- and post-conditions, in a fashion similar to Hoare and Separation Logic, so that programmers can modularly specify the requirements and effects of computations within types.
This paper extends HTT with quantification over abstract predicates (i.e., higher-order logic), thus embedding into HTT the Extended Calculus of Constructions. When combined with the Hoare-like specifications, abstract predicates provide a powerful way to define and encapsulate the invariants of private state that may be shared by several functions, but is not accessible to their clients. We demonstrate this power by sketching a number of abstract data types that demand ownership of mutable memory, including an idealized custom memory manager.

1 Background

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

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

The problem can be addressed by severely restricting dependencies to only effect-free terms (as in for instance DML [30]). But the goal of our work is to try to realize the full power of dependent types for specification of effectful programs. To that end, we have been developing the foundations of a language that we call *Hoare Type Theory* or HTT [22], which we intend to be an expressive and explicitly annotated internal language, providing a semantic framework for elaborating more practical external languages.

HTT starts with a pure, dependently typed core language and augments it with an indexed monadic type of the form $\{P\}x{:}A\{Q\}$. This type encapsulates and describes effectful computations that may diverge or access a mutable store. The type can be read as a Hoare-like partial correctness specification, asserting that if the computation is run in a world satisfying the pre-condition P, then if it terminates, it will return a value x of type A and be in a world 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 [26, 24], where the pre- and postconditions only describe the part of the store that the program actually uses; the unspecified part is automatically assumed invariant.

Recently, several variants of Hoare Logic for higher-order, effectful languages have appeared. Yoshida, Honda and Berger [31, 4] define a logic for PCF with references, Krishnaswami [13] defines a Separation Logic for core ML extended with a monad, and Birkedal et al. [5] define a Higher-Order Separation Logic for reasoning about ADTs in first-order programs. However, we believe that HTT has several key advantages over these and other proposed logics. First, HTT supports *strong* (i.e., type-varying) updates of mutable locations, while the above program logics require that the types of memory locations are *invariant.* This restriction makes it difficult to model stateful protocols as in the Vault language [7], or low-level languages such as TAL [20] and Cyclone [12] where memory management is intended to be coded within the language. Second, none of these logics considers pointer arithmetic, nor source language features like type abstraction, modules, or dependent types, which we consider here. Third, and most significant, Hoare logics cannot really interact with the type systems of the underlying language, unlike HTT where specifications are *integrated with types.* In Hoare Logic, it is not possible to abstract over specifications in the source programs, aggregate the logical invariants of the data structures with the data itself, compute with such invariants, or nest the specifications into larger specifications or types. These features are essential ingredients for data abstraction and information hiding, and, in fact, a number of works have been proposed towards integrating Hoare-like reasoning with type checking. Examples include tools and languages like Spec# [1], SPLint [9], ESC/Java [8], and JML [6].

There are several important outstanding problems in the design of such languages for integrated programming and verification. As discussed in [6], for example: (1) It is desirable to use effectful code in the specifications, but most languages insist that specifications must be pure, in order to preserve soundness. Such a restriction frequently leads to implementing the same functionality twice – once purely for specification, and once impurely for execution. (2) Specifications should be able to describe and control pointer aliasing. (3) It is tricky to define a useful notion of object or module invariant, primarily because of local state owned by the object. Most definitions are too restrictive to support some important programming patterns, for example, reentrant callbacks [2].

Our prior work on HTT [22] addresses the first two problems: (1) we allow effectful code in specifications by granting such code first-class status, via the monad for Hoare triples, and (2) we control pointer aliasing, by employing the small footprint approach of Separation Logic. Both of these properties were discussed at the beginning of this section. The focus of this paper are extensions to HTT that enable us to also address problem (3), among others.

In a language like HTT that integrates programming and verification, truly reusable program components (e.g., libraries of data types and first-class objects) require that their *internal invariants* are appropriately abstracted. The component interfaces need to include not only abstract types, but also abstract specifications. Thus it is natural to extend HTT with support for abstraction over predicates (i.e., higher-order logic). More specifically, we describe a variant of HTT that includes the Extended Calculus of Constructions (ECC) [14], modulo minor differences described in Section 5. This allows terms, types, and predicates to all be abstracted within terms, types, and predicates respectively.

There are several benefits of this extension. First, higher-order logic can formulate almost any predicate that may be encountered during program verification, including predicates defined by induction and coinduction. Second, we can reason *within* the system, about the equality of terms, types and predicates, *including abstract types and abstract predicates.* In the previous version of HTT [22], we could only reason about the equality of terms, whereas equality on types and predicates was a judgment (accessible to the typechecker), but *not* a proposition (accessible to the programmer). Internalized reasoning on types endows HTT with a form of *first-class* modules that can contain types, terms, *and axioms.* It is also important in order to fully support strong updates of locations. Third, higher-order logic can *define* many constructs that, in the previous version, had to be primitive. For instance, the definition of heaps can now be encoded within the language, thus simplifying some aspects of the meta theory.

Most importantly, however, *abstraction over predicates suffices to represent the private state of functions or ADTs within the type system.* Private state can be hidden from the clients by existentially abstracting over the state invariant. Thus, libraries for mutable state can provide precise specifications, yet have sufficient abstraction mechanisms that different implementations can share a common interface. Moreover, specifications may choose to reveal certain aspects of private state to the client, thus granting the client partial or complete access to, or even ownership of portions of the private state.

We demonstrate these ideas with a few idealized examples including a module for memory allocation and deallocation.

2 Overview

Similar to the modern monadic functional languages [19], HTT syntax splits into the pure and the impure fragment. The pure fragment contains higher-order functions and pairs, and the impure fragment contains the effectful commands for memory lookup and strong update (memory allocation and deallocation can be defined), as well as conditionals and recursion. The expressions from the effectful fragment can be coerced into the pure one by monadic encapsulation.

The type constructors include the primitive types of booleans, natural numbers and the unit type, the standard constructors Π and Σ for dependent products and sums, as well as Hoare types $\{P\}x{:}A\{Q\}$, and subset types $\{x{:}A.\, P\}$. The Hoare type $\{P\}x{:}A\{Q\}$ is the *monadic type* which classifies effectful computations that may execute in any initial heap satisfying the assertion P, and either diverge, or terminate returning a value $x{:}A$ and a final heap satisfying the assertion Q. The subset type $\{x{:}A.\, P\}$ classifies all the elements of A that satisfy the predicate P. We adopt the standard convention and write $A{\rightarrow}B$ and $A{\times}B$ instead of $\Pi x{:}A.\, B$ and $\Sigma x{:}A.\, B$ when B does not depend on x.

The syntax of our extended HTT is presented in the following table.

