Dependent Type Theory of Stateful Higher-Order Functions

Aleksandar Nanevski and Greg Morrisett
Harvard University
{aleks|greg}@eecs.harvard.edu

Abstract

We present a dependent Hoare Type Theory (HTT) which provides support for reasoning about programs with higher-order functions and effects, including non-termination, state with aliasing and pointer arithmetic. The type structure encapsulates effectful commands using a monad indexed by pre- and post-conditions in the style of Hoare logic. The theory carefully distinguishes between an appropriate notion of definitional equality and propositional equality, in order to maintain the relative decidability of type-checking.

1 Introduction

This paper describes a type theoretic approach to developing a programming language for higher-order stateful functions together with its associated program logic.

Hoare logic [10] has been most successfully employed to reason about first-order imperative programs, whereas extensions to programs with procedures and more generally higher-order functions have been known to be subtle [5, 7, 6, 1, 2]. As discussed by Apt in his survey of Hoare Logic, the problems usually appear because "*... the semantics of parameter passing is always modeled syntactically by some form of variable substitution in a program, and this leads to various subtle problems concerning variable clashes. These difficulties are especially acute in the presence of recursion and static scoping*" [2, page 462]. That is, the modularity features that make typed lambda calculus so successful appear to be in direct conflict with Hoare-style reasoning.

The goal of this work is to marry dependent type theory and Hoare logic in a fashion that preserves the strengths of each. In particular, we seek a Hoare-like logic for higher-order, imperative programs while retaining the proof-theoretic reasoning of the Curry-Howard isomorphism. To that end, we describe a system called HTT (short for Hoare Type Theory) that revolves around a pure, dependently-typed functional core extended with imperative commands, together with the associated type theory which is extended to account for encapsulation of effects and logical specifications of their behavior. The specifications are based upon the classical work of Cartwright and Oppen [6] (first-order, multi-sorted classical logic with McCarthy-style arrays [16]). We use classical, rather than a substructural logic (e.g. Separation logic [22, 25, 23]), mainly because it has a well-studied proof theory, and has been employed in several practical verification systems, like ESC/Java [14, 8] and Cyclone [12], but we believe the framework is extensible to other logics.

At any rate, our main focus and contribution is not any particular assertion logic, but the formulation of HTT as a dependent type theory which reconciles state and higher-order functions. This formulation has several important consequences: First, the treatment of variables at the term, command, type, and specification levels is handled uniformly which avoids many inconvenient side conditions that are usually present in Hoare logics. Second, the encapsulation of effects, via an indexed monad [17, 27, 13], makes it possible for both types and specifications to depend upon terms (including effectful computations). Third, the language treats heap addresses as natural numbers, and thus easily supports reasoning about aliasing and pointer arithmetic. Fourth, the separation of definitional and propositional equality on terms makes it possible to reduce the decidability of type-checking to provability in the specification logic, and soundness of the type system to soundness of the specification logic. Fifth, HTT re-establishes the Curry-Howard isomorphism between logic and higher-order functional programming with side effects, in the important special case when the side effects in question concern nontermination and state with first-class locations and aliasing. This naturally leads to a degree of modularity in the specification and reasoning about programs. Finally, type theory in general has certain advantages over Hoare logic. To mention but an example, in type theory, data invariants can be captured in the types to facilitating abstraction and program re-use. In Hoare logic, data invariants can only be specified in the pre- and postconditions of the code, which may make it cumbersome to write specifications that are parametric in

the data invariants.

To achieve the features described above, HTT is split into two fragments. The first fragment consist of constructs for which we can employ Hoare-like reasoning by pre- and post-conditions. This includes the stateful commands, conditionals and recursion. The second fragment consists only of pure, dependent, higher-order functions, and the equational reasoning about this fragment follows the traditional approach of type theory. Stateful computations can be internalized into the pure fragment, by means of a monadic type constructor $\{P\}x{:}A\{Q\}$. This type classifies suspended computations that when executed in a state satisfying the proposition P, produce a value of type A and a new state satisfying Q, if they terminate. The variable x names the return value of the computation, and Q may depend on x. Our approach is based on the judgmental presentation of monads by Pfenning and Davies [24], and we also employ a dependently typed formulation with canonical forms and hereditary substitutions as proposed by Watkins et al. [28], with significant extensions that are particular to our application.

The main monadic judgment, and the type $\{P\}x{:}A\{Q\}$ which internalizes it, may be seen as a formalization of a verification condition generator in the style of Necula [20]. This sets HTT apart from the usual approaches to Hoare logics, where the meaning of Hoare triples is defined semantically based on program evaluation. Because in HTT the Hoare triples are types, arranging the theory around verification condition generation, rather than evaluation, avoids the circular dependence between typing and evaluation, and preserves the predicative nature of the system.

We have proven progress and preservation theorems for the language with respect to a standard, call-by-value interpretation, thus establishing the soundness of HTT (relative to the soundness of the underlying logic of assertions used in the Hoare triples). The detailed technical development of HTT is presented in the accompanying report [18].

2 Overview

In this section we describe the constructs of our Hoare type theory, the intuition behind our memory model, and define several different notions of substitution that will be used in the later sections to provide the semantic foundations. We start by presenting the HTT syntax in Table 1.

Types and propositions. The primitive types of HTT are natural numbers and booleans. We also have the unit type 1, function type $\Pi x{:}A.\ B$ (where B can depend on the variable x) and computation type $\{P\}x{:}A\{Q\}$.

The type $\{P\}x{:}A\{Q\}$ classifies computations that execute under a precondition P and, if they terminate, return a value $x{:}A$ and a postcondition Q. Here, P and Q are propositions that describe and relate the properties of the memory (i.e., the heap) at the beginning and the end of the computation. The variable x is bound in the type, and its scope extends through the postcondition Q.

Types	A, B, C	$::=$	$\mathsf{bool} \mid \mathsf{nat} \mid 1 \mid \Pi x{:}A.\ B \mid \{P\}x{:}A\{Q\}$
Propositions			
	P, Q, R, I	$::=$	$\mathsf{Id}_A(M,N) \mid \mathsf{Hld}(H,G) \mid \mathsf{indom}(H,M)$
			$\mid \top \mid \bot \mid P \wedge Q \mid P \vee Q \mid P \supset Q \mid \neg P$
			$\mid \exists x{:}A.\ P \mid \forall x{:}A.\ P \mid \exists h{:}\mathsf{heap}.\ P \mid \forall h{:}\mathsf{heap}.\ P$
Heaps	H, G	$::=$	$h \mid \mathsf{emp} \mid \mathsf{upd}_A(H,M,N)$
Elim terms	K, L	$::=$	$x \mid K\ M \mid M : A$
Intro terms			
	M, N, O	$::=$	$K \mid (\,) \mid \lambda x.\ M \mid \mathsf{dia}\ E \mid \mathsf{true} \mid \mathsf{false}$
			$\mid \mathsf{eq}(M,N) \mid \mathsf{z} \mid \mathsf{s}\ M \mid M + N \mid M \times N$
Computations			
	E, F	$::=$	$M \mid \mathsf{let\ dia}\ x = K\ \mathsf{in}\ E \mid c; E$
Commands	c	$::=$	$x = \mathsf{alloc}_A(M) \mid x = [M]_A$
			$\mid [M]_A = N \mid x = \mathsf{if}_A(M, E_1, E_2)$
			$\mid x = \mathsf{fix}_A(f.y.F, M)$
Variable contexts	Δ	$::=$	$\cdot \mid \Delta, x{:}A$
Heap contexts	Ψ	$::=$	$\cdot \mid \Psi, h$
Prop contexts	Γ	$::=$	$\cdot \mid \Gamma, P$

Table 1. Syntax of HTT.

The syntactic category of propositions contains the primitive propositions $\mathsf{Id}_A(M,N)$, $\mathsf{Hld}(H,G)$ and $\mathsf{indom}(H,M)$. The first asserts the equality of terms M and N of type A, and we refer to it as *propositional equality*. The second asserts the equality of the heaps H and G; it is *propositional heap equality*. The third asserts that the heap H allocates a chunk of memory at address given by the integer M; however, it does not state the type of the value stored at that address. The rest of the propositional constructs includes the standard classical connectives, together with quantification over types and quantification over heaps. Keeping with the tradition of Hoare logic, we will frequently refer to propositions used in the computation types as *assertions*.The assertion logic is multi-sorted, where the sorts include heaps and all the elements of the type hierarchy.

