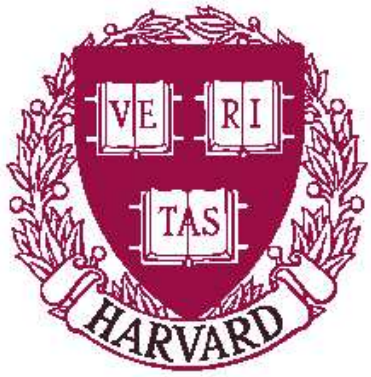
Polymorphism and Separation in Hoare Type Theory

Polymorphism and Separation in Hoare Type Theory

Aleksandar Nanevski, Greg Morrisett and Lars Birkedal

TR-10-06



Computer Science Group
Harvard University
Cambridge, Massachusetts

Polymorphism and Separation in Hoare Type Theory

Aleksandar Nanevski
Harvard University
aleks@eecs.harvard.edu

Greg Morrisett
Harvard University
greg@eecs.harvard.edu

Lars Birkedal
IT University of Copenhagen
birkedal@itu.dk

April 8, 2006

Abstract

In previous work we have proposed a Dependent Hoare Type Theory (HTT) as a framework for development and reasoning about higher-order functional programs with effects of state, aliasing and nontermination. The main feature of HTT is the type of Hoare triples $\{P\}x{:}A\{Q\}$ specifying computations with precondition P and postcondition Q, that return a result of type A.

Here we extend HTT with predicative type polymorphism. Type quantification is possible in both types and assertions, and we can also quantify over Hoare triples. We show that as a consequence it becomes possible to reason about disjointness of heaps in the assertion logic of HTT. We use this expressiveness to interpret the Hoare triples in the "small footprint" manner advocated by Separation Logic, whereby a precondition tightly describes the heap fragment required by the computation. We support stateful commands of allocation, lookup, strong update, deallocation, and pointer arithmetic.

1 Introduction

Modern programming languages such as Java, ML, Haskell, C#, etc. use type systems to statically enforce many desirable aspects of program behavior, e.g. memory safety, and are thus crucial to any kind of application where reliability or security are an issue.

Most type systems, however, only address very simple properties and cannot handle precise specifications about program correctness. Reasoning about such specifications is the task of program logics like Hoare Logic [12]. While significant efforts have been devoted to bridging this gap between type systems and Hoare-like logics – we list ESC/Java [10, 19], Splint [11], and Cyclone [16], among others – it is clear that foundational issues abound, arising from the complexity of higher-order functions, polymorphism and imperative features, which are all indispensable in modern programming languages.

On the other hand, a type system capable of expressing and enforcing precise specifications may potentially offer significant advantages over an ordinary Hoare Logic. For one, such a system could freely combine and abstract over types, specifications and data invariants in a uniform manner. But more importantly, it could use specifications within program syntax to describe the conditions under which any particular computation makes sense. Programs that violate the conditions set by the specifications would not be considered well-formed. This is an instance of the general mechanism by which type systems facilitate scalable and modular program development. In contrast, Hoare Logic does not admit mixing of programs and specifications, and cannot make use of this mechanism.

In previous work [26], we proposed a Hoare Type Theory (HTT) which combines Hoare Logic with dependent types, and facilitates reasoning about higher-order imperative functions. It also serves as a model for the internal program logic and the type system of Cyclone [16]. HTT follows the specifications-as-types principle by providing the type of Hoare triples $\{P\}x{:}A\{Q\}$. This type is ascribed to a stateful computation if the computation, when executed in a heap satisfying the precondition P, returns a heap satisfying the postcondition Q and a result of type A, if it terminates.

While HTT points the way toward a modular program logic, in the sense described previously, the current formulation falls short in several ways. First, the language of HTT does not support polymorphism, which is necessary for Java, ML or Cyclone. Second, the approach to specifying program heaps – which in HTT is based on functional arrays of Cartwright and Oppen [7] and McCarthy [23] – is itself not modular. Preconditions and postconditions in HTT describe the whole heap, rather than just the heap fragment that any particular program requires. Furthermore, the postconditions must explicitly describe how the heap in which the program terminates differs from the heap in which it started. Keeping track of both heaps in the postcondition is cumbersome and may lead to spurious statements about inequality of locations. It is much better to simply assert the properties of the ending heap, and automatically assume that all unspecified disjoint heap portions remain invariant throughout the computation. This is known as the "small footprint" approach to specification, and has been advocated recently by the work on Separation Logic [29, 35, 30, 36].

In this paper, we extend HTT with type polymorphism (including abstraction over Hoare triples) and small footprints. It is interesting that these two additions significantly overlap. At first, we considered simply replacing the functional array approach with the ideas from Separation Logic, but then we realized that in the presence of polymorphism, the functional array approach could already define the separation connectives of spatial conjunction and implication, that are needed to describe heap disjointness [29]. Not only that, but in order to accommodate higher-order functions, we needed additional operators that are not expressible using the separation connectives alone, but are definable in the presence of polymorphism. Thus, functional arrays with polymorphism are utilized in an essential way to obtain the small footprints. An important example that is possible in HTT, but is formally not admitted in Separation Logic, is naming and explicitly manipulating individual fragments of the heap. We contend that it is useful to be able to do so directly. In particular, it alleviates the need for an additional representation of heaps in assertions as was used in the verification of Cheney's garbage collection algorithm in Separation Logic by Birkedal et al. [5]. An additional feature admitted by polymorphism is that HTT can support strong updates, whereby a location can point to values of different types in the course of the execution.

From the type-theoretic standpoint, the Hoare type $\{P\}x{:}A\{Q\}$ of HTT is a monad [24, 25, 17, 41], and it internalizes the process of generating the verification condition for an effectful computation by calculating strongest postconditions. If the verification condition is provable, then the computation matches its specification [28]. Verification conditions are obtained from the computation in a syntax-directed and compositional manner; there is no need for whole-program reasoning. As a consequence, an HTT computation can be seen as (part of) a proof of its specification.[1]

The proof terms of HTT are split into two fragments. The impure, or monadic, fragment consists of the first-order programming constructs amenable to Hoare-like reasoning with pre- and postconditions. This includes commands for allocation, lookup, strong update and deallocation of memory, conditionals and recursion. The pure fragment consists of higher-order functions and constructs for predicative polymorphism. Equational reasoning over the pure fragment admits the usual beta reductions and eta expansions. Equational reasoning over the impure fragment admits the monadic laws [32, 25].

Our formulation of the monads is based on the judgmental reconstruction of Pfenning and Davies [32]. The equational reasoning is organized around hereditary substitutions over canonical forms as developed by Watkins et al. [42], which we here extend with predicative polymorphism. Using hereditary substitutions disentangles the mutual dependence between equational reasoning and typechecking, and thus avoids the major source of complexity in dependent type theories. In HTT we also allow canonical and non-canonical forms to interact through the type system, which is a non-trivial extension required by our application.

The rest of the paper is organized as follows. In Section 2, we present the syntax and overview of HTT. In Section 3, we formulate the notions of canonical forms and hereditary substitutions, and prove the main commutation properties. Our contribution here is the treatment of predicative polymorphism. In Section 4 we define the type system of HTT. The main distinctions from the old HTT proposal concerns the formulation of the monadic judgment whose constructs now follow the small footprint approach. It is interesting that the elimination rule of the monadic type may be seen as a formulation of a higher-order frame rule from Separation Logic [30]. In Section 5, we prove the main meta-theoretic properties of HTT,

[1] The remaining part must, of course, certify the verification condition.

including the substitution principles, as well as the usual admissible rules from Hoare Logic like strengthening the precedent, weakening the consequent, and, specific to the small footprint approach – the Frame rule. One of the main properties that appear in our approach based on strongest postconditions, but does not appear in other treatments of Hoare Logics is the property we call Preservation of History. This property shows that a semantics of a program only depends on the heap in which the program executes, but does not depend on how that heap was computed. Preservation of History allows us to freely substitute and combine effectful computations. In Section 6, we formulate the operational semantics of HTT, thus showing that the proof terms of HTT can be given a constructive meaning and viewed as programs. We establish that the type system is sound with respect to evaluation by proving the appropriate progress and preservation theorems. These proofs are relative to the soundness of the assertion logic of HTT, which we prove subsequently using denotational methods in Section 7.