Types	A, B, C	$::=$	$K \mid \mathsf{nat} \mid \mathsf{bool} \mid 1 \mid \mathsf{prop} \mid \mathsf{mono} \mid \Pi x{:}A.\, B \mid$
			$\Sigma x{:}A.\, B \mid \{P\}x{:}A\{Q\} \mid \{x{:}A.\, P\}$
Elim terms	K, L	$::=$	$x \mid K\ N \mid \mathsf{fst}\ K \mid \mathsf{snd}\ K \mid \mathsf{out}\ K \mid M : A$
Intro terms	M, N, O	$::=$	$K \mid (\,) \mid \lambda x.\, M \mid (M, N) \mid \mathsf{do}\, E \mid \mathsf{in}\ M \mid$
			$\mathsf{true} \mid \mathsf{false} \mid \mathsf{z} \mid \mathsf{s}\ M \mid M + N \mid M \times N \mid \mathsf{eq}_{\mathsf{nat}}(M, N) \mid$
(Assertions)	P, Q, R		$\top \mid \bot \mid \mathsf{xid}_{A,B}(M, N) \mid \neg P \mid P \wedge Q \mid$
			$P \vee Q \mid P \supset Q \mid \forall x{:}A.\, P \mid \exists x{:}A.\, P \mid$
(Small types)	τ, σ		$\mathsf{nat} \mid \mathsf{bool} \mid 1 \mid \mathsf{prop} \mid \Pi x{:}\tau.\, \sigma \mid \Sigma x{:}\tau.\, \sigma \mid \{P\}x{:}\tau\{Q\} \mid \{x{:}\tau.\, P\}$
Commands	c	$::=$	$!_\tau\, M \mid M :=_\tau N \mid \mathsf{if}_A\ M\ \mathsf{then}\ E_1\ \mathsf{else}\ E_2 \mid$
			$\mathsf{case}_A\ M\ \mathsf{of}\ \mathsf{z} \Rightarrow E_1\ \mathsf{or}\ \mathsf{s}\ x \Rightarrow E_2 \mid$
			$\mathsf{fix}\ f(y{:}A){:}B = \mathsf{do}\, E\ \mathsf{in}\ \mathsf{eval}\ f\, M$
Computations	E, F	$::=$	$\mathsf{return}\, M \mid x \leftarrow K; E \mid x \Leftarrow c; E \mid x =_A M; E$
Context	Δ	$::=$	$\cdot \mid \Delta, x{:}A \mid \Delta, P$

HTT supports predicative type polymorphism [18], by differentiating *small types*, which do not admit type quantification, from *large types* (or just types for short), which can quantify over small types only. For example, the polymorphic identity function can be written as $\lambda\alpha.\lambda y.y : \Pi\alpha{:}\mathsf{mono}.\Pi y{:}\alpha.\alpha$, but α ranges over only small types. The restriction to predicative polymorphism is crucial for ensuring that during type-checking, normalization of terms, types, and predicates terminates [22]. Note that "small" Hoare triples $\{P\}x{:}\tau\{Q\}$ and subset types $\{x{:}\tau.\, P\}$, where P and Q (but not τ) may contain type quantification are considered small. This is because P and Q are *refinements*, i.e. they do not influence the underlying semantics and the equational reasoning about terms: If two terms of some Hoare or subset types are semantically equal, then they remain equal even if P and Q are replaced by some other assertions.

To support abstraction over types and predicates, HTT introduces types mono and prop which classify small types and assertions respectively. With the type mono, HTT can *compute* with small types as if they were data. For example, if $x{:}\mathsf{mono}{\times}(\mathsf{nat}{\rightarrow}\mathsf{nat})$, then the variable x may be seen as a module declaring a small type and a function on nats. The expression $\mathsf{fst}\ x$ extracts the small type.

Terms. The terms are classified as introduction or elimination terms, according to their standard logical properties. The split facilitates equational reasoning and bidirectional typechecking [25]. The terms are not annotated with types, as the typechecker can infer most of them. When this is not the case, the construct $M : A$ may supply the type explicitly. This construct also switches the direction in the bidirectional typechecking.

HTT features the usual terms for lambda abstraction and applications, pairs and the projections, as well as natural numbers, booleans and the unit element. The introduction form for the Hoare types is $\mathsf{do}\, E$, which encapsulates the effectful computation E, and suspends its evaluation. The notation is intended to closely resemble the familiar Haskell-style do-notation for writing effectful computations. The constructor in is a coercion from A into a subset type $\{x{:}A.\, P\}$, and out is the opposite coercion.

Terms also include small types τ and assertions P, which are the elements of mono and prop respectively. HTT does not currently have any constructors to inspect the structure of such elements. They are used solely during typechecking, and can be safely erased before program execution.

We illustrate the HTT syntax using the following example. Consider an ML-like function $f = \lambda y{:}\mathsf{unit}.\, x := !x + 1;\ \mathsf{if}\ (!x = 1)\ \mathsf{then}\ 0\ \mathsf{else}\ 1$, where we assume a free variable $x{:}\mathsf{nat\ ref}$. A computation in HTT that defines this function and then immediately applies it, may be written as follows.

$$\begin{array}{l} f = \lambda y.\, \mathsf{do}\, (u \Leftarrow !_{\mathsf{nat}}\, x; v \Leftarrow (x :=_{\mathsf{nat}}\, u + \mathsf{s\ z}); t \Leftarrow !_{\mathsf{nat}}\, x; \\ \qquad\qquad s \Leftarrow \mathsf{if}_{\mathsf{nat}}\ (\mathsf{eq}_{\mathsf{nat}}(t, \mathsf{s\ z}))\ \mathsf{then\ z\ else\ s\ z}; \mathsf{return}\, s); \\ x \leftarrow f\ (\,);\ \mathsf{return}\,(x) \end{array}$$

We point out some characteristic properties. This program, and all its stateful subcomponents belong to the syntactic domain of computations. Each computation can intuitively be described as a semi-colon-separated list of commands, which usually perform some imperative operation, and then bind to a variable. For example $x \Leftarrow c$ executes the primitive command c, and binds the return result to x. $x \leftarrow K$ executes the computation encapsulated in K, thus performing all the side effects that may have been suspended in K. $x =_A M$ does not perform any side-effects, but is simply the syntactic sugar for the usual let-binding of $M{:}A$ to x. In all these cases, the variable x is immutable, as is customary in functional programming, and its scope extends to the right, until the end of the block enclosed by the nearest do. Associated with these commands, is the construct $\mathsf{return}\, M$. It creates the trivial computation that immediately returns the value M. $\mathsf{return}\, M$ and $x \leftarrow K; E$ correspond to the standard monadic *unit* and *bind*, respectively.

The commands $!_\tau\, M$ and $M :=_\tau N$ are used to read and write memory respectively. The index τ is the type of the value being read or written. Note that unlike ML and most statically-typed languages, HTT supports *strong updates*. That is, if x is a location holding a nat, then we can update the contents of x with a value of an arbitrary (small) type, not just another nat. (Here, we make the simplifying assumption that locations can hold a value of any type (e.g., values are boxed).) Type-safety is ensured by the pre-condition for memory reads which captures the requirement that to read a τ value out of location M, we must be able to prove that M currently holds such a value.

In the if and case commands, the index type A is the type of the branches. The fixpoint command $\mathsf{fix}\ f(y{:}A){:}B = \mathsf{do}\, E\ \mathsf{in\ eval}\ f\, M$, first obtains the function $f{:}\Pi y{:}A.\, B$ such that $f(y) = \mathsf{do}(E)$, then evaluates the computation $f(M)$, and returns the result.

In the subsequent text we adopt a number of syntactic conventions for terms. First, we will represent natural numbers in their usual decimal form. Second, we omit the variable x in $x \Leftarrow (M :=_\tau N); E$, as x is of unit type. Third, we abbreviate the computation of the form $x \Leftarrow c; \mathsf{return}\, x$ simply as c, in order to avoid introducing a spurious variable x. For the same reason, we abbreviate $x \leftarrow K; \mathsf{return}\, x$ as $\mathsf{eval}\ K$.