Locations and the memory model. Each heap is a *finite* collection of *assignments* $M \mapsto_A N$, where A is a type, M is a natural denoting an address in the heap, and N is an element of type A. If a heap contains an assignment $M \mapsto_A N$, we say that the M points to N. Heaps are partial functions, as each address can point to at most one term.

The syntax of HTT provides the following constructs to represent heaps: emp is the empty heap, and $\mathsf{upd}_A(H,M,N)$ is a heap in which the location M points to N of type A, but all the other assignments equal those of

the heap H. We use h to range over heap variables.

We now define several propositions that will appear prominently in HTT assertions.

$$\begin{array}{rcl} \mathsf{seleq}_A(H, M, N) & = & \exists h{:}\mathsf{heap}.\ \mathsf{Hld}(H, \mathsf{upd}_A(h, M, N)) \\ \mathsf{seleq}_A(H, M, -) & = & \exists x{:}A.\ \mathsf{seleq}_A(H, M, x) \\ M \in \mathsf{dom(H)} & = & \mathsf{indom}(H, M) \\ M \notin \mathsf{dom(H)} & = & \neg(M \in \mathsf{dom(H)}) \end{array}$$

Terms. Terms form the pure fragment of HTT, and contain the basic operations on primitive types (e.g. equality, arithmetic), as well as the constructs for introduction and elimination of non-primitive types like 1, $\Pi x{:}A.\ B$ and $\{P\}x{:}A\{Q\}$. Following Watkins et al. [28], we separate the syntactic categories of introduction terms (or intro terms) and elimination terms (or elim terms).

This distinction allows that a significant amount of type information may be omitted from the programs, because the types of elim terms can be inferred automatically. For intro terms, the type information may be provided explicitly by the construct $M : A$, if needed. Most importantly, this formulation naturally leads to a syntactic criterion for normality of terms: a term is in *normal form* iff it does not contain a beta redex iff it does not contain the term constructor $M : A$.

Computations. Computations form the effectful fragment of HTT. They correspond rather closely to programs in a generic imperative and sequential programming language. Intuitively, each computations is a semicolon-separated sequence of commands that perform effects and return with a result that is subsequently bound to a designated variable. The commands of HTT are: (1) $x = \mathsf{alloc}_A(M)$, which allocates a portion of the heap, and initializes it with the value of M. The address of the heap segment is bound to x; (2) $x = [M]_A$, looks up the contents of the location M, and stores the value in x. Before the lookup can be performed, we must prove that M points to a value of type A; (3) $[M]_A = N$, mutates the contents of the heap space assigned to the location M, by writing N (of type A) into it; (4) $x = \mathsf{if}_A(M, E_1, E_2)$, is a conditional with branches E_1 and E_2 guarded by the Boolean term M; (5) $x = \mathsf{fix}_A(f.y.F, M)$ is a recursion construct, adapted for the monadic presentation of HTT. Here A is a type of the form $A = \Pi z{:}B.\{R_1\}x{:}C\{R_2\}$. The semantics of fix first defines the function f of type A, which is the least fixed point of the equation $f = \lambda y.\ \mathsf{dia}\ F$. The function f is immediately applied to $M : B$ to obtain a computation that is subsequently executed and its result (of type $[M/z]C$, where $[M/z]$ is a capture-avoiding substitution of M for z) is bound to x; (6) The computation that simply consist of the term M is the trivial computation that immediately returns M, without performing any effects; (7) The construct $\mathsf{let\ dia}\ x = K\ \mathsf{in}\ E$ sequentially composes the computation encapsulated in the term K, with the computation E. This construct is the elimination form for the computation types.

In the usual presentation of the monadic lambda calculus [17, 27, 13], the last two constructs above correspond to the monadic *unit* and *bind*, respectively. The term constructor $\mathsf{dia}\ E$ suspends a computation E, and coerces it into the pure fragment. This is the introduction form for the monadic types. The HTT formulation results in an easier proof theory and is directly adopted from the work of Pfenning and Davies [24].

Reductions, expansions and substitutions. The equational theory of HTT is based on beta reductions and eta expansions for the various non-primitive types. In case of the unit type and function types, the reductions and expansions are rather standard. In the case of computation types, eta expansion is given as in the equation below, where we assume that y is not a free variable of $M : \{P\}x{:}A\{Q\}$.

$$M : \{P\}x{:}A\{Q\} \implies_\eta \mathsf{dia}\ (\mathsf{let\ dia}\ y = M : \{P\}x{:}A\{Q\}\ \mathsf{in}\ y)$$

Beta reduction for computations is a bit more involved, as it needs to implement a sequential composition of two computations. To this end, we define an appropriate auxiliary operation of *monadic substitution* $\langle E/x : A\rangle F$, which composes E and F sequentially. The operation is defined by induction on the structure of E, and we again follow the presentation of Pfenning and Davies [24].

$$\begin{array}{ll} \langle M/x : A\rangle F & = [M : A/x]F \\ \langle \mathsf{let\ dia}\ y = K\ \mathsf{in}\ E/x : A\rangle F & = \mathsf{let\ dia}\ y = K\ \mathsf{in}\ \langle E/x : A\rangle E \\ \langle c; E/x : A\rangle F & = c; \langle E/x : A\rangle F \end{array}$$

With the monadic substitution defined, the beta reduction for computation types is given as:

$$\mathsf{let\ dia}\ x = \mathsf{dia}\ E : \{P\}y{:}A\{Q\}\ \mathsf{in}\ F \implies_\beta \langle E/x : A\rangle F$$

Examples. Our first example presents the HTT version of a function for swapping the contents of two (possibly aliased) locations. We allow the preconditions in the computation types to depend on one heap variable mem which stands for the heap at the time when the computation starts. The postcondition may depend on two heap variables: init stands for the heap prior to the computation, and mem stands for the heap obtained after the computation. These three variables are assumed bound by the precondition and postcondition respectively.

```
swap : Πx:nat. Πy:nat.
       {seleq(mem, x, -) ∧ seleq(mem, y, -)} r : 1
       {∀v1:A, v2:A. seleq(init, x, v1) ∧ seleq(init, y, v2) ⊃
              Hld(mem, upd(upd(init, x, v2), y, v1))} =
       λx. λy. dia(v1 = [x]; v2 = [y];
                   [y] = v1; [x] = v2; ())
```

The function swap takes the locations x and y, and then returns a suspended computation, which, when activated,

binds the contents of x and y to variables v_1 and v_2 respectively, and writes them back into the memory, but in a swapped order. We have implicitly assumed that all the seleq and upd constructs out of x and y are indexed by some type A (which we omit for brevity).

Activation of suspended computations is forced with the letdia construct, as in the following function which swaps the contents of two locations, and then swaps them again:

```
identity : Πx:nat. Πy:nat.
        {seleq(mem, x, -) ∧ seleq(mem, y, -)} r : 1
        {HId(init, mem)} =
        λx. λy. dia(let dia u = swap x y
                        dia v = swap x y in ())
```

The types of these two functions deserves further comments. For example, the precondition in the Hoare types requires that x and y are locations that are actually allocated when the computation starts. If this condition is not satisfied, the computation will get stuck because it dereferences x and y.

The precondition does not specify the values that x and y point to. It is the job of the postcondition to describe how these values may be changed by the computation. More generally, the postcondition relate the starting heap with the ending heap of the computation.

At this point we would like to relate our approach to computation typing, with the classical approach of Hoare logic [10, 11, 3, 25]. In most variants of Hoare logic, the program like swap x y that swaps the contents of locations x and y would be specified with the precondition $\mathsf{seleq}(\mathsf{mem}, x, v_1) \wedge \mathsf{seleq}(\mathsf{mem}, y, v_2)$ and the postcondition $\mathsf{seleq}(\mathsf{mem}, x, v_2) \wedge \mathsf{seleq}(\mathsf{mem}, y, v_1)$, where the variables v_1 and v_2 are not bound anywhere and can appear in the assertions, but not in the program itself. Such variables are frequently called *logic* variables. Because the scope of logic variables is global, they can appear simultaneously in the precondition and the postcondition, thus establishing the connection between the starting and ending states of the computation. Logic variables are somewhat cumbersome to reconcile with the type theoretic approach, precisely because they cannot appear in programs, so we avoid them in HTT and instead relate the input and output heaps of a computation via the postcondition, as discussed above.