2 Syntax and overview

In this section we describe the syntax of HTT, the definition of HTT heaps, as well as the substitutions and reductions used in reasoning about equality. There is a significant overlap between this system, and our previous proposal for HTT [26], but our presentation here is self-contained.

The syntax of HTT is summarized in the following table.

$$
\begin{array}{rrcl}
\textit{Types} & A, B, C & ::= & \alpha \mid \mathsf{bool} \mid \mathsf{nat} \mid 1 \mid \forall\alpha.\ A \mid \Pi x{:}A.\ B \mid \Psi.X.\{P\}x{:}A\{Q\} \\
\textit{Monotypes} & \tau, \sigma & ::= & \alpha \mid \mathsf{bool} \mid \mathsf{nat} \mid 1 \mid \Pi x{:}\tau.\ \sigma \mid \Psi.X.\{P\}x{:}\tau\{Q\} \\
\textit{Assertions} & P, Q, R, I & ::= & \mathsf{Id}_A(M, N) \mid \mathsf{seleq}_\tau(H, M, N) \mid \\
 & & & \top \mid \bot \mid P \wedge Q \mid P \vee Q \mid P \supset Q \mid \neg P \mid \\
 & & & \forall x{:}A.\ P \mid \forall\alpha.\ P \mid \forall h{:}\mathsf{heap}.\ P \mid \\
 & & & \exists x{:}A.\ P \mid \exists\alpha.\ P \mid \exists h{:}\mathsf{heap}.\ P \\
\textit{Heaps} & H, G & ::= & h \mid \mathsf{empty} \mid \mathsf{upd}_\tau(H, M, N) \\
\textit{Elim terms} & K, L & ::= & x \mid K\ M \mid K\ \tau \mid M : A \\
\textit{Intro terms} & M, N, O & ::= & K \mid (\,) \mid \lambda x.\ M \mid \Lambda\alpha.\ M \mid \mathsf{dia}\ E \mid \mathsf{true} \mid \mathsf{false} \mid \\
 & & & \mathsf{z} \mid \mathsf{s}\ M \mid M + N \mid M \times N \mid \mathsf{eq}(M, N) \\
\textit{Commands} & c & ::= & x = \mathsf{alloc}_\tau(M) \mid x = [M]_\tau \mid [M]_\tau = N \mid \\
 & & & \mathsf{dealloc}(M) \mid x = \mathsf{if}_A(M, E_1, E_2) \mid \\
 & & & x = \mathsf{fix}_A(M, f.y.F) \\
\textit{Computations} & E, F & ::= & M \mid \mathsf{let\ dia}\ x = K\ \mathsf{in}\ E \mid c; E \\
\textit{Variable context} & \Delta, \Psi & ::= & \cdot \mid \Delta, x{:}A \mid \Delta, \alpha \\
\textit{Heap context} & X & ::= & \cdot \mid X, h \\
\textit{Assertion context} & \Gamma & ::= & \cdot \mid \Gamma, P
\end{array}
$$

Types. The types of HTT include the primitive types of Booleans and natural numbers, unit type 1, dependent functions $\Pi x{:}A.\ B$, Hoare triples $\Psi.X.\{P\}x{:}A\{Q\}$, and polymorphic types $\forall\alpha.\ A$.

The type $\Psi.X.\{P\}x{:}A\{Q\}$ specifies an effectful computation with a precondition P and a postcondition Q, returning a result of type A: if the heap at the beginning of the computation satisfies the assertion P, then the computation executes without getting stuck, and if it terminates, then the ending heap will satisfy the assertion Q. The variable x names the return value of the computation, and Q may depend on x. The contexts Ψ and X list the variables and heap variables, respectively, which may appear in both P and Q, thus helping relate the properties of the beginning and the ending heap. In the literature on Hoare Logic, these are known under the name of *logic variables*. As customary, logic variables can only appear in the assertions, but not in the programs and types. In particular, the type A cannot contain any variables from Ψ and X.

The type $\forall\alpha.\ A$ polymorphically quantifies over the *monotype* variable α. A monotype is a type whose dependency-free version does not contain polymorphic quantification. For example, the type $\Psi.X.\{P\}x{:}A\{Q\}$ is a monotype as long as A is a monotype, but Ψ, P and Q are allowed to contain polymorphism. Allowing

polymorphism in this way does not change the predicative nature of HTT. As we discuss in Section 7, the logic variables and the assertions do not have any influence over the computational behavior or equational properties of effectful computations: if two terms of some Hoare type are semantically equal, then they are equal under any other Hoare type that they may belong to.

As customary, the dependent function type $\Pi x{:}A.\ B$ is abbreviated as $A \to B$ when B does not depend on x. The Hoare type $\{\top\}x{:}A\{\top\}$ with no logic variables, is abbreviated as $\Diamond A$.

Heaps and locations. In this paper, we assume that memory locations are natural numbers. A heap is a finite function, mapping a location N to a pair (τ, M) where τ is the monotype of M. In this case we say that N *points to* M, or that M is the *content* of location N, or that the heap *assigns* M to the location N.

We define syntax for denoting particular heaps, which we use in HTT assertions. For example, empty denotes the empty heap, and $\mathsf{upd}_\tau(H, M, N)$ denotes the heap obtained from H by updating the location M so that it points to N of type τ, while retaining all the other assignments of H. We also allow heap variables.

As indicated previously, heaps can point only to monotyped terms. The restriction to monotypes is realistic, as it is also found in some of the most popular functional languages today (e.g., Standard ML).

Assertions. Assertions comprise the usual connectives of classical multi-sorted first-order logic. The sorts include all the types of HTT, but also the domain of heaps, which we described above. In addition, we allow polymorphic quantification $\forall\alpha.\ P$ and $\exists\alpha.\ P$ over monotypes. The assertion $\mathsf{Id}_A(M, N)$ denotes propositional equality between the terms M and N at type A. The proposition $\mathsf{seleq}_\tau(H, M, N)$ states that the heap H at address M contains a term N of monotype τ.

We will frequently write $\forall\Psi.\ A$ and $\exists\Psi.\ A$ for an iterated universal (resp. existential) abstraction over the term and type variables of the context Ψ. Similarly, we write $\forall X.\ A$ and $\exists X.\ A$ for iterated quantification over heap variables of the context X.

We now introduce some derived assertions that will frequently feature in our Hoare types.

$$
\begin{array}{rcl}
P \subset\!\supset Q & = & P \supset Q \wedge Q \supset P \\
\mathsf{HId}(H_1, H_2) & = & \forall\alpha.\forall x{:}\mathsf{nat}, v{:}\alpha.\ \mathsf{seleq}_\alpha(H_1, x, v) \subset\!\supset \mathsf{seleq}_\alpha(H_2, x, v) \\
M \in H & = & \exists\alpha.\exists v{:}\alpha.\ \mathsf{seleq}_\alpha(H, M, v) \\
M \notin H & = & \neg(M \in H) \\
H_1 \sharp H_2 & = & \forall x{:}\mathsf{nat}.\ x \notin H_1 \vee x \notin H_2 \\
\mathsf{share}(H_1, H_2, M) & = & \forall\alpha.\forall v{:}\alpha.\ \mathsf{seleq}_\alpha(H_1, M, v) \subset\!\supset \mathsf{seleq}_\alpha(H_2, M, v) \\
\mathsf{splits}(H, H_1, H_2) & = & \forall x{:}\mathsf{nat}.\ (x \notin H_1 \wedge \mathsf{share}(H, H_2, x)) \vee (x \notin H_2 \wedge \mathsf{share}(H, H_1, x))
\end{array}
$$

It should be clear from the above equations that HId is the heap equality, $M \in H$ denotes that the location M is in the domain of H, $H_1 \sharp H_2$ states that H_1 and H_2 are disjoint (i.e., have disjoint domains), share state that heaps H_1 and H_2 agree on the location M, and splits states that H can be split into disjoint heaps H_1 and H_2.