Returning to the example above, the type of f in the translated HTT program is $1{\rightarrow}\{P\}s{:}\mathsf{nat}\{Q\}$ where, intuitively, the precondition P requires that the location x points to some value $v{:}\mathsf{nat}$, and the postcondition Q states that if v was zero, then the result s is 0, otherwise the result is 1, and regardless x now points to $v+1$. Furthermore, in HTT, the specifications capture the *small footprint* of f, reflecting that x is the only location accessed when the computation is run. Technically, realizing such a specification using the predicates we provide requires a number of auxiliary definitions and conventions which are explained below. For instance, we must define the relation $x \mapsto v$ stating that x points to v, the equalities, and how v can be scoped across both the pre- and post-condition.

Assertions. The assertion logic is classical and includes the standard propositional connectives and quantifiers over all types of HTT. Since prop is a type, we can quantify over propositions, and more generally over propositional functions, giving us the power of higher-order logic. The primitive proposition $\mathsf{xid}_{A,B}(M,N)$ implements *heterogeneous equality* (aka. *John Major equality* [17]), and is true only if the types A and B, as well as the terms $M{:}A$ and $N{:}B$ are propositionally equal. We will use this proposition to express that if two heap locations x_1 (pointing to value $M_1{:}\tau_1$) and x_2 (pointing to value $M_2{:}\tau_2$) are equal, then $\tau_1 = \tau_2$ and $M_1 = M_2$. When the index types are equal in the heterogeneous equality $\mathsf{xid}_{A,A}(M,N)$, we abbreviate that as $\mathsf{id}_A(M,N)$, and often also write $M =_A N$ or just $M = N$. We denote by $\mathsf{lfp}_A(Q)$ the least fixed point of the monotone predicate $Q{:}(A{\rightarrow}\mathsf{prop}){\rightarrow}A{\rightarrow}\mathsf{prop}$ (Q is monotone if it uses the argument only in positive positions). It is well-known that this construct is definable in higher-order logic [11]. Heaps in which HTT computations are evaluated can be defined as a simple subset type $\mathsf{heap} = \{h{:}(\mathsf{nat}{\times}\Sigma\alpha{:}\mathsf{mono}.\alpha){\rightarrow}\mathsf{prop}.\, \mathsf{Finite}(h) \wedge \mathsf{Functional}(h)\}$. The underlying type $\mathsf{nat}{\times}\Sigma\alpha{:}\mathsf{mono}.\,\alpha$ implies that a heap is a ternary relation which takes $M{:}\mathsf{nat}$, $\alpha{:}\mathsf{mono}$ and $N{:}\alpha$ and decides if the location M points to $N{:}\alpha$. The predicates Finite and $\mathsf{Functional}$ are easily definable to state that a heap assigns to at most finitely many locations, and at most one value to every location. In HTT, heap locations are natural numbers, rather than elements of an abstract type. This simplifies the semantics somewhat, and also enables pointer arithmetic. Note that heaps in HTT can store only values of small types. This is sufficient for modeling languages with predicative polymorphism like SML, but is too weak for modeling Java, or the impredicative polymorphism of Haskell.

We also adopt the usual predicates from Separation Logic [26, 24]: emp, $(n \mapsto_\tau x)$ and $(n \hookrightarrow_\tau x)$ all have type $\mathsf{heap}{\rightarrow}\mathsf{prop}$. $\mathsf{emp}\ h$ holds iff h is the empty relation; $(n \mapsto_\tau x)(h)$ holds if h contains *only one* location n pointing to a value $x{:}\tau$. Similarly, $(n \hookrightarrow_\tau x)(h)$ states that h contains *at least* the location n pointing to $x{:}\tau$. Finally, given $P, Q{:}\mathsf{heap}{\rightarrow}\mathsf{prop}$, the spatial conjunction $P * Q{:}\mathsf{heap}{\rightarrow}\mathsf{prop}$ is defined so that $(P * Q)(h)$ holds iff P and Q hold on disjoint subheaps of h. All of these predicates are easily definable using higher-order assertion logic.

3 Examples

Small footprints. HTT supports small-footprint specifications, as in Separation Logic [22]. If $\mathsf{do}\,E$ has type $\{P\}x{:}A\{Q\}$ — note that here $P : \mathsf{heap}{\to}\mathsf{prop}$ and $Q : \mathsf{heap}{\to}\mathsf{heap}{\to}\mathsf{prop}$ — then P and Q need only describe the properties of the heap fragment that E actually requires in order to run. The actual heap in which E will run may be much larger, but the unspecified portion will automatically be assumed invariant. To illustrate this idea, let us consider a simple program that reads from the location x and increases its contents.

$$\begin{aligned}\mathsf{incx}\ :\ & \{\lambda i.\,\exists n{:}\mathsf{nat}.\,(x \mapsto_{\mathsf{nat}} n)(i)\}\ r{:}1\\ & \{\lambda i.\,\lambda m.\,\forall n{:}\mathsf{nat}.\,(x \mapsto_{\mathsf{nat}} n)(i) \supset (x \mapsto_{\mathsf{nat}} n{+}1)(m)\}\\ =\ & \mathsf{do}(u \Leftarrow !_{\mathsf{nat}}\, x; x :=_{\mathsf{nat}} u + 1; \mathsf{return}\ (\,))\end{aligned}$$

Notice that the precondition states that the initial heap i contains *exactly one* location x, while the postcondition relates i with the heap m obtained after the evaluation (and states that m contains exactly one location too). This does not mean that incx can evaluate only in singleton heaps. Rather, incx requires a heap from which it can carve out a fragment that satisfies the precondition, i.e. a fragment containing a location x pointing to a nat. For example, we may execute incx against a larger heap, which contains the location y as well, and the contents of y is guaranteed to remain unchanged.

$$\begin{aligned}\mathsf{incxy}\ :\ & \{\lambda i.\,\exists n.\,\exists k{:}\mathsf{nat}.\,(x \mapsto_{\mathsf{nat}} n * y \mapsto_{\mathsf{nat}} k)(i)\}\ r{:}1\\ & \{\lambda i.\,\lambda m.\,\forall n.\,\forall k{:}\mathsf{nat}.\,(x \mapsto_{\mathsf{nat}} n * y \mapsto_{\mathsf{nat}} k)(i) \supset (x \mapsto_{\mathsf{nat}} n{+}1 * y \mapsto_{\mathsf{nat}} k)(m)\}\\ =\ & \mathsf{do}(\mathsf{eval}\ \mathsf{incx})\end{aligned}$$

To avoid clutter in specifications, we introduce a convention: if $P, Q{:}\mathsf{heap}{\to}\mathsf{prop}$ are predicates that may depend on the free variable $x{:}A$, we write $x{:}A.\,\{P\}y{:}B\{Q\}$ instead of $\{\lambda i.\,\exists x{:}A.\,P(i)\}y{:}B\{\lambda i.\,\lambda m.\,\forall x{:}A.\,P(i) \supset Q(m)\}$. This notation lets x seem to scope over both the pre- and post-condition. For example the type of incx can now be written $n{:}\mathsf{nat}.\,\{x \mapsto_{\mathsf{nat}} n\}r{:}1\{x \mapsto_{\mathsf{nat}} n{+}1\}$. The convention is easily generalized to a finite context of variables, so that we can also abbreviate the type of incxy as $n{:}\mathsf{nat}, k{:}\mathsf{nat}.\,\{x \mapsto_{\mathsf{nat}} n * y \mapsto_{\mathsf{nat}} k\}r{:}1\{x \mapsto_{\mathsf{nat}} n{+}1 * y \mapsto_{\mathsf{nat}} k\}$. Following the terminology of Hoare Logic, we call the variables abstracted outside of the Hoare triple, like n and k above, *logic variables* or *ghost variables.*

Nontermination. The following is a computation of an arbitrary monadic type that diverges upon forcing.

$$\begin{aligned}\mathsf{diverge}\ :\ & \{P\}x{:}A\{Q\}\\ =\ & \mathsf{do}\,(\mathsf{fix}\ f(y : 1) : \{P\}x{:}A\{Q\} = \mathsf{do}\,(\mathsf{eval}\ (f\ y))\\ & \quad \mathsf{in}\ \mathsf{eval}\ f\ (\,))\end{aligned}$$

diverge sets up a recursive function $f(y : 1) = \mathsf{do}\,(\mathsf{eval}\,(f\,y))$; then applies it to $(\,)$ to obtain another suspended computation $\mathsf{do}\,(\mathsf{eval}\,f\,(\,))$, which is immediately forced by eval to trigger another application to $(\,)$, and so on.