3 Normal forms and hereditary substitutions

Equational reasoning about terms in type theories usually requires that terms be converted into some kind of normal form before they can be compared for equality. The conversion to normal form is usually defined only on well-typed terms, making the equational reasoning and typechecking mutually dependent on each other. For HTT, we adopt the approach due to Watkins et al. [28], where equational reasoning and typechecking are disentangled, essentially by allowing normalization of terms that are not necessarily well typed. This leads to significant conceptual simplifications of the system and its meta theory.

The main idea is to consider the syntactic structure of (not necessarily well-typed) normal terms and define substitutions which preserve this structure. For example, in places where ordinary capture-avoiding substitution of a normal term into another normal term creates a redex, like $(\lambda x.\ M)\ N$, we need to continue the substitution process by substituting N for x in M. This may produce another redex, which must be immediately reduced, initiating another substitution, and so on. Following [28], we call this kind of repeated substitution operation *hereditary substitution*. To ensure termination, hereditary substitutions are parameterized by a metric based on types which is reduced as the substitution proceeds.

Hereditary substitutions will operate only on normal terms. As explained in Section 2, normal terms do not contain beta redexes, or equivalently, they do not contain the constructor $M : A$. Here, we extend the notion further by requiring that terms of a primitive type are in as simple form as possible. For example, an integer expression $\mathsf{s}\ M + N$, is not consider normalized, and it reduces to $\mathsf{s}\ (M + N)$. More generally, terms $M + N$ and $M \times N$ are normal only if M and N are normal, but different from z or $\mathsf{s}\ M'$ for some term M'. Similarly, the comparison $\mathsf{eq}(M, N)$ is normal only if at least one of the arguments M, N does not start with z or s.

We denote by $[M/x]^*_S(-)$ the hereditary substitution that substitutes a normal expression M for a variable x into a given argument. The superscript $*$ ranges over $\{a, p, k, m, e, h\}$ and determines the syntactic category of the argument (type, proposition, elim term, intro term, computation or a heap, respectively). The subscript S is a simple type which is the decreasing metric that prevents the hereditary substitutions from diverging. In the typing judgments in Section 4, S is the dependency-free version of the type associated with M and x. We also have a hereditary monadic substitution $\langle E/x\rangle_S(-)$ into computations.

Hereditary substitutions are defined by nested induction, first on the structure of S, and second on the structure of the expression being substituted into. For illustration, we only present here the cases for the hereditary substitution $[M/x]^k_S(-)$ into elim terms, which may either return an elim term, or an intro term decorated with a type S'. From this definition, it should be clear that hereditary substitutions are well-defined, as we always either decrease the index type S (in which case the expression we substitute into may become larger), or the index type remains the same,

but the expressions decrease.

$$
\begin{array}{lll}
[M/x]^k_S(x) & = M : S & \\
[M/x]^k_S(y) & = y & \text{if } y \neq x \\
[M/x]^k_S(K\ N) & = K'\ N' & \text{if } [M/x]^k_S(K) = K' \\
 & & \text{and } [M/x]^k_S(N) = N' \\
[M/x]^k_S(K\ N) & = O' : S_2 & \text{if } [M/x]^k_S(K) = \lambda y.\ M' : S_1 \to S_2 \\
 & & \text{where } S_1 \to S_2 \text{ is subexpression of } S \\
 & & \text{and } [M/x]^k_S(N) = N' \\
 & & \text{and } O' = [N'/y]^m_{S_1}(M') \\
[M/x]^k_A(K') & \text{fails} & \text{otherwise}
\end{array}
$$

Theorem 1 (Termination of hereditary substitutions)

1. *If $[M/x]^k_S(K) = N' : S_1$, then S_1 is a subexpression of S.*
2. *$[M/x]^*_S(-)$, and $\langle E/x\rangle_S(-)$ terminate, either by returning a result, or failing in a finite number of steps.*

We write $[M/x]^*_A(-)$ instead of $[M/x]^*_S(-)$ when S is the simple type obtained by erasing the dependencies in A.

4 Type system

The HTT type system uses *canonical forms* to facilitate equational reasoning [28]. Canonical form is beta normal and eta long (i.e. all of its intro subterms are eta expanded), so that comparing two terms for equality modulo beta reduction and eta expansion can be done by simply comparing the respective canonical forms for alpha equivalence.

The typing judgments of HTT synthesize the canonical forms of terms in parallel with type checking, and the synthesis employs hereditary substitutions. Hereditary substitutions were defined on normal forms, but here we restrict them to canonical forms. In [18], we show that hereditary substitutions over canonical forms produce canonical results.

The type system consists of the following judgments.

$$
\begin{array}{ll}
\Delta \vdash K \Rightarrow A\,[N'] & \vdash \Delta\ \mathsf{ctx} \\
\Delta \vdash M \Leftarrow A\,[M'] & \Delta; \Psi \vdash \Gamma\ \mathsf{pctx} \\
\Delta; P \vdash E \Rightarrow x{:}A.\ Q\,[E'] & \Delta; \Psi \vdash P \Leftarrow \mathsf{prop}\,[P'] \\
\Delta; P \vdash E \Leftarrow x{:}A.\ Q\,[E'] & \Delta \vdash A \Leftarrow \mathsf{type}\,[A'] \\
\Delta; \Psi; \Gamma_1 \Longrightarrow \Gamma_2 & \Delta; \Psi \vdash H \Leftarrow \mathsf{heap}\,[H']
\end{array}
$$

The first four judgments on the left are explicitly oriented, so that each of the involved expressions is either considered given as input, or is synthesized as output. The judgment $\Delta \vdash K \Rightarrow A\,[N']$ infers the canonical type A of the elimination term K, and synthesizes the canonical form N' of K. Of course, K is arbitrary, i.e., it is not necessarily canonical.

The judgment $\Delta\Gamma \vdash M \Leftarrow A\,[M']$ *checks* whether intro term M matches against the canonical type A. Naturally, M and A are inputs. If the two match, the canonical form M' is synthesized as output.

Similarly, $\Delta; P \vdash E \Rightarrow x{:}A.\ Q\,[E']$ infers the strongest postcondition Q for the input computation E. The precondition P and the type A are inputs (assumed canonical), and Q and E' are outputs, which will also be canonical. The judgment $\Delta; P \vdash E \Leftarrow x{:}A.\ Q\,[E']$ checks that P and Q are a pre- and postcondition for E, but Q is not required to be strongest. The computation E is arbitrary, A, P and Q are canonical inputs, and E' is the output which is the canonical form of E.

The judgment $\Delta; \Psi; \Gamma_1 \Longrightarrow \Gamma_2$ defines a sequent calculus for a multi-sorted variant of classical first-order logic. Here Ψ is a list of heap variables, and Γ_1, Γ_2 are lists of propositions. The judgment holds if assuming that all propositions in Γ_1 are true, one of the propositions in Γ_2 is true.

The judgments on the right side of the above table deal with formation of canonical contexts, proposition contexts, propositions, types and heaps, respectively. In the last three cases, the judgments return the canonical form of the input expression.

Terms. We only present the rules for the derived types of HTT, as the rules for bools and nats are trivial. We first need two auxiliary functions: $\mathsf{apply}_A(M, N)$ and $\mathsf{expand}_A(N)$. In $\mathsf{apply}_A(M, N)$, A is a canonical type, and the arguments M and N are canonical intro terms. The function normalizes the application $M\ N$, if it is well-typed.

$$
\begin{array}{lll}
\mathsf{apply}_A(K, M) & = K\ M \text{ if } K \text{ is an elim term} & \\
\mathsf{apply}_A(\lambda x.\ N, M) & = N' & \text{where } N' = [M/x]^m_A(N) \\
\mathsf{apply}_A(N, M) & \text{fails} & \text{otherwise}
\end{array}
$$

In the function $\mathsf{expand}_A(N)$, A is a canonical type, and the argument N is a term. The function turns N into a canonical intro term (if it is not already) by computing its eta long form.

$$
\begin{array}{lcll}
\mathsf{expand}_a(K) & = & K & \text{if } a \text{ is a primitive type} \\
\mathsf{expand}_1(K) & = & () & \\
\mathsf{expand}_{\Pi x:A_1.\ A_2}(K) & = & & \text{where } M = \mathsf{expand}_{A_1}(x) \\
\quad \lambda x.\ \mathsf{expand}_{A_2}(K\ M) & & & \text{and } x \notin \mathsf{FV}(K) \\
\mathsf{expand}_{\{P\}x:A\{Q\}}(K) & = & & \text{where } M = \mathsf{expand}_A(x) \\
\quad \mathsf{dia}\ (\mathsf{let}\ \mathsf{dia}\ x = K\ \mathsf{in}\ M) & & & \\
\mathsf{expand}_A(N) & = & N & \text{if } N \text{ is an intro term}
\end{array}
$$