To illustrate these definitions, we list several equations that can be proved using the axioms of the HTT assertion logic, which will be introduced in Section 4.

$$
\begin{array}{l}
\mathsf{splits}(H_1, H_2, \mathsf{emp}) \supset \mathsf{HId}(H_1, H_2) \\
\mathsf{splits}(H_1, \mathsf{emp}, H_2) \supset \mathsf{HId}(H_1, H_2) \\
\mathsf{splits}(H, H_1, G) \wedge \mathsf{splits}(H, H_2, G) \supset \mathsf{HId}(H_1, H_2) \\
\mathsf{splits}(H, H, \mathsf{emp}) \\
\mathsf{splits}(H, H_1, H_2) \supset \mathsf{splits}(H, H_2, H_1) \\
\mathsf{share}(H_1, H_2, M) \supset \mathsf{share}(H_2, H_1, M) \\
\mathsf{share}(H_1, H_2, M) \wedge \mathsf{share}(H_2, H_3, M) \supset \mathsf{share}(H_1, H_3, M) \\
x \notin H_1 \wedge x \notin H_2 \supset \mathsf{share}(H_1, H_2, x) \\
x \notin H_1 \wedge \mathsf{share}(H_1, H_2, x) \supset x \notin H_2
\end{array}
$$

We next define the assertions familiar from Separation Logic [29, 35, 30, 36]. All of these are relative to the free variable mem, which denotes the current heap fragment of reference. In the definition of HTT type system in Section 4, we will arrange that the pre- and postconditions in Hoare types all well-formed with respect to this variable.

$$\begin{aligned}
\mathsf{emp} &= \mathsf{Hld}(\mathsf{mem}, \mathsf{empty}) \\
M \mapsto_\tau N &= \mathsf{Hld}(\mathsf{mem}, \mathsf{upd}_\tau(\mathsf{empty}, M, N)) \\
M \mapsto_\tau - &= \exists v{:}\tau.\ M \mapsto_\tau v \\
M \mapsto - &= \exists \alpha.\ M \mapsto_\alpha - \\
M \hookrightarrow_\tau N &= \mathsf{seleq}_\tau(\mathsf{mem}, M, N) \\
M \hookrightarrow_\tau - &= \exists v{:}\tau.\ M \hookrightarrow_\tau v \\
M \hookrightarrow - &= \exists \alpha.\ M \hookrightarrow_\alpha - \\
P * Q &= \exists h_1, h_2{:}\mathsf{heap}.\ \mathsf{splits}(\mathsf{mem}, h_1, h_2) \wedge [h_1/\mathsf{mem}]P \wedge [h_2/\mathsf{mem}]Q \\
P \mathrel{-\!\!*} Q &= \forall h_1, h_2{:}\mathsf{heap}.\ \mathsf{splits}(h_2, h_1, \mathsf{mem}) \supset [h_1/\mathsf{mem}]P \supset [h_2/\mathsf{mem}]Q \\
\mathsf{this}(H) &= \mathsf{Hld}(\mathsf{mem}, H)
\end{aligned}$$

As expected, emp denotes that the current heap mem is empty; $M \mapsto_\tau N$ denotes that the current heap consists of a single location M which points to the term $N{:}\tau$; $M \hookrightarrow_\tau N$ states that the current heap contains at least the location M pointing to $N{:}\tau$. $P * Q$ holds of the current heap if the heap can be split into two disjoint fragments so that P holds of one, and Q hold of the other fragment. $P \mathrel{-\!\!*} Q$ holds of the current heap if any extension by a heap of which P holds, produces a heap of which Q holds. $\mathsf{this}(H)$ is true of the current heap, iff it equals H.

Terms. The programming language of HTT consists of terms and computations (the other categories, like assertions and heaps, describe the assertion logic of HTT). Computations include the effectful fragment of HTT, and are described below. As a rule of thumb, computations contain all kinds of first-order constructs which can be treated in a Hoare Logic-like manner, by pre- and postconditions. All the other constructors are in the domain of terms: this includes higher-order functions, polymorphic abstraction and instantiation, the Booleans (true and false) and the natural numbers in Peano arithmetic style (zero z, successor function s, equality eq, and the $+$ and $\times$ operations).

The domain of terms is split into two categories: introduction (intro) terms and elimination (elim) terms, according to their standard logical classification. For example, $\lambda x.\ M$ is an intro term for the dependent function type, and $K\ M$ is the appropriate elim term. Similarly, $\Lambda\alpha.\ M$ and $K\ \tau$ are the intro and elim terms for polymorphic quantification. The intro term for the unit type is $(\,)$, and, as customary, there is no corresponding elimination term. The intro term for computations is $\mathsf{dia}\ E$. It encapsulates and suspends the computation E. The corresponding elim form activates a suspended computation. However, this elim form is not a term, but a computation, and is described below.

The separation into intro and elim terms facilitates bidirectional typechecking [33], whereby most of the type information can be omitted from the terms, as it can be inferred automatically. In the occasions when the type information must be supplied explicitly, the elim term $M : A$ can be used. This kind of formulation also facilitates the equational reasoning, and simplifies the definition of normal forms in Section 3.

Computations. Computations form the effectful fragment of HTT, and are loosely similar to programs in a generic imperative first-order language. There are several important distinctions, however. First, variables in HTT are statically scoped and immutable, as customary in modern functional programming. Second, computations can freely invoke any kind of terms, including higher-order functions and other suspended computations. Third, computations returns a result, unlike in imperative languages where programs are usually evaluated for their effect.

Each computation is a semicolon-separated list of commands. We describe the commands below.

1. $x = \mathsf{alloc}_\tau(M)$ allocates space in the heap and initializes it with $M{:}\tau$. The address of the allocated space is returned in the variable x.

2. $x = [M]_\tau$ looks up the term that the current heap assigns to the location M. The terms is stored in the variable x. To perform this operation, it must be proved that the location M indeed points to a term of type τ.

3. $[M]_\tau = N$ updates the heap so that the location M points to the term $N{:}\tau$. To perform this operation, it must be proved that the location M is allocated, with a term of *arbitrary type*. Since the old type is arbitrary, the operation implements *strong update*.

4. $\mathsf{dealloc}(M)$ frees the heap space pointed to by M. To perform this operation, it must be proved that M is allocated, with a term of arbitrary type.

5. $x = \mathsf{if}_A(M, E_1, E_2)$ is a conditional which executes the computation E_1 or E_2 depending on the value of the Boolean term M. The return type of both E_1 and E_2 is A, and the return value is stored in x.

6. $x = \mathsf{fix}_A(M, f.y.E)$ is a recursion construct. A is a type of the form $\Pi z{:}B.\ \Psi.X.\{R_1\}x{:}C\{R_2\}$. The construct first computes the least fixpoint of the equation $f = \lambda x.\ \mathsf{dia}\ E$. The obtained function is immediately applied to the initial value $M{:}B$ and the resulting computation is activated to compute a result (of type $[M/z]C$) which gets bound to x. Here $[M/z]$ is a capture-avoiding substitution of M for z.