Allocation and Deallocation. The reader may be surprised that we provide no primitives for allocating (or deallocating) locations within the heap. This is because we can encode such primitives *within* the language in a style similar to Benton's recent semantic framework for specification of machine code [3]. We can encode a number of memory management implementations and give them a uniform interface, so that clients can choose from among different allocators.

We assume that upon start up, the memory module already "owns" all of the free memory of the program. It exports two functions, alloc and dealloc, which can transfer the ownership of locations between the allocator module and its clients. The functions share the memory owned by the module, but this memory will not be accessible to the clients (except via direct calls to alloc and dealloc).

The definitions of the allocator module will use two essential features of HTT. First, there is a mechanism in HTT to abstract the local state of the module and thus protect it from access from other parts of the program. Second, HTT supports strong updates, and thus it is possible for the memory module to recycle locations to hold values of different type at different times throughout the course of the program execution.

The interface for the allocator can be captured with the type:

$$\begin{aligned}\mathsf{Alloc} = [\,& I : \mathsf{heap}{\rightarrow}\mathsf{prop},\\ & \mathsf{alloc} : \Pi\alpha{:}\mathsf{mono}.\, \Pi x{:}\alpha.\, \{I\}r{:}\mathsf{nat}\{\lambda i.\, (I * r \mapsto_\alpha x)\},\\ & \mathsf{dealloc} : \Pi n{:}\mathsf{nat}.\, \{I * n \mapsto -\}r{:}1\{\lambda i.\, I\}\ ]\end{aligned}$$

where the notation $[x_1{:}A_1, \ldots, x_n{:}A_n]$ abbreviates a sum $\Sigma x_1{:}A_1 \cdots \Sigma x_n{:}A_n.1$. In English, the interface says that there is some abstract invariant I, reflecting the internal invariant of the module, paired with two functions. Both functions require that the invariant I holds before and after calls to the functions. In addition, a call $\mathsf{alloc}\,\tau\,x$ will yield a location r and a guarantee that r points to x. Furthermore, we know from the use of the spatial conjunction that r is disjoint from the internal invariant I. Thus, updates by the client to r will not break the invariant I. On the other hand, accessing locations hidden by I becomes impossible. As will be apparent from the typing rules in Section 4, each location access requires proving that the location exists. But, when I is abstracted, the knowledge needed to construct this proof, is hidden as well. Dually, dealloc requires that we are given a location n, pointing to some value and disjoint from the memory covered by the invariant I. Upon return, the invariant is restored and the location consumed.

If M is a module with this signature, then a program fragment that wishes to use this module will have to start with a pre-condition fst M. That is, clients will generally have the type $\Pi M{:}\mathsf{Alloc}.\{(\mathsf{fst}\ M) * P\}r{:}A\{\lambda i.\,(\mathsf{fst}\ M) * Q(i)\}$ where Alloc is the signature given above.

Allocator Module 1. Our first implementation of the allocator module assumes that there is a location r such that all the locations $n \geq r$ are free. The value of r is recorded in the location 0. All the free locations are initialized with the unit value ( ). Upon a call to alloc, the module returns the location r and sets $0 \mapsto_{\mathsf{nat}} r{+}1$, thus removing r from the set of free locations. Upon a call $\mathsf{dealloc}\,n$, the value of r is decreased by one if $r = n$ and otherwise, nothing happens. Obviously, this kind of implementation is very naive. For instance, it assumes unbounded memory and will leak memory if a deallocated cell was not the most recently allocated. However, the example is still interesting to illustrate the features of HTT. First, we define a predicate that describes the free memory as a list of consecutive locations initialized with ( ) : 1.

$$\begin{aligned}\mathsf{free} &: (\mathsf{nat} \times \mathsf{heap}) \rightarrow \mathsf{prop}\\ &= \mathsf{lfp}\ (\lambda F.\, \lambda(r,h).\, (r \mapsto_1 (\,) * \lambda h'.\, F\ (r{+}1, h'))(h))\end{aligned}$$

Then we can implement the allocator module as follows:

$$\begin{aligned}[\,I &= \lambda h.\, \exists r{:}\mathsf{nat}.\, (0 \mapsto_{\mathsf{nat}} r * \lambda h'.\, \mathsf{free}(r,h') * \lambda h''.\, \top)(h),\\ \mathsf{alloc} &= \lambda\alpha.\, \lambda x.\, \mathsf{do}\, (u \Leftarrow !_{\mathsf{nat}}\, 0; u :=_\alpha x; 0 :=_{\mathsf{nat}} u{+}1; \mathsf{return}\, u),\\ \mathsf{dealloc} &= \lambda n.\, \mathsf{do}\, (u \Leftarrow !_{\mathsf{nat}}\, 0;\\ &\qquad \mathsf{if}\ \mathsf{eq}_{\mathsf{nat}}(u, n{+}1)\ \mathsf{then}\ n :=_1 (\,); 0 :=_{\mathsf{nat}} n; \mathsf{return}\,(\,)\ \mathsf{else}\ \mathsf{return}\,(\,))\ ]\end{aligned}$$

Allocator Module 2. In this example we present a (slightly) more sophisticated allocator module. The module will have the same Alloc signature as in the previous example, but the implementation does not leak memory upon deallocation. We take some liberties and assume as primitive a standard set of definitions and operations for the inductive type of lists.

$$\begin{aligned}\mathsf{list} &: \mathsf{mono}{\rightarrow}\mathsf{mono}\\ \mathsf{nil} &: \Pi\alpha{:}\mathsf{mono}.\, \mathsf{list}\ \alpha\\ \mathsf{cons} &: \Pi\alpha{:}\mathsf{mono}.\, \alpha{\rightarrow}\mathsf{list}\ \alpha{\rightarrow}\mathsf{list}\ \alpha\\ \mathsf{snoc} &: \Pi\alpha{:}\mathsf{mono}.\, \Pi x{:}\{y{:}\mathsf{list}\ \alpha.\, y \neq_{\mathsf{list}\ \alpha} \mathsf{nil}\ \alpha\}.\, \{z{:}\alpha \times \mathsf{list}\ \alpha.\, x = \mathsf{in}\ (\mathsf{cons}(\mathsf{fst}\ z)(\mathsf{snd}\ z))\}\\ \mathsf{nil?} &: \Pi\alpha{:}\mathsf{mono}.\, \Pi x{:}\mathsf{list}\ \alpha.\, \{y{:}\mathsf{bool}.(y =_{\mathsf{bool}} \mathsf{true}) \subset\supset (x =_{\mathsf{list}\ \alpha} \mathsf{nil}\ \alpha)\}\end{aligned}$$

The operation snoc maps non-empty lists back to pairs so that the head and tail can be extracted (without losing equality information regarding the components.) The operation nil? tests a list, and returns a bool which is true iff the list is nil.