Now we can present the main typing rules.

$$
\frac{}{\Delta, x{:}A, \Delta_1 \vdash x \Rightarrow A\,[x]}\ \mathsf{var}
\qquad
\frac{}{\Delta \vdash () \Leftarrow 1\,[()]}\ \mathsf{unit}
$$

$$
\frac{\Delta, x{:}A \vdash M \Leftarrow B\,[M']}{\Delta \vdash \lambda x.\ M \Leftarrow \Pi x{:}A.\ B\,[\lambda x.\ M']}\ \Pi\mathsf{I}
$$

$$
\frac{\Delta \vdash K \Rightarrow \Pi x{:}A.\ B\,[N'] \quad \Delta \vdash M \Leftarrow A\,[M']}{\Delta \vdash K\ M \Rightarrow [M'/x]^a_A(B)\,[\mathsf{apply}_A(N', M')]}\ \Pi\mathsf{E}
$$

$$
\frac{\Delta \vdash K \Rightarrow A\,[N'] \quad A = B}{\Delta \vdash K \Leftarrow B\,[\mathsf{expand}_A(N')]}\ \Rightarrow\!\Leftarrow
$$

$$
\frac{\Delta \vdash A \Leftarrow \mathsf{type}\,[A'] \quad \Delta \vdash M \Leftarrow A'\,[M']}{\Delta \vdash M : A \Rightarrow A'\,[M']}\ \Leftarrow\!\Rightarrow
$$

The introduction forms are associated with the checking judgment, and thus the rule $\Pi\mathsf{I}$ checks that the term $\lambda x.\ M$ has the given function type. The rule also computes the canonical form $\lambda x.\ M'$.

The elimination rule $\Pi\mathsf{E}$ first synthesizes the type $\Pi x{:}A.\ B$ and the canonical form N' of the function part of the application. Then the synthesized type is used in checking the argument part of the application. The result type is synthesized using hereditary substitutions in order to remove the dependency of the type B on the variable x. Notice that the arguments to this hereditary substitution are canonical. Finally, we compute the canonical form of the whole application, with the auxiliary function apply.

In the rule $\Rightarrow\Leftarrow$, we are checking an elim form against a type B. But for elim forms we can already synthesize a type A, so we simply check if A and B are actually equal canonical types. The canonical form synthesized from K in the premise, may not be an intro form (because K itself is elim), so we may need to appropriately expand it.

In the rule $\Leftarrow\Rightarrow$, if M checks against the type A, we we return the canonical form A' as a type synthesized for M.

Computations. Before we state the rules for the computation judgments, we need several additional constructs. We first introduce the auxiliary function $\mathsf{reduce}_A(M, x.\ E)$ which normalizes the term $\mathsf{let\ dia}\ x = M\ \mathsf{in}\ E$:

$$\begin{array}{lll}
\mathsf{reduce}_A(K, x.\ E) & = & \text{if } K \text{ is an elim term} \\
\quad \mathsf{let\ dia}\ x = K\ \mathsf{in}\ E & & \\
\mathsf{reduce}_A(\mathsf{dia}\ F, x.\ E) = E' & \text{where } E' = \langle F/x\rangle_A(E) & \\
\mathsf{reduce}_A(N, x.\ E) & \text{fails otherwise} &
\end{array}$$

Given the propositions P and Q, we write $P; Q$ for $\exists h{:}\mathsf{heap}.\ [h/\mathsf{mem}]P \wedge [h/\mathsf{init}]Q$. This new connective captures how the heap evolves with the computation, as $P; Q$ holds of the current heap mem, if Q is true of mem, and there exist a prior heap h of which P was true.

We can now present the typing rules for computations, which are essentially a formalization of a verification condition generator that works by computing strongest postconditions. We start with the rules that correspond to the monadic types, and then proceed with the rules for the individual effectful commands.

$$\frac{\Delta; P \vdash E \Rightarrow x{:}A.\ R\,[E'] \quad \Delta, x{:}A; \mathsf{init}, \mathsf{mem}; R \Longrightarrow Q}{\Delta; P \vdash E \Leftarrow x{:}A.\ Q\,[E']}\ \mathsf{consq}$$

$$\frac{\Delta \vdash M \Leftarrow A\,[M']}{\Delta; P \vdash M \Rightarrow x{:}A.\ P \wedge \mathsf{Id}_A(\mathsf{expand}_A(x), M')\,[M']}\ \mathsf{comp}$$

$$\frac{\Delta; \mathsf{HId}(\mathsf{init}, \mathsf{mem}) \wedge P \vdash E \Leftarrow x{:}A.\ Q\,[E']}{\Delta \vdash \mathsf{dia}\ E \Leftarrow \{P\}x{:}A\{Q\}\,[\mathsf{dia}\ E']}\ \{\,\}\mathsf{I}$$

$$\frac{\Delta \vdash K \Rightarrow \{R_1\}x{:}A\{R_2\}\,[N'] \quad \Delta; \mathsf{init}, \mathsf{mem}; P \Longrightarrow R_1 \quad \Delta, x{:}A; P; R_2 \vdash E \Rightarrow y{:}B.\ Q\,[E']}{\begin{array}{l}\Delta; P \vdash \mathsf{let\ dia}\ x = K\ \mathsf{in}\ E \\ \qquad \Rightarrow y{:}B.\ (\exists x{:}A.\ Q)\,[\mathsf{reduce}_A(N', x.\ E')]\end{array}}\ \{\,\}\mathsf{E}$$

The consq rule coerces the inference judgment $E \Rightarrow x{:}A.\ R$ into the checking judgment $E \Leftarrow x{:}A.\ Q$, if the assertion logic can establish that R implies Q. In other words, this rule allows weakening of the consequent into an arbitrary postcondition.

The comp rule types the trivial computation that immediately returns the result $x = M$ and performs no changes to the heap. Thus, the generated postcondition equals the precondition extended with the proposition stating the equality between the canonical forms of x and M.

The $\{\,\}\mathsf{I}$ rule internalizes the monadic judgment into the computation type. A suspended computation $\mathsf{dia}\ E$ has the type $\{P\}x{:}A\{Q\}$ if E is a computation with a precondition P and a postcondition Q. Before typechecking E we need to establish that P marks the starting heap of the computation, by equating the heap variable mem from P to the heap variable init from Q.

The $\{\,\}\mathsf{E}$ rule describes how a suspended computation $K \Rightarrow \{R_1\}x{:}A\{R_2\}$ is sequentially composed with another computation E. The two can be composed if the the assertion logic proves that the precondition R_1 for K is implied by the precondition P for the composite command, and if the computation E checks against the postcondition $P; R_2$, which should hold of the heap after the computation encapsulated by K is executed. The normal form of the whole computation is obtained by invoking the auxiliary function reduce. Because the type B which is the result type of the computations E and $\mathsf{let\ dia}\ x = K\ \mathsf{in}\ E$ is an input of the typing judgments, we implicitly assume that B is well-formed in the context Δ, and in particular, B does not depend on the variable x. Thus, the rule does not need need to make any special considerations about x when passing from the premise about the typing of E to the conclusion. No such convention applies to the postcondition Q, so we need to existentially abstract x in the postcondition of the conclusions. A similar remark applies to the rules for the specific effectful constructs that we present below.

The rules for allocation, lookup and mutation first compute the canonical forms of the involved types and terms. Then, in order to compose a command with an arbitrary computation E, we need to check E against a precondition obtained as a postcondition of the command. But first, we define the *strongest postconditions* for each of these commands.

$$\begin{array}{rl}
\mathsf{sp}(x = \mathsf{alloc}_A(M)) = & \mathsf{HId}(\mathsf{mem}, \mathsf{upd}_A(\mathsf{init}, x, M)) \wedge \\
& x \notin \mathsf{dom}(\mathsf{init}) \\
\mathsf{sp}(x = [M]_A) = & \mathsf{HId}(\mathsf{mem}, \mathsf{init}) \wedge \\
& \mathsf{seleq}_A(\mathsf{mem}, M, \mathsf{expand}_A(x)) \\
\mathsf{sp}([M]_A = N) = & \mathsf{HId}(\mathsf{mem}, \mathsf{upd}_A(\mathsf{init}, M, N))
\end{array}$$

The postcondition for a command is a proposition that most precisely captures the relationship between the heap init prior to the execution of the command, and the heap mem obtained after the execution. Dually, the command may be

seen as a witness for its strongest postcondition. In the definition of sp we assume that all the involved expressions are canonical. The strongest postcondition for allocation $x = \mathsf{alloc}_A(M)$ states that the new heap init differs from the old heap mem in that there is a location x pointing to a term M. The location x is fresh, because it does not appear in the domain of init. The strongest postcondition for lookup $x = [M]_A$ states that the new and the old heap are equal, but the variable x equals the value stored in the location M. Because the propositions we consider here are in canonical form, instead of x, we must use the canonical form $\mathsf{expand}_A(x)$ in the definition of $\mathsf{sp}(x = [M]_A)$. The strongest postcondition for mutation $[M]_A = N$ simply states that the new heap extends the old heap with an assignment where M points to N.