7. The computation that simply consists of an intro term M is the trivial computation that just returns M as its result.

8. The computation $\mathsf{let\ dia}\ x = K\ \mathsf{in}\ E$ activates the computation that is encapsulated and suspended by K, bind its result to x and proceeds to evaluate E. This essentially achieves the sequential composition of K and E. The construct is the elimination form for the Hoare types in HTT. Notice that activating a suspended computation can only be carried out by another computation. Thus, once we activate a computation and perform an effect, we cannot leave the effectful fragment anymore. This is a characteristic property of monadic type systems [24, 41], and should not be surprising; as mentioned earlier, each Hoare type in HTT is a monad. In the monadic literature, the $\mathsf{let\ dia}$ construct is often denoted as $\mathsf{let\ val}$ or monadic bind.

Equational reasoning and substitutions. We illustrate here some of the equations that HTT uses to reason about equality of terms and computations. The precise characterization of these equations is given in Section 3. Here, we list only the beta reductions and eta expansions for the various type constructors as they provide a valuable insight into the meaning of HTT programs.

The function type has the standard reductions and expansions (where $[K/x]$ denotes the capture-avoiding substitution, and FV denotes the free variables of its argument).

$$\begin{array}{rcll} (\lambda x.\ M : \Pi x{:}A.\ B)\ N & \Longrightarrow_\beta & [N : A/x]M & \\ M : \Pi x{:}A.\ B & \Longrightarrow_\eta & \lambda y.\ (M : \Pi x{:}A.\ B)\ y & \text{where } y \notin \mathsf{FV}(M : \Pi x{:}A.\ B) \end{array}$$

Notice how the redexes in the above terms are all annotated with types, in accord with the syntactic rules of HTT. We must also decorate the term N with its type A before we substitute it for the variable x, because the substitution itself may create new redexes.

The unit type 1 has no beta reductions, but it has an eta expansion.

$$M : 1 \Longrightarrow_\eta ()$$

The reductions and expansions for polymorphic quantification are also standard (here FTV denotes the set of free monotype variables of its argument).

$$\begin{array}{rcll} (\Lambda\alpha.\ M : \forall\alpha.\ A)\ \tau & \Longrightarrow_\beta & [\tau/\alpha]M & \\ M : \forall\alpha.\ A & \Longrightarrow_\eta & \Lambda\beta.\ (M : \forall\alpha.\ A)\ \beta & \text{where } \beta \notin \mathsf{FTV}(M : \forall\alpha.\ A) \end{array}$$

The equations for the Hoare type should account for the sequential composition of two computations. To that end, we define the 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.

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

With the monadic substitution defined, we can specify the equation for the Hoare types as follows.

$$\begin{array}{rcl}
\mathsf{let\ dia}\ x = \mathsf{dia}\ E : \Psi.X.\{P\}y{:}A\{Q\}\ \mathsf{in}\ F & \Longrightarrow_\beta & \langle E/x : A\rangle F \\
M : \Psi.X.\{P\}x{:}A\{Q\} & \Longrightarrow_\eta & \mathsf{dia}\ (\mathsf{let\ dia}\ y = M : \Psi.X.\{P\}x{:}A\{Q\}\ \mathsf{in}\ y)
\end{array}$$

where $y \notin \mathsf{FV}(M : \Psi.X.\{P\}x{:}A\{Q\})$. The definition of monadic substitution and the corresponding reduction and expansion are taken directly from the work of Pfenning and Davies [32]. Pfenning and Davies show that these equations are equivalent to the standard monadic equational laws [25], with the benefit that the monadic substitution subsumes the associativity laws of [25], thus simplifying the equational theory.

We conclude the section with a definition of yet another capture-avoiding substitution. The operation $[H/h]$ substitutes the heap H for the heap variable h into heaps and assertions. The substitution simply commutes with most of the constructors, except that it leaves terms and types invariant. This is justified as terms and types will not depend on any free heap variables.

$$\begin{array}{lcll}
[H/h](h) & = & H & \\
[H/h](g) & = & g & \text{if } g \neq h \\
[H/h](\mathsf{empty}) & = & \mathsf{empty} & \\
[H/h](\mathsf{upd}_\tau(G, M, N)) & = & \mathsf{upd}_\tau([H/h]G, M, N) & \\
 & & & \\
[H/h](\mathsf{Id}_A(M, N)) & = & \mathsf{Id}_A(M, N) & \\
[H/h](\mathsf{seleq}_\tau(G, M, N)) & = & \mathsf{seleq}_\tau([H/h]G, M, N) &
\end{array}$$

The rest of the assertions follow the same pattern, so we omit them.

Example. In this example we present a polymorphic function swap for swapping the contents of two locations. In a simply-typed language like ML, with a type $A\,\mathsf{ref}$ of references, swap can be given the type $\alpha\,\mathsf{ref}{\times}\alpha\,\mathsf{ref}{\to}1$. This type is an underspecification, of course, as it does not describe how the function works. In HTT, we can be more precise. Furthermore, in HTT we can use strong updates to swap locations pointing to values of different types. One possible definition of swap is presented below.

$$\begin{array}{l}
\mathsf{swap} : \forall\alpha.\forall\beta.\Pi\mathsf{x}{:}\mathsf{nat}.\Pi\mathsf{y}{:}\mathsf{nat}. \\
\quad \mathsf{m}{:}\alpha,\mathsf{n}{:}\beta.\{\mathsf{x} \mapsto_\alpha \mathsf{m}\ *\ \mathsf{y} \mapsto_\beta \mathsf{n}\}\ \mathsf{r} : 1 \\
\qquad\qquad \{\mathsf{x} \mapsto_\beta \mathsf{n}\ *\ \mathsf{y} \mapsto_\alpha \mathsf{m}\} = \\
\quad \Lambda\alpha.\Lambda\beta.\lambda\mathsf{x}.\lambda\mathsf{y}.\ \mathsf{dia}(\mathsf{u} = [\mathsf{x}]_\alpha;\ \mathsf{v} = [\mathsf{y}]_\beta; \\
\qquad\qquad\qquad [\mathsf{y}]_\alpha = \mathsf{u};\ [\mathsf{x}]_\beta = \mathsf{v};\ ())
\end{array}$$

The function takes two monotypes α and β, two locations x and y and produces a computation which looks up both locations, and then writes them back in a reversed order.

The precondition of this computation specifies a heap in which x and y point to values $\mathsf{m}{:}\alpha$ and $\mathsf{n}{:}\beta$, respectively, for some logic variables m and n. The locations must not be aliased, due to the use of $*$ which forces x and y to appear in disjoint portions of the heap. Similar specifications that insists on non-aliasing are possible in several related systems, like Alias Types [38] and ATS with stateful views [44]. However, in HTT, like in Separation Logic, we can include the non-aliasing case as well.

One possible specification which covers both aliasing and non-aliasing has the precondition $(\mathsf{x} \mapsto_\alpha \mathsf{m}\ *\ \mathsf{y} \mapsto_\beta \mathsf{n}) \vee (\mathsf{x} \mapsto_\alpha \mathsf{m} \wedge \mathsf{y} \mapsto_\beta \mathsf{n})$, with the symmetric postcondition. The second disjunct uses $\wedge$ instead of

$*$, and can be true only if the heap contains exactly one location, thus forcing $x = y$. This specification is interesting because it precisely describes the smallest heap needed for swap as the heap containing only x and y.

Another possibility is to admit an arbitrarily large heap in the assertions, but then explicitly state the invariance of the heap fragment not containing x and y. Such a specification will have the precondition (x $\hookrightarrow_\alpha$ m) $\wedge$ (y $\hookrightarrow_\beta$ n) $\wedge$ this(h), and postcondition this(upd$_\beta$(upd$_\alpha$(h, y, m), x, n)), where h is a logic variable denoting an arbitrary heap. Thus heap variables allow us to express some of the invariance that one may express in higher-order separation logic [4].