As before, we define the predicate free that describes the free memory, but this time, we collect the (finitely many) addresses of the free locations into a list.

$$\begin{aligned}\mathsf{free} &: ((\mathsf{list\ nat}){\times}\mathsf{heap}){\rightarrow}\mathsf{prop}\\ &= \mathsf{lfp}\ (\lambda F.\, \lambda(l,h).\, (l = \mathsf{nil\ nat}) \vee \exists x'{:}\mathsf{nat}.\, \exists l'{:}\mathsf{list\ nat}.\\ &\qquad l = \mathsf{cons\ nat}\ x'\ l' \wedge (x' \mapsto_1 (\,) * \lambda h'.\, F(l',h'))(h))\end{aligned}$$

The intended invariant now is that the list of free locations is stored at address 0, so that the module is implemented as follows:

$$\begin{aligned}[\,I &= \lambda h.\, \exists l{:}\mathsf{list\ nat}.\, (0 \mapsto_{\mathsf{list\ nat}} l * \lambda h'.\, \mathsf{free}(l,h'))(h),\\ \mathsf{alloc} &= \lambda\alpha.\, \lambda x.\, \mathsf{do}\, (l \Leftarrow !_{\mathsf{list\ nat}}\, 0;\ \mathsf{if}\ (\mathsf{out}\ (\mathsf{nil?\ nat}\ l))\ \mathsf{then}\ \mathsf{eval}\ (\mathsf{diverge})\\ &\qquad \mathsf{else}\ p \Leftarrow \mathsf{out}\ (\mathsf{snoc\ nat}\ (\mathsf{in}\ l));\ 0 :=_{\mathsf{list\ nat}} \mathsf{snd}\ p;\\ &\qquad \mathsf{fst}\ p :=_\alpha x;\ \mathsf{return}\,(\mathsf{fst}\ p)),\\ \mathsf{dealloc} &= \lambda x.\, \mathsf{do}\, (l \Leftarrow !_{\mathsf{list\ nat}}\, 0; x :=_1 (\,); 0 :=_{\mathsf{list\ nat}} \mathsf{cons\ nat}\ x\ l; \mathsf{return}\,(\,))\ ]\end{aligned}$$

This version of alloc reads the free list out of location 0. If it is empty, then the function diverges. Otherwise, it extracts the first free location z, writes the rest of the free list back into 0, and returns z. The dealloc simply adds its argument back to the free list.

Functions with local state. Now, we consider examples that illustrate various modes of use of the invariants on local state. We assume the allocator from the previous example, and admit the free variables I and alloc, with types as in Alloc. These can be instantiated with either of the two implementations above.

Let us consider an HTT computation that allocates a location x with integer content, and then returns a computation for incrementing x. The first attempt at writing this computation may be as:

$$E = \mathsf{do}\,(x \leftarrow \mathsf{alloc\ nat}\ 0;\ \mathsf{do}\,(z \Leftarrow !_{\mathsf{nat}}\, x; x :=_{\mathsf{nat}} z{+}1; \mathsf{return}\,(z{+}1))).$$

E can be given several different types that describe its behavior with various levels of precision. But, here, we are interested in a type for E that describes it "fully". In other words, we would like to specify that the return value of E is a computation whose successive executions return an increasing sequence of natural numbers. The computation remembers the last computed natural number in its local store (here, the location x) which persists between successive calls. But the details of this store should be hidden from the clients of E, precisely to preserve its locality.

In HTT we can use the ability to combine terms, propositions and Hoare triples, and abstract x away, while exposing only the invariant that the computation increases the content of x.

$$\begin{array}{lcl} E_1 & = & \mathsf{do}\,(x \leftarrow \mathsf{alloc\ nat}\ 0\ \mathsf{in}\ (\lambda v.\, x \mapsto_{\mathsf{nat}} v, \mathsf{do}\,(z \Leftarrow !_{\mathsf{nat}}\, x; x :=_{\mathsf{nat}} z{+}1; \mathsf{return}\,(z{+}1)))) \\ & : & \{I\} \\ & & \quad t{:}\Sigma_{inv:\mathsf{nat}\to\mathsf{heap}\to\mathsf{prop}}.\, v{:}\mathsf{nat}.\ \{inv\ v\}r{:}\mathsf{nat}\{\lambda h.\,(inv\ (v{+}1)\ h) \wedge r = v{+}1\} \\ & & \{\lambda i.\, I * (\mathsf{fst}\ t\ 0)\} \end{array}$$

E_1 differs from E in that it also defines the invariant $inv = \lambda v.\, x \mapsto_{\mathsf{nat}} v$. When used in the specifications, inv brings out the important aspects of the local store, which are the last computed natural number v, and the fact that initially $v = 0$ (as the separating conjunct ($\mathsf{fst}\ t\ 0$) in the postcondition formally states because $\mathsf{fst}\ t = inv$). However, the type of E_1 hides the existence of the local reference x which stores v. In fact, from the outside, there is no reason to believe that the local store of E_1 consists of only one location. We could imagine a similar program E_2 that maintains two different locations x and y, increases them at every call, and returns their mean. Such a program will have a different invariant $inv = \lambda v.\, x \mapsto_{\mathsf{nat}} v * y \mapsto_{\mathsf{nat}} v$ for its local store. However, because the type abstracts over the invariant, E_1 and E_2 would have the same type. The equal types hint that the two programs would be observationally equivalent, i.e. they could freely be interchanged in any context. We do not prove this property here, but it is intriguing future work, related to the recent result of Yoshida, Honda and Berger [31, 4] on observational completeness of Hoare Logic.

In the next example, we consider an HTT equivalent of the following SML program $\lambda f{:}(\mathsf{unit}\to\mathsf{unit})\to\mathsf{unit}.\ \mathsf{let\ val}\ x = \mathsf{ref}\ 0\ \mathsf{val}\ g = \lambda y.\, x := !x + 1; ()\ \mathsf{in}\ f\ g$. The HTT specification should bring out the property that the argument function f can only access the local reference x by invoking g. Part of the problem is similar to that with E; the local state of g must be abstracted in order to make the dependence on x invisible to f. However, this is not sufficient. Because we evaluate $f\ g$ at the end, we need to know how f uses g, in order to describe the postcondition for the whole program. In other words, we also need to provide an invariant for f, *which is a higher-order predicate, because it depends on the invariant of g.*

One possible HTT implementation is as follows.