Now we can state the typing rules.

$$\frac{\Delta \vdash A \Leftarrow \mathsf{type}\,[A'] \quad \Delta \vdash M \Leftarrow A'\,[M'] \quad \Delta, x{:}\mathsf{nat}; P; \mathsf{sp}(x = \mathsf{alloc}_{A'}(M')) \vdash E \Rightarrow y{:}B.\ Q\,[E']}{\Delta; P \vdash x = \mathsf{alloc}_A(M); E \Rightarrow y{:}B.\ (\exists x{:}\mathsf{nat}.\ Q)\,[x = \mathsf{alloc}_{A'}(M'); E']}$$

$$\frac{\Delta \vdash A \Leftarrow \mathsf{type}\,[A'] \quad \Delta \vdash M \Leftarrow \mathsf{nat}\,[M'] \quad \Delta; \mathsf{init}, \mathsf{mem}; P \Longrightarrow \mathsf{seleq}_{A'}(\mathsf{mem}, M', -) \quad \Delta, x{:}A'; P; \mathsf{sp}(x = [M']_{A'}) \vdash E \Rightarrow y{:}B.\ Q\,[E']}{\Delta; P \vdash x = [M]_A; E \Rightarrow y{:}B.\ (\exists x{:}A'.\ Q)\,[x = [M']_{A'}; E']}$$

$$\frac{\Delta \vdash A \Leftarrow \mathsf{type}\,[A'] \quad \Delta \vdash M \Leftarrow \mathsf{nat}\,[M'] \quad \Delta \vdash N \Leftarrow A'\,[N'] \quad \Delta; \mathsf{init}, \mathsf{mem}; P \Longrightarrow \mathsf{seleq}_{A'}(\mathsf{mem}, M', -) \quad \Delta; P; \mathsf{sp}([M']_{A'} = N') \vdash E \Rightarrow y{:}B.\ Q\,[E']}{\Delta; P \vdash [M]_A = N; E \Rightarrow y{:}B.\ Q\,[[M']_{A'} = N'; E']}$$

In the cases of lookup and mutation, the sequent $\Delta; \mathsf{init}, \mathsf{mem}; P \Longrightarrow \mathsf{seleq}_{A'}(\mathsf{mem}, M', -)$ is used to prove that the location being dereferenced or updated is actually allocated in the current heap, and is initialized with a value of an appropriate type. The sequent is invoked with parametric heap variables init and mem because these may appear in the involved propositions.

The typing rule for $x = \mathsf{if}_A(M, E_1, E_2)$ first checks the two branches E_1 and E_2 against the preconditions stating the two possible outcomes of the boolean expression M. The respective postconditions P_1 and P_2 are generated, and their disjunction is taken as a precondition for the subsequent computation E.

$$\frac{\Delta \vdash A \Leftarrow \mathsf{type}\,[A'] \quad \Delta \vdash M \Leftarrow \mathsf{bool}\,[M'] \quad \Delta; P \wedge \mathsf{Id}_{\mathsf{bool}}(M', \mathsf{true}) \vdash E_1 \Rightarrow x{:}A'.\ P_1\,[E_1'] \quad \Delta; P \wedge \mathsf{Id}_{\mathsf{bool}}(M', \mathsf{false}) \vdash E_2 \Rightarrow x{:}A'.\ P_2\,[E_2'] \quad \Delta, x{:}A'; P_1 \vee P_2 \vdash E \Rightarrow y{:}B.\ Q\,[E']}{\Delta; P \vdash x = \mathsf{if}_A(M, E_1, E_2); E \Rightarrow y{:}B.\ (\exists x{:}A'.\ Q)\,[x = \mathsf{if}_{A'}(M', E_1', E_2'); E']}$$

The recursion construct requires the body of a recursive function $f.\ x.\ E$, and the term M which is supplied as the initial argument to the recursive function. The body of the function may depend on the function itself (variable f) and one argument (variable x). As an annotation, we also need to present the type of f, which is a dependent function type $\Pi x{:}A.\ \{R_1\}y{:}B\{R_2\}$, expressing that f is a function whose range is a computation with precondition R_1 and postcondition R_2.

$$\frac{\Delta \vdash \Pi x{:}A.\{R_1\}y{:}B\{R_2\} \Leftarrow \mathsf{type}\,[\Pi x{:}A'.\{R_1'\}y{:}B'\{R_2'\}] \quad \Delta \vdash M \Leftarrow A'\,[M'] \quad \Delta; \mathsf{init}, \mathsf{mem}; P \Longrightarrow [M'/x]^p_{A'}(R_1') \quad \Delta, f{:}\Pi x{:}A'.\ \{R_1'\}y{:}B'\{R_2'\}, x{:}A'; \mathsf{Hld}(\mathsf{init}, \mathsf{mem}) \wedge R_1' \vdash E \Leftarrow y{:}B'.\ R_2'\,[E'] \quad \Delta, y{:}[M'/x]^p_{A'}(B'); P; [M'/x]^p_{A'}(R_2') \vdash F \Rightarrow z{:}C.\ Q\,[F']}{\Delta; P \vdash y = \mathsf{fix}_{\Pi x:A.\{R_1\}y:B\{R_2\}}(f.x.E, M); F \Rightarrow z{:}C.\ (\exists y{:}[M/x]^p_{A'}(B').Q)\ [y = \mathsf{fix}_{\Pi x:A'.\{R_1'\}y:B'\{R_2'\}}(f.x.E', M'); F']}$$

Before M can be applied to the recursive function, and the obtained computation executed, we need to check that the main precondition P implies R_1. Because after the recursive call we are in a heap of which R_2 holds, the computation following the recursive call is checked with a precondition $P; R_2$. Of course, because the recursive call was started with using M for the argument x, we need to substitute M in R_1, B and R_2 for x everywhere.

Sequents. The sequent calculus of the the assertion logic formalizes a multi-sorted first-order logic with equality, where the sorts include naturals (with the usual Peano axioms), bools, heaps, functions (with extensionality) and computations. We present here only the rules pertaining to heaps, as all the other sorts are rather standard. Currently HTT has no axioms dealing with computations, because in this paper we do not consider propositions over computations (except the equality Id, which is applicable to any type, and is axiomatized parametrically in this type).

Heaps that differ up to the permutation of assignments are considered equal.

$$\frac{}{\Delta; \Psi; \Gamma_1 \Longrightarrow \mathsf{Id}_{\mathsf{nat}}(M_1, M_2), \mathsf{Hld}(\mathsf{upd}_A(\mathsf{upd}_B(H, M_1, N_1), M_2, N_2), \mathsf{upd}_B(\mathsf{upd}_A(H, M_2, N_2), M_1, N_1)), \Gamma_2}$$

If a heap updates the same address twice, only the latter assignment counts.

$$\frac{}{\Delta; \Psi; \Gamma_1 \Longrightarrow \mathsf{Hld}(\mathsf{upd}_A(\mathsf{upd}_B(H, M, N_1), M, N_2), \mathsf{upd}_A(H, M, N_2)), \Gamma_2}$$

Each address in a heap can point to at most one term.

$$\frac{}{\Delta; \Psi; \Gamma_1, \mathsf{Hld}(\mathsf{upd}_A(H_1, M, N_1), \mathsf{upd}_A(H_2, M, N_2)) \Longrightarrow \mathsf{Id}_A(N_1, N_2), \Gamma_2}$$

Empty heaps do not contain any assignments.

$$\Delta; \Psi; \Gamma_1, \mathsf{Hld}(\mathsf{emp}, \mathsf{upd}_A(H, M, N)) \Longrightarrow \Gamma_2$$

If an address points to something, then it is in the heap's domain.

$$\Delta; \Psi; \Gamma_1, \mathsf{Hld}(H_1, \mathsf{upd}_A(H_2, M, -)) \Longrightarrow \mathsf{indom}(H_1, M), \Gamma_2$$

We note here that the sequent calculus proves the usual McCarthy-style axioms about heaps [16], e.g. $\mathsf{seleq}_A(\mathsf{upd}_A(H, M, N), M, N)$, and $\mathsf{seleq}_A(H, M_1, N_1) \wedge \neg\mathsf{Id}(M_1, M_2) \supset \mathsf{seleq}_A(\mathsf{upd}_A(H, M_2, N_2), M_1, N_1)$.