We next illustrate how swap can be used in a larger program. For example, swapping the same locations twice in a row does not change anything.

```
identity : ∀α.∀β.Πx:nat.Πy:nat.
        h.{x ↪α - ∧ y ↪β - ∧ this(h)} r : 1 {this(h)} =
        λx. λy. dia(let dia u = swap x y
                        dia v = swap x y in ())
```

This function generates a computation for swapping x and y, and then activates it twice with the let dia construct. Here we assumed a specification for swap that admits aliasing.

3 Normal forms, canonical forms and hereditary substitutions

In this section we describe in detail the equational reasoning about HTT terms and computations. The general strategy that we employ for checking equality of expressions is to reduce them to appropriate canonical forms (to be defined soon), and then simply compare the canonical forms for alpha equivalence. This is the *definitional equality* of expressions, which we show is decidable, and will be used in typechecking when the comparisons of terms and types is needed. As described in the previous section, HTT features another form of equality – *propositional equality* – represented by the assertion $\mathsf{Id}_A(M, N)$. Propositional equality allows many more equations (for example, it admits the induction principle of Peano arithmetic), but it is not decidable, and is thus not used in typechecking.

In this section, we formulate the definitional equality of HTT. The development is adopted from the work of Watkins et al. [42], which we here extend with primitive types and polymorphism. The equations of the definitional equality include the beta reductions and eta expansions listed in the previous section, as well as several simple laws about natural numbers that we explain below.

We say that a term is in beta-normal, or simply *normal form* if it does not contain beta redexes. We say that a term is in *canonical form* if it is beta-normal and in eta-long form, i.e. all of its intro subterms are eta expanded. For example, if $f : (\mathsf{nat} \to \mathsf{nat}) \to (\mathsf{nat} \to \mathsf{nat}) \to \mathsf{nat}$ and $g : \mathsf{nat} \to \mathsf{nat}$, then the term $f\ g$ is normal, but not canonical. Its canonical version is $\lambda h.\ f\ (\lambda y.\ g\ y)\ (\lambda x.\ h\ x)$.

The main insight of this section (due to [42]) is that normalization can be defined on ill-typed terms. This is important, as it will allow us to avoid the mutual dependency between equational reasoning and typechecking, which is one of the main sources of complexity in dependent type theories.

At the center of the development is the notion of *hereditary substitution*, which preserves canonicity. For example, in places where an ordinary capture-avoiding substitution creates a redex like $(\lambda x.\ M)\ N$, a hereditary substitution continues by immediately substituting N for x in M. This may produce another redex, that is immediately reduced initiating another hereditary substitution and so on. To ensure termination, hereditary substitutions are parametrized by a metric based on types, which decreases as the substitution proceeds.

We next define the subdomain of HTT terms that will encompass all the canonical forms. First, we exploit the distinction between intro and elim forms from Section 2, and notice that normal terms cannot contain the constructor $M : A$.[2] Thus, the syntax for canonical terms can clearly omit this constructor. Second, we add an intro term $\mathsf{eta}_\alpha\ K$ which remembers to eta expand K once the variable α is instantiated

[2]The reader is encouraged to verify this statement by trying to produce a term with a beta redex without using $M : A$.

with a concrete monotype. Third, we exclude some natural number expressions. For example, $\mathsf{s}\ M + N$ is not canonical, as it can be simplified into $\mathsf{s}\ (M + N)$. The later is simpler, because it makes more of the structure apparent (e.g., just by looking at the syntax of $\mathsf{s}\ (M + N)$, we know that it must be non-zero). Similarly, addition $\mathsf{z} + N$ reduces to N, multiplication $\mathsf{s}\ M \times N$ reduces to $N + (M \times N)$, and multiplication $\mathsf{z} \times N$ reduces to z. These reductions are required in order for the normalization to be *adequate* with respect to the evaluation of natural number terms that we define in Section 6. The syntax of canonical terms is summarized in the table below.

$$
\begin{array}{llll}
\textit{Canonical elim terms} & K, L & ::= & x \mid K\ M \mid K\ \tau \\
\textit{Canonical intro terms} & M, N, O & ::= & K \mid \mathsf{eta}_\alpha\ K \mid (\,) \mid \lambda x.\ M \mid \mathsf{dia}\ E \mid \mathsf{true} \mid \mathsf{false} \mid \mathsf{z} \mid \mathsf{s}\ M \\
 & & & M_1^{-\mathsf{z},\mathsf{s}} + M_2^{-\mathsf{z},\mathsf{s}} \mid M_1^{-\mathsf{z},\mathsf{s}} \times M_2^{-\mathsf{z},\mathsf{s}} \\
 & & & \mathsf{eq}(M_1^{-\mathsf{z},\mathsf{s}}, M_2) \mid \mathsf{eq}(M_1, M_2^{-\mathsf{z},\mathsf{s}})
\end{array}
$$

Here, $M^{-\mathsf{z},\mathsf{s}}$ stands for a canonical term syntactically different from z or $\mathsf{s}\ N$ for some N (because, as commented previously, additions of this form may be reduced). Similar comments apply to multiplication and equality. The canonical forms of the other syntactic categories are build as in Section 2, but instead of ordinary intro and elim terms, they use the canonical intro and elim terms. We use the same letters to denote canonical and general forms; the intended meaning will always be distinguishable from the context.

Next we note that the equational properties of HTT terms do not depend on the full HTT type, but only on its dependency-free version. As noted before, two terms that are equal at some Hoare type are equal at any other Hoare type that they belong to. Thus, when computing the canonical forms, we can clearly ignore the assertions from the Hoare types. With this in mind, given an HTT type A, we define the *shape* A^-, which is the simple type obtained as follows.

$$
\begin{array}{lcl}
(\mathsf{nat})^- & = & \mathsf{nat} \\
(\mathsf{bool})^- & = & \mathsf{bool} \\
(1)^- & = & 1 \\
(\forall\alpha.\ A) & = & \forall\alpha.\ A^- \\
(\Pi x{:}A.\ B)^- & = & A^- \to B^- \\
(\Psi.X.\{P\}x{:}A\{Q\})^- & = & \Diamond(A^-)
\end{array}
$$

We impose an ordering on simple types and write $S_1 \le S_2$ and $S_1 < S_2$ if S_1 can be obtained by substituting type variables by simple monotypes in some subexpression of S_2 (a proper subexpression in the second case). This is clearly a well-defined ordering, as instantiating type variables with simple monotypes always decreases the number of type quantifiers, and thus results in a smaller type.

We proceed to define the operation of eta expansion $\mathsf{expand}_S(N)$. The function takes a simple type S and expands N accordingly.

$$
\begin{array}{lcll}
\mathsf{expand}_a(K) & = & K & \text{if } a \text{ is a primitive type } (\mathsf{nat} \text{ or } \mathsf{bool}) \\
\mathsf{expand}_\alpha(K) & = & \mathsf{eta}_\alpha\ K & \\
\mathsf{expand}_1(K) & = & (\,) & \\
\mathsf{expand}_{\forall\alpha.\ S}(K) & = & \Lambda\alpha.\ \mathsf{expand}_S(K\ \alpha) & \text{where } \alpha \notin \mathsf{FTV}(K) \\
\mathsf{expand}_{S_1 \to S_2}(K) & = & \lambda x.\ \mathsf{expand}_{S_2}(K\ M) & \text{where } M = \mathsf{expand}_{S_1}(x) \\
 & & & \text{and } x \notin \mathsf{FV}(K) \\
\mathsf{expand}_{\Diamond S}(K) & = & & \text{where } M = \mathsf{expand}_S(x) \\
\quad \mathsf{dia}\ (\mathsf{let}\ \mathsf{dia}\ x = K\ \mathsf{in}\ M) & & & \\
\mathsf{expand}_S(N) & = & N & \text{if } N \text{ is an intro term, but not an elim term.}
\end{array}
$$

To reduce clutter, we write $\mathsf{expand}_A(N)$ for $\mathsf{expand}_{A^-}(N)$, but reiterate that type dependencies do not influence eta expansion.