$$\begin{array}{rcl} F & = & \lambda f.\, \mathsf{do}\,(\, x \leftarrow \mathsf{alloc\ nat}\ 0; \\ & & \qquad\qquad g = (\lambda v.\, x \mapsto_{\mathsf{nat}} v, \mathsf{do}\,(z \Leftarrow !_{\mathsf{nat}}\, x; x :=_{\mathsf{nat}} z{+}1; \mathsf{return}\ (\,))); \\ & & \qquad\qquad \mathsf{eval}\,((\mathsf{snd}\ f)\ g)) \\ & : & \Pi f{:}\Sigma p{:}\mathsf{nat}{\rightarrow}(\mathsf{nat}{\rightarrow}\mathsf{heap}{\rightarrow}\mathsf{prop}){\rightarrow}\mathsf{heap}{\rightarrow}\mathsf{prop}. \\ & & \qquad \Pi g{:}\Sigma inv{:}\mathsf{nat}{\rightarrow}\mathsf{heap}{\rightarrow}\mathsf{prop}.\, v{:}\mathsf{nat}.\,\{inv\ v\}r{:}1\{inv\ (v+1)\}. \\ & & \qquad\quad w{:}\mathsf{nat}.\,\{\mathsf{fst}\ g\ w\}s{:}1\{p\ w\ (\mathsf{fst}\ g)\}. \\ & & \{I\}t{:}1\{\lambda i.\, I * \lambda h.\, \exists x{:}\mathsf{nat}.\,(\mathsf{fst}\ f)\ 0\ (\lambda v.\, x \mapsto_{\mathsf{nat}} v)\ h\} \end{array}$$

In this program, f and g carry the invariants of their local states (e.g., $p = \mathsf{fst}\ f$ is the invariant of $\mathsf{snd}\ f$ and $inv = \mathsf{fst}\ g = \lambda v.\, x \mapsto_{\mathsf{nat}} v$ is the invariant of $\mathsf{snd}\ g$). The predicate p takes a natural number n and an argument inv, and returns a description of the state obtained after applying f to g in a state where $inv(n)$ holds. The postcondition for F describes the ending heap as $p\ 0\ inv$ thus revealing that initially the local reference x stores the value 0. The last two examples show that HTT can hide, but also reveal information about local state when needed.

4 Type system

The type system presented in this paper extends our previous work [22] with several features associated with the ECC [14]. The extensions include dependent sums and subset types, as well as the type prop of assertions, the type mono of small types, and the ability to compute with elements of both of these types. The additions introduce non-trivial changes in the equational reasoning of HTT. This involves the algorithms for computing canonical forms (a canonical form of an expression is its beta-reduced and eta-long version), as well as the corresponding proof of soundness. The type system of HTT consists of the following judgments: (1) $\Delta \vdash K \Rightarrow A\,[N']$ *infers* that K is an elim term of type A, and N' is its canonical form. A and N' are synthesized as outputs of the judgment. (2) $\Delta \vdash M \Leftarrow A\,[M']$ *checks* that M is an intro term of type A, and computes the canonical form M'. (3) $\Delta; P \vdash E \Rightarrow x{:}A.\, Q\,[E']$ *infers* that E is a computation with result $x{:}A$, precondition P, *strongest* postcondition Q, and canonical form E'. Q and E' are synthesized as outputs. (4) $\Delta; P \vdash E \Leftarrow x{:}A.\, Q\,[E']$ *checks* that E is a computation with result $x{:}A$, precondition P and postcondition (not necessarily strongest) Q. The canonical form E' is the output. (5) $\Delta \Longrightarrow P$ defines when the assertion P is true. It implements classical higher-order logic. (6) $\vdash \Delta\ \mathsf{ctx}\,[\Delta']$ states that Δ is a well-formed variable context, with canonical form Δ'. (7) $\Delta \vdash A \Leftarrow \mathsf{type}\,[A']$ states that A is a well-formed type, with canonical form A'. As can be noticed, the computation of canonical forms is hard-wired into the judgments, so that it becomes part of type checking. However, space precludes us from presenting the full details about canonical forms here. In the following text, we illustrate the typing rules of HTT, but we ignore the canonical forms and other aspects of equational reasoning (i.e., we omit from the judgments the information enclosed in [brackets]). The complete details can be found in the technical report [21].

The type system implements bidirectional typechecking [25, 28], to automatically compute a significant portion of omitted types. A fragment of the rules is given in the figure below.

$$\frac{}{\Delta, x{:}A, \Delta_1 \vdash x \Rightarrow A}\ \mathsf{var} \qquad \frac{\Delta, x{:}A \vdash M \Leftarrow B}{\Delta \vdash \lambda x.\, M \Leftarrow \Pi x{:}A.\, B}\ \Pi\mathsf{I}$$

$$\frac{\Delta \vdash K \Rightarrow \Pi x{:}A.\, B \qquad \Delta \vdash M \Leftarrow A}{\Delta \vdash K\ M \Rightarrow [M/x]B}\ \Pi\mathsf{E} \qquad \frac{\Delta \vdash M \Leftarrow A \qquad \Delta \vdash N \Leftarrow [M/x]B}{\Delta \vdash (M, N) \Leftarrow \Sigma x{:}A.\, B}\ \Sigma\mathsf{I}$$

$$\frac{\Delta \vdash K \Rightarrow \Sigma x{:}A.\, B}{\Delta \vdash \mathsf{fst}\ K \Rightarrow A}\ \Sigma\mathsf{E1} \qquad \frac{\Delta \vdash K \Rightarrow \Sigma x{:}A.\, B}{\Delta \vdash \mathsf{snd}\ K \Rightarrow [\mathsf{fst}\ K/x]B}\ \Sigma\mathsf{E2}$$

$$\frac{\Delta \vdash M \Leftarrow A \qquad \Delta \Longrightarrow [M/x]P}{\Delta \vdash \mathsf{in}\ M \Leftarrow \{x{:}A.\, P\}}\ \{\}\mathsf{I} \qquad \frac{\Delta \vdash K \Rightarrow \{x{:}A.\, P\}}{\Delta \vdash \mathsf{out}\ K \Rightarrow A}\ \{\}\mathsf{E1}$$

$$\frac{\Delta \vdash K \Rightarrow \{x{:}A.\, P\}}{\Delta \Longrightarrow [\mathsf{out}\ K/x]P}\ \{\}\mathsf{E2} \qquad \frac{\Delta \vdash K \Rightarrow A \qquad A = B}{\Delta \vdash K \Leftarrow B}\ \Rightarrow\Leftarrow$$

$$\frac{\Delta \vdash A \Leftarrow \mathsf{type} \qquad \Delta \vdash M \Leftarrow A}{\Delta \vdash M : A \Rightarrow A}\ \Leftarrow\Rightarrow$$

In general, the typing rules for elim terms break down the type when read from premise to the conclusion. In the base case, the type of a variable can always be read off from the context, and therefore, elim terms can always synthesize their types. Dually, the typing rules for intro terms break down a type when read from the conclusion to the premise. If the conclusion type is given, the types for the premises can be computed and need not be provided.

When considering an elim term that happens to be intro (i.e. has the form $M{:}A$), the rule $\Leftarrow\Rightarrow$ synthesizes the type A, assuming that M checks against it. Conversely, when checking an intro term that happens to be be elim (i.e. has form K) against a type B, the rule $\Rightarrow\Leftarrow$ synthesizes the type A for K and explicitly compares if $A = B$. This comparison invokes the equational reasoning, which we do not explain here. It suffices to say that the equations used in this reasoning are derived from the usual alpha, beta and eta laws for pure functions and pairs, and the generic monadic laws [19] for the Hoare types (i.e., the unit laws and associativity).