Examples. Consider the function double below, which takes two integer locations and doubles their contents, before returning the sum of the original contents. We annotated the function with propositions (enclosed in /slashes/) that are generated at the various control points during typechecking. We denote by P and Q respectively, the precondition and the postcondition listed below. For simplicity, we use the decimal instead of Peano numerals.

```
double : Πx:nat. Πy:nat.
    {seleq(mem,x,-) ∧ seleq(mem,y,-) ∧ ¬Id(x,y)} r : nat
    {∀v1:nat, v2:nat. seleq(init, x, v1) ∧ seleq(init, y, v2) ⊃
       Id(r, v1+v2) ∧
           Hld(mem, upd(upd(init, x, 2×v1), y, 2×v2))} =
    λx.λy. dia (/P/
          w1 = [x];
          /P1 = P; Hld(mem,init)∧seleq(mem, x, w1)/
          [x] = w1 + w1;
          /P2 = P1; Hld(mem, upd(init, x, 2×w1))/
          w2 = [y];
          /P3 = P2; Hld(mem, init)∧seleq(mem, y, w2)/
          [y] = w2 + w2;
          /P4 = P3; Hld(mem, upd(init, y, 2×w2))/
          w1 + w2)
```

Typechecking requires that the following sequents be proved, which correspond to the verification condition of double. The sequents can easily be proved, after expanding the definition of the operator ";": (1) $P \Longrightarrow \mathsf{seleq}(\mathsf{mem}, x, -)$ so that x can be dereferenced in the first command; (2) $P_1 \Longrightarrow \mathsf{seleq}(\mathsf{mem}, x, -)$ so that x can be updated in the second command; (3) $P_2 \Longrightarrow \mathsf{seleq}(\mathsf{mem}, y, -)$ so that y can be dereferenced in the third command. (4) $P_3 \Longrightarrow \mathsf{seleq}(\mathsf{mem}, y, -)$ so that y can be updated in the fourth command, and (5) $\exists w_1, w_2{:}\mathsf{nat}.\ P_4 \wedge \mathsf{Id}(r, w_1 + w_2) \Longrightarrow Q$ so that Q is a valid postcondition.

As a second example, consider a function that loops through the first n naturals, and computes their sum. We assume the primitive ordering relations ($\leq$, $<$, $>$, $\geq$), and their implementations as boolean functions (<=, <, >,>= respectively). We assume that the relations and the boolean functions are tied through the assertion logic, so that x <= y can be proved equal to true iff $x \leq y$ is provable as a proposition of the assertion logic. We use the customary if M then E else F instead of $\mathsf{if}(M, E, F)$.

```
sumfunc : Πn:nat. {true} r : nat
            {Hld(mem, init) ∧ Id(2×r, n×(n + 1))} =
  λn. dia(y = fix(f. x. /P0 = Hid(init, mem)∧ true/
                    t = if x > n then
                        /P1 = P0; Hld(init, mem) ∧ x > n/
                          0
                        else
                          /P2 = P0; Hld(init, mem) ∧ x ≤ n/
                          let dia s = f (x + 1)
                          in
                            /P3 = P2; [x + 1/x, s/y]^p_nat(Q)/
                            s + x
                          end;
                    /P4 = (P1 ∧ Id(t, 0)) ∨
                          (∃s:nat. P3 ∧ Id(t, s + x))/
                    t, 1);
          /P5 = P0; [1/x]^p_nat(Q)/
          y)
```

Recursion requires a type annotation for the function variable f. In this case, the type is

$$\Pi x{:}\mathsf{nat}.\ \{\mathsf{true}\} y{:}\mathsf{nat}\{\mathsf{Hld}(\mathsf{mem}, \mathsf{init}) \wedge (x \leq n \supset \mathsf{Id}(2 \times y + x \times x, n \times (n+1) + x)) \wedge (x > n \supset \mathsf{Id}(y, 0))\}$$

essentially expressing that the fixpoint construct computes the sum $y = x + (x+1) + \cdots + n$. We denote by Q the postcondition from this type. The sequents generated during typechecking are: (1) $P_2 \Longrightarrow \mathsf{true}$, so that the computation obtained from $f\ (x+1)$ can be executed, (2) $P_4 \wedge \mathsf{Id}(y, t) \Longrightarrow Q$, so that the body of the recursive function satisfies the specified postcondition, (3) $P_5 \wedge \mathsf{Id}(r, y) \Longrightarrow \mathsf{Hld}(\mathsf{mem}, \mathsf{init}) \wedge \mathsf{Id}(2 \times r, n \times (n+1))$, so that the function sumfunc satisfies the specified postcondition.

5 Properties

In this section we present the two most characteristic properties of HTT. For the thorough development, including the substitution principles and all the proofs, we refer the reader to the accompanying technical report [18].

The first property establishes the decidability of the typing judgments of HTT, under the assumption of an oracle that decides the sequents of the assertion logic.

Theorem 2 (Relative decidability of type checking) *If the validity of every assertion logic sequent $\Delta; \Psi; \Gamma_1 \Longrightarrow \Gamma_2$ can be determined, then all the typing judgments of the HTT are decidable.*

The proof relies on the fact that the judgments of HTT are syntax directed, and involve typechecking smaller expressions, or deciding syntactic equality of types, or computing hereditary substitutions, or deciding sequents of the assertion logics. Checking syntactic equality is obviously a terminating algorithm, and as shown in Theorem 1, hereditary substitutions are terminating as well. Thus, if the validity of each assertion logic sequent can be decided, so too can the typing judgments.

Clearly, in the above theorem, an oracle deciding the sequents from the assertion logic may potentially be substituted with a proof that serves as a *checkable* witness of the sequent's validity. In the spirit of Proof-carrying code [20], it is possible to embed such proofs into HTT terms and computations, which we plan to do in the future work.

The second property that we present shows that a computation does not depend on how the heap in which it executes may have been obtained.

Lemma 3 (Preservation of history)
Suppose that $\Delta; P \vdash E \Leftarrow x{:}A.\ Q\,[E']$. *If* $\Delta; \mathsf{init}, \mathsf{mem} \vdash R \Leftarrow \mathsf{prop}\,[R]$, *then* $\Delta; (R; P) \vdash E \Leftarrow x{:}A.\ (R; Q)\,[E']$.

We emphasize here the relationship between history preservation and the frame rule of Separation logic [22, 25, 23]. In Separation logic, if E is a computation satisfying the Hoare triple $\{P\}\ E\ \{Q\}$, we can use the frame rule to derive $\{P * C\}\ E\ \{Q * C\}$. Here C is an arbitrary proposition and $*$ is a propositional connective defined so that $P{*}Q$ holds of a heap if the heap can be split into two *disjoint* parts so that P holds of the first and Q holds of the second part. Thus, in essence, the frame rule states that the parts of the heap that are not touched by E actually remain invariant.

Our preservation of history can be given a similar interpretation. If E does not modify certain locations in the heap, then the history of these locations (and in particular, their present state) is transferred into the postcondition. But notice that preservation of history is a slightly stronger than the frame rule. The frame rule states the invariance of the untouched locations, but does not say anything about locations that may have been touched, but not necessarily modified (e.g., locations may have only been looked up). In contrast, preservation of history can establish the invariance of untouched as well as touched but unchanged locations.

This is not to say that HTT possesses all the facilities that make the reasoning in Separation logic local and modular. In particular, in order to fully employ Preservation of history, the postcondition of E has to express the ending heap mem in terms of the accumulated changes to the beginning heap given by the variable init. A related problem is that HTT currently cannot specify that a certain heap can be split into disjoint parts, because this requires universal quantification over types. These problems are well-known shortcomings of assertion logics based on arrays, and we plan to overcome them in the future work by moving into stronger assertion logics which allow quantification over types and propositions.