The operation of monotype substitution $[\tau/\alpha](-)$ into types, elim terms and intro terms is defined as

follows.

$$
\begin{array}{lcll}
[\tau/\alpha](\alpha) & = & \tau & \\
[\tau/\alpha](\beta) & = & \beta & \text{if } \alpha \neq \beta \\
[\tau/\alpha](a) & = & a & \text{if } a \text{ is a primitive type or 1} \\
[\tau/\alpha](\Pi x{:}A_1.\ A_2) & = & \Pi x{:}[\tau/\alpha]A_1.\ [\tau/\alpha]A_2 & \text{choosing } x \notin \mathsf{FV}(\tau) \\
[\tau/\alpha](\Psi.X.\{P\}x{:}A\{Q\}) & = & [\tau/\alpha]\Psi.X.\{[\tau/\alpha]P\}x{:}[\tau/\alpha]A\{[\tau/\alpha]Q\} & \text{choosing } \Psi, X, x \text{ disjoint from } \mathsf{FV}(\tau) \\
\\
[\tau/\alpha](\forall\beta.\ A) & = & \forall\beta.\ [\tau/\alpha]A & \text{choosing } \beta \notin \mathsf{FTV}(\tau) \\
\\
[\tau/\alpha](x) & = & x & \\
[\tau/\alpha](K\ N) & = & [\tau/\alpha](K)\ [\tau/\alpha](N) & \\
[\tau/\alpha](K\ \sigma) & = & [\tau/\alpha](K)\ [\tau/\alpha](\sigma) & \\
\\
[\tau/\alpha](\mathsf{eta}_\alpha\ K) & = & \mathsf{expand}_\tau(K) & \\
[\tau/\alpha](\mathsf{eta}_\beta\ K) & = & \mathsf{eta}_\beta\ K & \text{if } \alpha \neq \beta \\
[\tau/\alpha](\lambda x.\ M) & = & \lambda x.\ [\tau/\alpha](M) & \text{choosing } x \notin \mathsf{FV}(\tau) \\
[\tau/\alpha](\Lambda\beta.\ M) & = & \Lambda\beta.\ [\tau/\alpha](M) & \text{choosing } \beta \notin \mathsf{FTV}(\tau) \\
[\tau/\alpha](\mathsf{dia}\ E) & = & \mathsf{dia}\ ([\tau/\alpha]E) &
\end{array}
$$

The substitution commutes with the constructors in other categories. In fact, the only interesting case is substituting into $\mathsf{eta}_\alpha\ K$, where an on-the-fly expansion is carried out. This expansion cannot create new redexes, and thus monotype substitution preserves the canonicity of involved terms.

We next define the following hereditary substitutions. The table below lists their domains and ranges. We use the superscripts $\{k, m, e, a, p, h\}$ to range over the syntactic domains of elim terms, intro terms, computations, types, assertions and heaps, respectively. We will use $*$ to range over all these categories. The index S is a putative shape of the type of M. This index will serve as a termination metric for the substitutions.

$$
\begin{array}{lcll}
[M/x]^k_S(K) & = & K' \text{ or } M' :: S' & \text{substitution into elim term } K \\
[M/x]^m_S(N) & = & N' & \text{substitution into intro term } N \\
[M/x]^e_S(E) & = & E' & \text{substitution into computation } E \\
[M/x]^a_S(A) & = & A' & \text{substitution into type } A \\
[M/x]^p_S(P) & = & P' & \text{substitution into assertion } P \\
[M/x]^h_S(H) & = & H' & \text{substitution into heap } H \\
\langle E/x\rangle_S(F) & = & F' & \text{monadic substitution into computation } F
\end{array}
$$

The substitution into elim terms may return either another elim term, or an intro term. In the later case, we also obtain shape $S' \leq S$ of the putative type of the results.

The substitutions are defined by nested induction, first on the structure of S, and then on the structure of the term being substituted into (in case of the monadic substitution, we use the substituted computation instead). In other words, we either go to a smaller shape (in which case the expressions may become larger), or the shape remains the same, but the expressions decrease.

We note that the hereditary substitutions are partial functions. If the involved expressions are not well-typed, the substitution, while terminating, may fail to return a meaningful result. We will prove in Section 5 that hereditary substitutions are total when restricted to well-typed expressions. As conventional when working with expressions that may fail to be defined, whenever we state an equality $T_1 = T_2$, we imply that T_1 and T_2 are also defined.

The cases for the hereditary substitution into elim terms are as follows.

$$
\begin{array}{lcll}
[M/x]^k_S(x) & = & M :: S & \\
[M/x]^k_S(y) & = & y & \text{if } y \neq x \\
[M/x]^k_S(K\ N) & = & K'\ N' & \text{if } [M/x]^k_S(K) = K' \text{ and } [M/x]^k_S(N) = N' \\
[M/x]^k_S(K\ N) & = & O' :: S_2 & \text{if } [M/x]^k_S(K) = \lambda y.\ M' :: S_1 \to S_2 \\
 & & & \text{where } S_1 \to S_2 \leq S \text{ and } [M/x]^k_S(N) = N' \\
 & & & \text{and } O' = [N'/y]^m_{S_1}(M') \\
[M/x]^k_S(K\ \tau) & = & K'\ \tau' & \text{if } [M/x]^k_S(K) = K' \text{ and } [M/x]^k_S(\tau) = \tau' \\
[M/x]^k_S(K\ \tau) & = & [\tau'/\alpha](M') :: [\tau'^{-}/\alpha](S_2) & \text{if } [M/x]^k_S(K) = \Lambda\alpha.\ M' :: \forall\alpha.\ S_2 \\
 & & & \text{where } \forall\alpha.\ S_2 \leq S \text{ and } [M/x]^a_S(\tau) = \tau' \\
[M/x]^k_S(K') & & \text{fails} & \text{otherwise}
\end{array}
$$

Notice that the substitution into $K\ N$ and $K\ \tau$ may fail to be defined depending on what is returned as a result of substituting into K. For example, a failure will appear if $[M/x]^k_S(K)$ returns an intro term which is not a lambda abstraction, or if the returned shape is not smaller than S.

When the substitution is invoked on well-typed terms, the side conditions about S are always satisfied (this property is true of all hereditary substitutions), so the actual implementation of hereditary substitutions does not need to check for these side conditions. We include the checks here nevertheless, to make it obvious that the hereditary substitution is well-founded, because recursive appeals to substitutions take place on smaller shapes, or on equal shapes and smaller expressions.

The substitution into introduction terms is slightly more complicated, because we require auxiliary functions to deal with normalization of primitive functions like $+$, $\times$ and equality. For example, if one argument to $+$ is of the form $\mathsf{s}\ N$, then s can be moved in front of the $+$ symbol. We use the the following auxiliary functions.

$$
\mathsf{plus}(M, N) = \begin{cases}
N & \text{if } M = \mathsf{z} \\
M & \text{if } N = \mathsf{z} \\
\mathsf{s}\ (\mathsf{plus}(M', N)) & \text{if } M = \mathsf{s}\ M' \\
\mathsf{s}\ (\mathsf{plus}(M, N')) & \text{if } N = \mathsf{s}\ N' \\
M + N & \text{otherwise}
\end{cases}
$$

$$
\mathsf{times}(M, N) = \begin{cases}
\mathsf{z} & \text{if } M = \mathsf{z} \text{ or } N = \mathsf{z} \\
\mathsf{plus}(M', \mathsf{times}(M', N)) & \text{if } M = \mathsf{s}\ M' \\
\mathsf{plus}(\mathsf{times}(M, N'), N') & \text{if } N = \mathsf{s}\ N' \\
M \times N & \text{otherwise}
\end{cases}
$$

$$
\mathsf{equals}(M, N) = \begin{cases}
\mathsf{true} & \text{if } M = N = \mathsf{z} \\
\mathsf{false} & \text{if } M = \mathsf{z} \text{ and } N = \mathsf{s}\ N' \\
 & \text{or } M = \mathsf{s}\ M' \text{ and } N = \mathsf{z} \\
\mathsf{equals}(M', N') & \text{if } M = \mathsf{s}\ M' \text{ and } N' = \mathsf{s}\ N' \\
\mathsf{eq}(M, N) & \text{otherwise}
\end{cases}
$$