We next describe the typing judgments for the impure fragment. The main intuition here is that a computation E may be seen as a heap transformer, because its execution turns the input heap into the output heap. The judgment $\Delta; P \vdash E \Rightarrow x{:}A.\, Q\, [E']$ essentially converts E into the equivalent binary relation on heaps, so that the assertion logic can reason about E using standard mathematical machinery for relations. The predicates $P, Q{:}\mathsf{heap}{\to}\mathsf{heap}{\to}\mathsf{prop}$ represent binary heap relations. P is the starting relation onto which the typing rules build as they convert E one command at a time. The generated strongest postcondition Q is the relation that most precisely captures the semantics of E. The judgment $\Delta; P \vdash E \Leftarrow x{:}A.\, Q\, [E']$ checks if Q is a postcondition for E, by generating the strongest postcondition S and then trying to prove the implication $S \Longrightarrow Q$ in the assertion logic.

Given $P, Q, S{:}\mathsf{heap}{\to}\mathsf{heap}{\to}\mathsf{prop}$, and $R, R_1, R_2{:}\mathsf{heap}{\to}\mathsf{prop}$ we define the following predicates of type $\mathsf{heap}{\to}\mathsf{heap}{\to}\mathsf{prop}$.

$$\begin{array}{lcl}
P \circ Q & = & \lambda i.\, \lambda m.\, \exists h{:}\mathsf{heap}.\, (P\ i\ h) \wedge (Q\ h\ m) \\
R_1 \multimap R_2 & = & \lambda i.\, \lambda m.\, \forall h{:}\mathsf{heap}.\, (R_1 * \lambda h'.\, h' = h)\ (i) \supset (R_2 * \lambda h'.\, h' = h)\ (m) \\
R \gg Q & = & \lambda i.\, \lambda m.\, \forall h{:}\mathsf{heap}.\, (\lambda h'.\, R(h') \wedge h = h') \multimap Q(h)
\end{array}$$

$P \circ Q$ is standard relational composition. $R_1 \multimap R_2$ is the relation that selects a fragment R_1 from the input heap, and *replaces* it with some fragment R_2 in the output heap. We will use this relation to describe the action of memory update, where the old value stored into the memory must be replaced with the new value. The relation $R \gg Q$ selects a fragment R of the input heap, and then behaves like Q on that fragment. This captures precisely the semantics of the "most general" computation of Hoare type $\{R\}x{:}A\{Q\}$, in the small footprint semantics, leading to the following typing rules.

$$\frac{\Delta; \lambda i.\, \lambda m.\, i = m \wedge (R * \lambda h'.\, \top)(m) \vdash E \Leftarrow x{:}A.\, (R \gg Q)}{\Delta \vdash \mathsf{do}\, E \Leftarrow \{R\}x{:}A\{Q\}}$$

$$\frac{\Delta \vdash K \Rightarrow \{R\}x{:}A\{S\} \qquad \Delta, i{:}\mathsf{heap}, m{:}\mathsf{heap}, (P\ i\ m) \Longrightarrow (R * \lambda h'.\, \top)(m) \qquad \Delta, x{:}A; P \circ (R \gg S) \vdash E \Rightarrow y{:}B.\, Q}{\Delta; P \vdash x \leftarrow K; E \Rightarrow y{:}B.\, (\lambda i.\, \lambda m.\, \exists x{:}A.\, (Q\ i\ m))}$$

To check if $\mathsf{do}\, E$ has type $\{R\}x{:}A\{Q\}$, we verify that E has a postcondition $R \gg Q$. The checking is initialized with a relation stating that the initial heap $i = m$ contains a sub-fragment satisfying R (c.f., the conjunct $(R * \lambda h'.\, \top)(m)$).

To check $x \leftarrow K; E$, where K has type $\{R\}x{:}A\{S\}$, we must first prove that the beginning heap contains a sub-fragment satisfying R so that K can be executed at all (c.f. $(P\ i\ m) \Longrightarrow (R * \lambda h'.\, \top)(m)$). The strongest postcondition for K, is $P \circ (R \gg S)$, which is taken as the precondition for checking E.

$$\frac{\Delta \vdash M \Leftarrow A}{\Delta; P \vdash \mathsf{return}\, M \Rightarrow x{:}A.\, (\lambda i.\, \lambda m.\, (P\ i\ m) \wedge x =_A M)}$$

$$\frac{\Delta \vdash \tau \Leftarrow \mathsf{mono} \qquad \Delta \vdash M \Leftarrow \mathsf{nat} \qquad \Delta, i{:}\mathsf{heap}, m{:}\mathsf{heap}, (P\ i\ m) \Longrightarrow (M \hookrightarrow_\tau -)(m) \qquad \Delta, x{:}\tau; \lambda i.\, \lambda m.\, (P\ i\ m) \wedge (M \hookrightarrow_\tau x)(m) \vdash E \Rightarrow y{:}B.\, Q}{\Delta; P \vdash x \Leftarrow !_\tau\, M; E \Rightarrow y{:}B.\, (\lambda i.\, \lambda m.\, \exists x{:}\tau.\, (Q\ i\ m))}$$

$$\frac{\Delta \vdash \tau \Leftarrow \mathsf{mono} \qquad \Delta \vdash M \Leftarrow \mathsf{nat} \qquad \Delta \vdash N \Leftarrow \tau \qquad \Delta, i{:}\mathsf{heap}, m{:}\mathsf{heap}, (P\ i\ m) \Longrightarrow (M \hookrightarrow -)(m) \qquad \Delta; P \circ ((M \mapsto -) \multimap (M \mapsto_\tau N)) \vdash E \Rightarrow y{:}B.\, Q}{\Delta; P \vdash M :=_\tau N; E \Rightarrow y{:}B.\, Q}$$

The postcondition for the trivial, pure, computation $\mathsf{return}\, M$ includes the precondition (as M does not change the heap) but must also state that M is the return value. Before the lookup $x = !_\tau\, M$, we must prove that M points to a value of type τ at the beginning (c.f., $(P\ i\ m) \Longrightarrow (M \hookrightarrow_\tau -)(m)$). After the lookup, the heap looks exactly as before $(P\ i\ m)$ but we also know that x equals the content of M, that is, $(M \hookrightarrow_\tau x)(m)$. Before the update $M :=_\tau N$, we must prove that M is allocated and initialized with some value *with an arbitrary type* (i.e., $(M \hookrightarrow -)(m)$). After the lookup, the old value is removed from the heap, and replaced with N, that is $(P \circ ((M \mapsto -) \multimap (M \mapsto_\tau N)))$.

Finally, we briefly illustrate the judgment $\Delta \Longrightarrow P$, which defines the assertion logic of HTT in the style of natural deduction. The assertion logic contains the rules for introduction and elimination of implication and the universal quantifier, and the rest of the propositional constructs are formalized using axiom schemas that encode the standard introduction and elimination rules. We present here the axioms for conjunction and heterogeneous equality.

$$\begin{array}{llll} \mathsf{andi} & : \forall p, q{:}\mathsf{prop}.\, p \supset q \supset p \wedge q & \mathsf{ande} & : \forall p, q, r{:}\mathsf{prop}.\, p \wedge q \supset (p \supset q \supset r) \supset r \\ \mathsf{xidi}_A & : \forall x{:}A.\, \mathsf{xid}_{A,A}(x, x) & \mathsf{xide}_A & : \forall p{:}A{\rightarrow}\mathsf{prop}.\, \forall x, y{:}A.\, \mathsf{xid}_{A,A}(x, y) \supset p\ x \supset p\ y \end{array}$$

For each index type A, the axiom xidi_A asserts the reflexivity of the equality relation, and the axiom xide_A asserts that equal values are not distinguishable by any arbitrary propositional contexts. The logic includes axioms for extensionality of functions and pairs, Peano arithmetic, booleans and excluded middle.