6 Operational semantics

In the previous sections, we have defined HTT as a logic, with the associated notions of proof equality, normalization and canonical forms. We now proceed to develop the view of HTT proofs as programs that can be executed. For that purpose, in this section we define the call-by-value left-to-right structured operational semantics, and present the Preservation and Progress theorems, which establish that HTT is sound with respect to evaluation. The proofs (here omitted, but presented in [18]) are relative to the assumed soundness of the HTT assertion logic. The soundness of the assertion logic is not established here and is left for future work.

The operational semantics assumes the following syntactic categories.

Values	v, l	$::= (\,) \mid \lambda x.\ M \mid \mathsf{dia}\ E \mid \mathsf{true} \mid \mathsf{false} \mid \mathsf{z} \mid \mathsf{s}\ v$
Heap values	χ	$::= \cdot \mid \chi, l \mapsto_A v$
Continuations	κ	$::= \cdot \mid x{:}A.\ E; \kappa$
Control expressions	ρ	$::= \kappa \triangleright E$
Abstract machines	α	$::= \chi, \kappa \triangleright E$

The definition of values is rather standard. We use v to range over values, and l to range over numbers when they are used as pointers into the heap. Heap values are functions assigning naturals to values, where each assignment is indexed by a type, and two heap values are considered equal up to the reordering of their assignments. We will frequently need to convert heap values into heap canonical forms, for reasoning purposes, so we introduce the following conversion function from heap values into heaps from Section 2.

$$[\![\cdot]\!] = \mathsf{emp}$$
$$[\![\chi, l \mapsto_A v]\!] = \mathsf{upd}_A([\![\chi]\!], l, M), \quad \text{where } \cdot \vdash v \Leftarrow A\,[M]$$

We abbreviate $\Delta; \mathsf{mem}; \mathsf{Hld}(\mathsf{mem}, [\![\chi]\!]) \Longrightarrow P$ as $\Delta; \chi \vdash P$. This judgment denotes that the proposition P holds of the heap value χ.

A continuation is a sequence of computations of the form $x{:}A.E$, where each computation in the sequence depends on a bound variable $x{:}A$. The continuation is executed by passing a value to the variable x in the first computation E. If that computation terminates, its return value is passed to the second computation, and so on.

A control expression $\kappa \triangleright E$ pairs up a computation E and a continuation κ, so that E provides the initial value with which the execution of κ can start. Thus, a control expression is in a sense a self-contained computation.

We make the similarity with computations more explicit by providing a typing judgment for control expressions.

The judgment has the form $\Delta; P \vdash \kappa \triangleright E \Leftarrow x{:}A.\ Q$, and its meaning is similar to the one for computations: if executed in a heap of which the proposition P holds, the control expression $\rho = \kappa \triangleright E$ results with a value $x{:}A$ and a heap of which the proposition Q holds. We omit the rules of the judgment here, but it suffices to say that they closely follow the typing rules for computations.

An abstract machine α is a pair of a heap value χ and a control expression $\kappa \triangleright E$. The idea is that $\kappa \triangleright E$ can be evaluated against χ, to eventually produce a result and possibly change the starting heap. The typing judgment for abstract machines has the form $\vdash \chi, \kappa \triangleright E \Leftarrow x{:}A.\ Q$, where A is the result type and Q is a proposition describing the resulting heap. The judgment holds iff $\cdot; P \vdash \kappa \triangleright E \Leftarrow x{:}A.\ Q$, where $P = \mathsf{Hld}(\mathsf{mem}, [\![\chi]\!])$. In other words, we first convert the heap value χ into a canonical proposition P which uniquely defines χ, and then check that the control expression $\kappa \triangleright E$ is well-typed with respect to P, A and Q.

Evaluation. There are three evaluation judgments in HTT; one for elimination terms $K \hookrightarrow_k K'$, one for introduction terms $M \hookrightarrow_m M'$ and one for abstract machines $\chi, \kappa \triangleright E \hookrightarrow_e \chi', \kappa' \triangleright E'$. Each judgment relates an expression with its one-step reduct.

The evaluation rules are mostly straightforward, with several peculiarities arising from the syntactic structure of HTT. For example, HTT is dependently typed, so we must substitute into types and not only into terms. Furthermore, in substitutions, a value must always be paired up with a type, as in $v : A$. Substituting the value alone may result in syntactically ill-formed terms, because of the intro/elim distinction.

Due to space considerations, here we only present several characteristic evaluation rules for abstract machines.

$$\frac{M \hookrightarrow_m M'}{\chi, \kappa \triangleright M \hookrightarrow_e \chi, \kappa \triangleright M'}$$

$$\frac{}{\chi, x{:}A.\ E; \kappa \triangleright v \hookrightarrow_e \chi, \kappa \triangleright [v : A/x]E}$$

$$\frac{}{\chi, \kappa \triangleright \mathsf{let\ dia}\ x = (\mathsf{dia}\ F) : \{P\}x{:}A\{Q\}\ \mathsf{in}\ E \hookrightarrow_e \chi, (x{:}A.\ E; \kappa) \triangleright F}$$

$$\frac{\cdot \vdash A \Leftarrow \mathsf{type}\,[A'] \quad l \notin \mathsf{dom}(\chi)}{\chi, \kappa \triangleright x = \mathsf{alloc}_A(v); E \hookrightarrow_e (\chi, l \mapsto_{A'} v), \kappa \triangleright [l{:}\mathsf{nat}/x]E}$$

$$\frac{\cdot \vdash A \Leftarrow \mathsf{type}\,[A'] \quad l \mapsto_{A'} v \in \chi}{\chi, \kappa \triangleright x = [l]_A; E \hookrightarrow_e \chi, \kappa \triangleright [v : A/x]E}$$

$$\frac{\cdot \vdash A \Leftarrow \mathsf{type}\,[A']}{(\chi_1, l \mapsto_{A'} v', \chi_2), \kappa \triangleright [l]_A = v; E \hookrightarrow_e (\chi_1, l \mapsto_{A'} v, \chi_2), \kappa \triangleright E}$$

The preservation theorem, as usual, states that the evaluation step on a well-typed expression results with well-typed result. In the pure fragment of HTT (i.e., in the case of elim and intro terms), there is an additional claim that evaluation preserves the canonical form of the evaluated term, so that evaluation agrees with normalization.

Theorem 4 (Preservation)

1. *if* $K_0 \hookrightarrow_k K_1$ *and* $\vdash K_0 \Rightarrow A\,[N']$, *then* $\vdash K_1 \Rightarrow A\,[N']$.
2. *if* $N_0 \hookrightarrow_m N_1$ *and* $\vdash N_0 \Leftarrow A\,[N']$, *then* $\vdash N_1 \Leftarrow A\,[N']$.
3. *if* $\alpha_0 \hookrightarrow_e \alpha_1$ *and* $\vdash \alpha_0 \Leftarrow x{:}A.\ Q$, *then* $\vdash \alpha_1 \Leftarrow x{:}A.\ Q$.

When evaluating abstract machines, occasionally we must check that the types given at the input abstract machine are well-formed, so that the output abstract machine is well-formed as well. The type information does not influence which rule applies to any given abstract machine, but may influence whether the evaluation gets stuck. If the evaluation starts with well-typed expressions, then no stuck state can be reached, as the Progress theorem below states. In this sense, Progress theorem establishes the *soundness* of typing with respect to evaluation. But we first need to define the property of the assertion logic which we call *heap soundness*.

Definition 5 (Heap soundness)
The assertion logic of HTT is heap sound iff for every heap value χ, *the existence of a derivation for the sequent* $\cdot; \mathsf{mem}; \mathsf{Hld}(\mathsf{mem}, [\![\chi]\!]) \Longrightarrow \mathsf{seleq}_A(\mathsf{mem}, l, -)$ *implies that* $l \mapsto_A v \in \chi$, *for some value* v.

We do not prove in this paper that our assertion logic is heap sound, and we leave that proof as an important future work, especially since in the future we plan to significantly extend the assertion logic with second-order features and with inductive definitions, as we discuss in Section 7.

In the light of this comment, the Progress theorem should be consider as a *relative soundness*, because it relies on the unproved heap soundness of the assertion logic.

Theorem 6 (Progress)
Suppose that the assertion logic of HTT is heap sound. If $\vdash \chi_0, \kappa_0 \triangleright E_0 \Leftarrow x{:}A.\ Q$, *then either* $E_0 = v$ *and* $\kappa_0 = \cdot$, *or* $\chi_0, \kappa_0 \triangleright E_0 \hookrightarrow_e \chi_1, \kappa_1 \triangleright E_1$, *for some* χ_1, κ_1, E_1.