We note at this point that all of the above auxiliary functions are total, as whenever their input cannot be reduced, it is simply returned unchanged. The cases of the hereditary substitution into intro terms are now

defined as:

$$
\begin{array}{lcll}
[M/x]^m_S(K) & = & K' & \text{if } [M/x]^k_S(K) = K' \\
[M/x]^m_S(K) & = & N' & \text{if } [M/x]^k_S(K) = N' :: S' \\
[M/x]^m_S(\mathsf{eta}_\alpha\ K) & = & \mathsf{eta}_\alpha\ K' & \text{if } [M/x]^k_S(K) = K' \\
[M/x]^m_S(\mathsf{eta}_\alpha\ K) & = & \mathsf{eta}_\alpha\ K' & \text{if } [M/x]^k_S(K) = \mathsf{eta}_\alpha\ K' :: \alpha \\
 & & & \text{where } \alpha \leq S \\
[M/x]^m_S((\,)) & = & (\,) & \\
[M/x]^m_S(\lambda y.\ N) & = & \lambda y.\ N' & \text{where } [M/x]^m_S(N) = N' \\
 & & & \text{choosing } y \notin \mathsf{FV}(M) \text{ and } y \neq x \\
[M/x]^m_S(\Lambda\alpha.\ N) & = & \Lambda\alpha.\ N' & \text{where } [M/x]^m_S(N) = N' \\
 & & & \text{choosing } \alpha \notin \mathsf{FTV}(M) \text{ and } \alpha \neq x \\
[M/x]^m_S(\mathsf{dia}\ E) & = & \mathsf{dia}\ E' & \text{if } [M/x]^e_S(E) = E' \\
[M/x]^m_S(\mathsf{true}) & = & \mathsf{true} & \\
[M/x]^m_S(\mathsf{false}) & = & \mathsf{false} & \\
[M/x]^m_S(\mathsf{z}) & = & \mathsf{z} & \\
[M/x]^m_S(\mathsf{s}\ N) & = & \mathsf{s}\ N' & \text{where } [M/x]^m_S(N) = N' \\
[M/x]^m_S(N_1 + N_2) & = & \mathsf{plus}(N_1', N_2') & \text{where } [M/x]^m_S(N_1) = N_1' \text{ and } [M/x]^m_S(N_2) = N_2' \\
[M/x]^m_S(N_1 \times N_2) & = & \mathsf{times}(N_1', N_2') & \text{where } [M/x]^m_S(N_1) = N_1' \text{ and } [M/x]^m_S(N_2) = N_2' \\
[M/x]^m_S(\mathsf{eq}(N_1, N_2)) & = & \mathsf{equals}(N_1', N_2') & \text{where } [M/x]^m_S(N_1) = N_1' \text{ and } [M/x]^m_S(N_2) = N_2' \\
[M/x]^m_S(N) & & \text{fails} & \text{otherwise}
\end{array}
$$

All of the cases, except the substitution into $\mathsf{eta}_\alpha\ K$ are compositional. In the later case, the substitution fails if substituting into subterm K does not return an intro term of putative type α. This side condition will always be satisfied when working with well-typed terms, as the only well-typed intro term of type α, which is not at the same time elim term, must be of the form $\mathsf{eta}_\alpha\ K'$.

The definition of hereditary substitution into computations follows.

$$
\begin{array}{lcll}
[M/x]^e_S(N) & = & N' & \text{if } [M/x]^m_S(N) = N' \\
[M/x]^e_S(\mathsf{let\ dia}\ y = K\ \mathsf{in}\ E) & = & \mathsf{let\ dia}\ y = K'\ \mathsf{in}\ E' & \text{if } [M/x]^k_S(K) = K' \text{ and } [M/x]^e_S(E) = E' \\
 & & & \text{choosing } y \notin \mathsf{FV}(M) \text{ and } y \neq x \\
[M/x]^e_S(\mathsf{let\ dia}\ y = K\ \mathsf{in}\ E) & = & F' & \text{if } [M/x]^k_S(K) = \mathsf{dia}\ F :: \Diamond S_1 \\
 & & & \text{and } [M/x]^e_S(E) = E' \text{ and } \Diamond S_1 \leq S \\
 & & & \text{and } F' = \langle F/y\rangle_{S_1}(E') \\
 & & & \text{choosing } y \notin \mathsf{FV}(M) \text{ and } y \neq x \\
[M/x]^e_S(E) & & \text{fails} & \text{otherwise}
\end{array}
$$

This definition is compositional as well, and the only interesting case arises when the substitution into the branch K of $\mathsf{let\ dia}\ y = K\ \mathsf{in}\ E$ returns a dia-suspended computation. That creates a redex which is immediately reduced by invoking a monadic hereditary substitution.

The hereditary monadic substitution $\langle E/x\rangle_S(F)$ differs from the non-hereditary version presented in Section 2 in that it recursively invokes hereditary, rather than ordinary substitutions. It also needs to be indexed with a shape S, which approximates the type of the variable x, and serves as a decreasing metric.

$$
\begin{array}{lcll}
\langle M/x\rangle_S(F) & = & F' & \text{if } F' = [M/x]^e_S(F) \\
\langle \mathsf{let\ dia}\ y = K\ \mathsf{in}\ E/x\rangle_S(F) & = & \mathsf{let\ dia}\ y = K\ \mathsf{in}\ F' & \text{if } F' = \langle E/x\rangle_S(F) \\
\langle c; E/x\rangle_S(F) & = & c; F' & \text{if } F' = \langle E/x\rangle_S(F)
\end{array}
$$

The substitution operations into types, assertions and heaps simply commute with all the constructors, so we do not present them here as they do not introduce any new insights.

We can now prove that hereditary substitutions terminate, independently of whether the terms involved are well typed or not. In the lemmas and theorems in this section we only consider canonical expressions, unless explicitly stated otherwise.

Theorem 1 (Termination of hereditary substitutions)

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

Proof: The first part is by induction on K. The second part is by a nested induction, first on the index shape S (under the ordering $\leq$) , and second on the structure of the argument we apply the substitution to. In each case of the definition, we either decrease S, or failing that, we apply the function to strict subexpressions of the input. ■

To reduce clutter, we will frequently write $[M/x]^*_A(-)$ and $\langle E/x\rangle_A(F)$, instead of $[M/x]^*_{A^-}(-)$ and $\langle E/x\rangle_{A^-}(F)$, correspondingly.

Before proceeding with the meta theoretic properties of hereditary substitutions, we need an auxiliary definition. We say that the *head variable*, or simply *head* of an elimination term is the variable that appears at the beginning of the term. More formally,

$$\begin{array}{rcl} \mathsf{head}(x) & = & x \\ \mathsf{head}(K\ N) & = & \mathsf{head}(K) \\ \mathsf{head}(K\ \tau) & = & \mathsf{head}(K) \end{array}$$

Now, whether a substitution into an elimination term K returns an elimination term, or an introduction term with an additional shape annotation depends solely on the head variable of K.

Lemma 2 (Hereditary substitutions and heads)
If $[M/x]^k_S(K)$ *exists, then*

1. $[M/x]^k_S(K) = K'$ *is elim iff* $\mathsf{head}(K) \neq x$.
2. $[M/x]^k_S(K) = M' :: S'$ *is intro iff* $\mathsf{head}(K) = x$.