We conclude this section with an informal description of the main theoretical result of the paper, which relates typechecking with evaluation.

Theorem 1 (Soundness). *The type system of HTT is sound, in the following sense: if $\Delta; P \vdash E \Leftarrow x{:}A.\, Q\, [E']$, and E terminates when evaluated in a heap i satisfying $P\ i\ i$, then the resulting heap m satisfies $Q\ i\ m$.*

Obviously, to establish this theorem, we must first define formally the operational semantics for HTT. Then the theorem follows from the Preservation and Progress lemmas, which take the customary form, but are much harder to prove than in the usual simply-typed setting. For example, Preservation must establish not only that evaluation preserves types, but also postconditions of effectful computations, as well as the canonical forms of pure terms. On the other hand, the Progress lemma first requires showing that the assertion logic of HTT is sound. This assertion logic is a higher-order logic over heaps, and its soundness basically implies that our axiomatization indeed correctly captures the properties of the real heaps encountered during evaluation. In particular, if we have proved that a certain location exists at a given program point, then when that program point is reached, we can safely take an operational step and dereference the location. We establish the soundness of the assertion logic, by developing a crude set-theoretic model based on the standard approach to modeling ECC. The interested reader is referred to the accompanying technical report [21] for full details of the proofs.

5 Conclusions and related work

In this paper we present an extension of our Hoare Type Theory (HTT) [22], with higher-order predicates, and allow quantification over abstract predicates at the level of terms, types and assertions. This significantly increases the power of the system to encompass definition of inductive predicates, abstraction of program invariants, and even first-class modules that can contain not only types and terms, but also axioms over types and terms. The novel application of this type system is to express sharing of local state between functions and/or datatypes, and transfer of state ownership between datatypes and the memory manager.

We have already discussed related work on program logics for higher-order, effectful programs in Section 1, as well as work on verification tools and languages (e.g., Spec#, ESC/Java, JML, and so on) aimed at integrating Hoare-like reasoning with type checking. The work on dependently typed systems with stateful features, has mostly focused on how to appropriately restrict the language so that effects do not pollute the types. Such systems have mostly employed singleton types to enforce purity. Examples include Dependent ML by Xi and Pfenning [30], Applied Type Systems by Xi et al. [29, 32], a type system for certified binaries by Shao et al. [27], and the theory of refinements by Mandelbaum et al. [15]. HTT differs from all these approaches, because types are allowed to depend on monadically encapsulated effectful computations.

We mention that HTT may be obtained by adding effects and the Hoare type to the Extended Calculus of Constructions (ECC) [14]. There are some differences between ECC and the pure fragment of HTT, but they are largely inessential. For example, HTT uses classical assertion logic, whereas ECC is intuitionistic, but consistent with classical extensions. The latter has been demonstrated in Coq [16] which implements and subsumes ECC. Also, HTT contains only two type universes (small and large types), while ECC is more general, and contains the whole infinite tower. However, we expect that it should be simple to extend HTT to the full tower of universes.

Finally, we mention here the representative work of Ni and Shao [23] and Filliâtre [10] who implement Hoare-style reasoning in Coq. Ni and Shao use Coq to verify properties of assembly code, while Filliâtre exploits Coq tactics and decision procedures to partially automate the verification of imperative programs. We note that these two approaches are fundamentally different from ours, as they impose an additional level of indirection. Where they use type theory to axiomatize Hoare-style reasoning, we integrate Hoare logic within the type system of the underlying language, so that specifications become an integral part of programming.

References

1. M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In *CASSIS 2004*. LNCS. Springer, 2004.
2. M. Barnett and D. Naumann. Friends need a bit more: Maintaining invariants over shared state. In *Mathematics of Program Construction*, LNCS 3125, 2004.
3. N. Benton. Abstracting Allocation: The New new Thing. In CSL'06.
4. M. Berger, K. Honda, and N. Yoshida. A logical analysis of aliasing in imperative higher-order functions. In ICFP'05, pages 280–293.
5. B. Biering, L. Birkedal, and N. Torp-Smith. BI hyperdoctrines, Higher-Order Separation Logic, and Abstraction. ITU-TR-2005-69, IT University, Copenhagen.

6. L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer*, 7(3):212–232, June 2005.
7. R. DeLine and M. Fahndrich. Enforcing high-level protocols in low-level software. In PLDI'01, pages 59–69, 2001.
8. D. L. Detlefs, K. R. M. Leino, G. Nelson, and J. B. Saxe. Extended static checking. Compaq Systems Research Center, Research Report 159, December 1998.
9. D. Evans and D. Larochelle. Improving security using extensible lightweight static analysis. *IEEE Software*, 19(1):42–51, 2002.
10. J.-C. Filliâtre. Verification of non-functional programs using interpretations in type theory. *Journal of Functional Programming*, 13(4):709–745, July 2003.
11. J. Harrison. Inductive definitions: automation and application. In *Higher Order Logic Theorem Proving and Its Applications*, LNCS 971, Springer, 1995.
12. T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of C. *USENIX Annual Technical Conference*, 2002.
13. N. Krishnaswami. Separation logic for a higher-order typed language. SPACE'06.
14. Z. Luo. *An Extended Calculus of Constructions.* PhD thesis, U of Edinburgh, 1990.
15. Y. Mandelbaum, D. Walker, and R. Harper. An effective theory of type refinements. In ICFP'03, pages 213–226.
16. The Coq development team. *The Coq proof assistant reference manual.* LogiCal Project, 2004. Version 8.0.
17. C. McBride. *Dependently Typed Functional Programs and their Proofs.* PhD thesis, University of Edinburgh, 1999.
18. J. C. Mitchell. *Foundations for Programming Languages.* MIT Press, 1996.
19. E. Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.
20. G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to typed assembly language. *TOPLAS*, 21(3):527–568, 1999.
21. A. Nanevski, A. Ahmed, G. Morrisett, and L. Birkedal. Abstract predicates and mutable ADTs in Hoare Type Theory. TR-14-06, Harvard University. Available at `http://www.eecs.harvard.edu/~aleks/papers/hoarelogic/htthol.pdf`.
22. A. Nanevski, G. Morrisett, and L. Birkedal. Polymorphism and separation in Hoare Type Theory. In ICFP'06, pages 62–73.
23. Z. Ni and Z. Shao. Certified assembly programming with embedded code pointers. In POPL'06, pages 320–333.
24. P. W. O'Hearn, H. Yang, and J. C. Reynolds. Separation and information hiding. In POPL'04, pages 268–280.
25. B. C. Pierce and D. N. Turner. Local type inference. *TOPLAS*, 22(1):1–44, 2000.
26. J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In LICS'02, pages 55–74.
27. Z. Shao, V. Trifonov, B. Saha, and N. Papaspyrou. A type system for certified binaries. *TOPLAS*, 27(1):1–45, January 2005.
28. K. Watkins, I. Cervesato, F. Pfenning, and D. Walker. A concurrent logical framework: The propositional fragment. LNCS 3085, Springer 2004.
29. H. Xi. Applied Type System (extended abstract). LNCS 3085, 2004.
30. H. Xi and F. Pfenning. Dependent types in practical programming. POPL'99.
31. N. Yoshida, K. Honda, and M. Berger. Logical reasoning for higher-order functions with local state. Personal communication, August 2006.
32. D. Zhu and H. Xi. Safe programming with pointers through stateful views. In PADL'05, pages 83–97.