7 Related and future work

In this section we compare against some of the recent related work on reasoning about languages with effectful higher-order functions.

Honda et al. [11, 3] present a succession of increasingly powerful Hoare logics for reasoning about functional programs with references. Their main feature is a proposition asserting a total correctness of function applications. In HTT, functions are pure and are not subject to Hoare-like

reasoning, which we believe leads to significant conceptual simplifications. Furthermore, HTT is a type theory, and thus is better suited than a Hoare logic to support abstraction and modularity in the specification of data invariants (as discussed in the Introduction). One of the main ingredients in this kinds of specifications is the type constructor Σ for dependent products, which is omitted here for simplicity, but should not be hard to add.

Shao et al. [26] and Xi et al. [29, 30] present dependent type systems for effectful programs, based on the separation between the levels of effectful and pure terms. Only pure terms can appear in specifications, and the connection between the two language levels is established via singleton types. In HTT, all terms (including the encapsulated effectful computations) can be used in the specifications, obviating the need for singleton types.

Mandelbaum et al. [15] present a theory of type refinements for reasoning about behavior of effectful programs. Here the pure terms correspond to the customary ML-style type system, while reasoning about effectful terms employs Hoare-like pre- and postconditions. The assertions are drawn from a substructural logic, and can be parametrized with respect to various effectful commands. The assertion logic, however, is rather restricted in order for typechecking to be decidable, so it is not clear whether it can be extended with equational reasoning about programs with state and aliasing. The language supports a variant of dependent typing via singleton types, somewhat similar in nature to the systems discussed previously [29, 26].

Hamid and Shao [9] and Ni and Shao [21] consider reasoning frameworks for assembly programs, and [21] allows embedded code pointers (and thus higher-order functions). The main technical feature of the later is a predicate expressing the safety of jumping to a given code pointer if a certain precondition is satisfied. All such predicates are later interpreted and proved correct with respect to the whole-program heap. In HTT, we use a type, rather than a proposition to capture the semantics of Hoare triples, and the semantics of Hoare triples is established not by interpretation, but directly as a meta theorem expressing the substitution properties for computations.

Recently, a type theoretic approach to Separation logic has been advocated by Birkedal et al. [4]. This work is similar to ours – at least in spirit – in the sense that it contains dependent types and a type of stateful computations. However, it is also significantly restricted; for example, the integer expressions that can appear in the dependent types are to be strictly second-class in the sense that they cannot appear as function arguments or be returned as function results. No such restrictions exist in HTT.

Abadi and Leino in [1] describe a logic for reasoning about object-oriented program, where, as in HTT, specifications are treated in a similar way as types. One of the described problems with this logic concerns the treatment of local variables; certain specifications cannot be proved because the inference rules for $\mathsf{let\ val}\ x = E\ \mathsf{in}\ F$ do not allow sufficient interaction between the specifications for E and F.

In HTT, the problem with local variables does not appear, as witnessed by our substitution principles, but we would like to mention that HTT currently cannot type computations with local state. The difficulty is that the type of Hoare triples can effectively describe only the state that is reachable from the local variables in Δ, or from the result of the computation, whereas we would like to have computations that manipulate state reachable from anonymous pointers. We plan to address this question in future work by enriching a computation type into $\{P\}\Delta', x{:}A\{Q\}$, where Δ' is a context from which all of the local state of the computation is reachable. This system would be similar to the Contextual modal type theory presented in [19]. A truly local state may then be obtained by introducing parameters that abstract over contexts, as also briefly discussed in [19]. Two computations that share the same context parameter will share the same local state.

A further issue that we plan to address in the future involves the addition of data structures, reasoning about which will also require introduction of inductive predicates into the assertion logic. We will also consider spatial propositional connectives, in the style of Separation logic [22, 25, 23], to support assertions about disjointness of heaps.

References

[1] M. Abadi and K. R. M. Leino. A logic of object-oriented programs. In *Verifi cation: Theory and Practice*, pages 11–41. Springer-Verlag, 2004.

[2] K. R. Apt. Ten years of Hoare's logic: A survey – part I. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 3:431–483, 1981.

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

[4] L. Birkedal, N. Torp-Smith, and H. Yang. Semantics of separation-logic typing and higher-order frame rules. In *Symposium on Logic in Computer Science, LICS'05*, pages 260–269, Chicago, Illinois, June 2005.

[5] R. Cartwright and D. C. Oppen. Unrestricted procedure calls in Hoare's logic. In *Symposium on Principles of Programming Languages, POPL'78*, pages 131–140, 1978.

[6] R. Cartwright and D. C. Oppen. The logic of aliasing. *Acta Informatica*, 15:365–384, 1981.

[7] E. M. Clarke Jr. Programming language constructs for which it is impossible to obtain good Hoare axiom systems. *Journal of the ACM*, 26(1):129–147, January 1979.

[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] N. A. Hamid and Z. Shao. Interfacing Hoare logic and type systems for foundational proof-carrying code. In *Applications of Higher Order Logic Theorem Proving, TPHOL'04*, volume 3223 of *Lecture Notes in Computer Science*, pages 118–135, Park City, Utah, 2004. Springer.

[10] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[11] K. Honda, N. Yoshida, and M. Berger. An observationally complete program logic for imperative higher-order functions. In *Symposium on Logic in Computer Science, LICS'05*, pages 270–279, Chicago, Illinois, June 2005.

[12] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of C. In *USENIX Annual Technical Conference*, pages 275–288, Monterey, Canada, June 2002.

[13] S. L. P. Jones and P. Wadler. Imperative functional programming. In *Symposium on Principles of Programming Languages, POPL'93*, pages 71–84, Charleston, South Carolina, 1993.

[14] K. R. M. Leino, G. Nelson, and J. B. Saxe. *ESC/Java User's Manual*. Compaq Systems Research Center, October 2000. Technical Note 2000-002.

[15] Y. Mandelbaum, D. Walker, and R. Harper. An effective theory of type refinements. In *International Conference on Functional Programming, ICFP'03*, pages 213–226, Uppsala, Sweden, September 2003.

[16] J. L. McCarthy. Towards a mathematical science of computation. In *IFIP Congress*, pages 21–28, 1962.

[17] E. Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.

[18] A. Nanevski and G. Morrisett. Dependent type theory of stateful higher-order functions. Technical Report TR-24-05, Harvard University, December 2005.

[19] A. Nanevski, F. Pfenning, and B. Pientka. Contextual modal type theory. Under consideration for publication in the ACM Transactions on Computation Logic, September 2005.

[20] G. C. Necula. Proof-carrying code. In *Symposium on Principles of Programming Languages, POPL'97*, pages 106–119, Paris, January 1997.

[21] Z. Ni and Z. Shao. Certified assembly programming with embedded code pointers. In *Symposium on Principles of Programming Languages, POPL'06*, pages 320–333, Charleston, South Carolina, January 2006.

[22] P. O'Hearn, J. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *International Workshop on Computer Science Logic, CSL'01*, volume 2142 of *Lecture Notes in Computer Science*, pages 1–19. Springer, 2001.

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

[24] F. Pfenning and R. Davies. A judgmental reconstruction of modal logic. *Mathematical Structures in Computer Science*, 11(4):511–540, 2001.

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

[26] Z. Shao, V. Trifonov, B. Saha, and N. Papaspyrou. A type system for certified binaries. *ACM Transactions on Programming Languages and Systems*, 27(1):1–45, January 2005.

[27] P. Wadler. The marriage of effects and monads. In *International Conference on Functional Programming, ICFP'98*, pages 63–74, Baltimore, Maryland, 1998.

[28] K. Watkins, I. Cervesato, F. Pfenning, and D. Walker. A concurrent logical framework: The propositional fragment. In S. Berardi, M. Coppo, and F. Damiani, editors, *Revised selected papers from the Thirds International Workshop on Types for Proofs and Programs, April 2003, Torino, Italy*, volume 3085 of *Lecture Notices in Computer Science*, pages 355–377. Springer, 2004.

[29] H. Xi. Applied Type System (extended abstract). In *TYPES'03*, pages 394–408. Springer-Verlag LNCS 3085, 2004.

[30] D. Zhu and H. Xi. Safe programming with pointers through stateful views. In *Practical Aspects of Declarative Languages, PADL'05*, volume 3350 of *Lecture Notices in Computer Science*, pages 83–97, Long Beach, California, January 2005. Springer.