We can now establish that hereditary substitutions indeed behave like substitutions. For example, Lemma 3 states that substituting for a variable x in an expression which does not contain x should not change the expression. We also need to consider hereditary substitutions under composition. For ordinary substitutions, we know that $[M/x]([N/y]O) = [[M/x]N/y]([M/x]O)$, if $y \notin \mathsf{FV}(M)$. A similar property holds of hereditary substitutions, as shown by Lemma 6, except that the statement is a bit more complicated because hereditary substitutions are partial operations on possibly ill-typed terms.

Lemma 3 (Trivial hereditary substitutions)
If $x \notin \mathsf{FV}(T)$, *then* $[M/x]^*_A(T) = T$, *where* T *ranges over normal expressions of any syntactic category (i.e., elim terms, intro terms, computations, types, assertions and heaps), and* $* \in \{k, m, e, a, p, h\}$, *correspondingly.*

Proof: By straightforward induction on the structure of T. ■

Lemma 4 (Hereditary substitutions and primitive operations)
Suppose that $[M/x]^m_S(N_1)$ *and* $[M/x]^m_S(N_2)$ *exist. Then the following holds.*

1. $[M/x]^m_S(\mathsf{plus}(N_1, N_2)) = \mathsf{plus}([M/x]^m_S(N_1), [M/x]^m_S(N_2))$.
2. $[M/x]^m_S(\mathsf{times}(N_1, N_2)) = \mathsf{times}([M/x]^m_S(N_1), [M/x]^m_S(N_2))$.
3. $[M/x]^m_S(\mathsf{equals}(N_1, N_2)) = \mathsf{equals}([M/x]^m_S(N_1), [M/x]^m_S(N_2))$.

Proof: By induction on the structure of N_1 and N_2. ■

Lemma 5 (Substitution of expansions)
For every canonical elim term K and canonical type A, $[\mathsf{expand}_A(K)/x]^m_A(\mathsf{expand}_A(x)) = \mathsf{expand}_A(K)$.

Proof: By straightforward induction on the structure of A^-. ■

Lemma 6 (Composition of hereditary substitutions)
Suppose that T ranges over expressions of any syntactic category (i.e., elim terms, intro terms, computations, types, assertions, and heaps), and let $* \in \{k, m, e, a, p, h\}$ respectively. Then the following holds.

1. *If $y \notin \mathsf{FV}(M_0)$, and $[M_0/x]^*_A(T) = T_0$, $[M_1/y]^*_B(T) = T_1$ and $[M_0/x]^m_A(M_1)$ exist, then $[M_0/x]^*_A(T_1) = [[M_0/x]^m_A(M_1)/y]^*_B(T_0)$.*
2. *If $\alpha \notin \mathsf{FTV}(M_0)$, and $[M_0/x]^*_A(T) = T_0$ and $[M_0/x]^a_A(\tau)$ exists, then $[M_0/x]^*_A([\tau/\alpha](T)) = [[M_0/x]^a_A(\tau)/\alpha]([M_0/x]^*_A(T))$.*
3. *If $y \notin \mathsf{FV}(M_0)$ and $[M_0/x]^e_A(F) = F_0$ and $\langle E_1/y\rangle_B(F) = F_1$ and $[M_0/x]^e_A(E_1)$ exists, then $[M_0/x]^e_A(F_1) = \langle [M_0/x]^e_A(E_1)/y\rangle_B(F_0)$.*
4. *If $y \notin \mathsf{FV}(\tau_0)$, and $[M_1/y]^*_B(T) = T_1$, then $[\tau_0/\alpha]([M_1/y]^*_B(T)) = [[\tau_0/\alpha](M_1)/y]^*_B([\tau_0/\alpha](T))$.*
5. *If $\beta \notin \mathsf{FTV}(\tau_0)$, then $[\tau_0/\alpha]([\tau_1/\beta](T)) = [[\tau_0/\alpha](\tau_1)/\beta]([\tau_0/\alpha](T))$.*
6. *If $y \notin \mathsf{FV}(\tau_0)$, and $\langle E_1/y\rangle_B(F) = F_1$, then $[\tau_0/\alpha](\langle E_1/y\rangle_B(F)) = \langle [\tau_0/\alpha](E_1)/y\rangle_B([\tau_0/\alpha](F))$.*
7. *If $x \notin \mathsf{FV}(F)$ and $\langle E_1/y\rangle_B(F) = F_1$ and $\langle E_0/x\rangle_A(E_1)$ exist, then $\langle E_0/x\rangle_A(F_1) = \langle \langle E_0/x\rangle_A(E_1)/y\rangle_B(F)$.*

Proof: By nested induction, first on the shapes A^- and B^-, and then on the structure of the expressions involved (T, T, F, T, T, E_1, and E_0, in the respective cases), using the previous lemmas. ■

4 Type system

The type system of HTT consists of several judgments which are divided into three groups: judgments for type checking, sequent calculus for the assertion logic, and the formation judgments. The typechecking group consists of the following.

$\Delta \vdash K \Rightarrow A\,[N']$	K is an elim term of type A, and N' is its canonical form
$\Delta \vdash M \Leftarrow A\,[M']$	M is an intro term of type A, and M' is its canonical form
$\Delta; P \vdash E \Rightarrow x{:}A.\ Q\,[E']$	E is a computation with precondition P, and *strongest* postcondition Q E returns value x of type A, and E' is its canonical form
$\Delta; P \vdash E \Leftarrow x{:}A.\ Q\,[E']$	E is a computation with precondition P, and postcondition Q E returns value x of type A, and E' is its canonical form

The judgments are explicitly oriented to symbolize whether the type or the assertion are given as input to be checked against, or are synthesized as an output of the judgment. This is a characteristic feature of type systems for bidirectional typechecking [33], which we here employ for typechecking terms and computations alike.

For example, $\Delta \vdash K \Rightarrow A\,[N']$ takes as input the elim form K and the context Δ, and computes the type A and the canonical form N' of K. The output type A will be in canonical form. On the other hand

$\Delta \vdash N \Leftarrow A\,[N']$ takes Δ, an intro form N and the type A as input, and computes the canonical form N' of N. The input type A is required to be canonical.

The judgment $\Delta; P \vdash E \Rightarrow x{:}A.\ Q\,[E']$ takes as input the context Δ, assertion P, computation E and type A, and computes the strongest postcondition Q for the computation E with respect to the precondition P. All the input to the judgment is in canonical form, except E. The output is E' which is the canonical form of E. Symmetrically, $\Delta; P \vdash E \Leftarrow x{:}A.\ Q\,[E']$ takes Δ, P, E, A and Q as inputs, and check if Q is a postcondition (not necessarily the strongest) for E with respect to P. The output of the judgment is the canonical form E' of E.

The sequent calculus for the assertion logic consists of the single judgment with the following form.

$$\Delta; X; \Gamma_1 \Longrightarrow \Gamma_2 \quad \text{if all assertions in } \Gamma_1 \text{ are true, then at least one assertion in } \Gamma_2 \text{ is true}$$

Here Δ is a variable context, X is a context of heap variables, and Γ_1 and Γ_2 are sets of canonical assertions. The sequent calculus implements a classical first-order logic with type polymorphism. The judgment holds if for every instantiation of the variables in Δ and X such that the conjunction of assertions in Γ_1 holds, the disjunction of assertions in Γ_2 holds as well.

The group of formation judgments is as follows.

$\vdash \Delta\ \mathsf{ctx}\,[\Delta']$	Δ is a variable context, and Δ' is its canonical form
$\Delta; X \vdash \Gamma\ \mathsf{pctx}$	Γ is a canonical assertion context
$\Delta; X \vdash P \Leftarrow \mathsf{prop}\,[P']$	P is an assertion, and P' is its canonical form
$\Delta \vdash A \Leftarrow \mathsf{type}\,[A']$	A is a type, and A' is its canonical form
$\Delta \vdash \tau \Leftarrow \mathsf{mono}\,[\tau']$	τ is a monotype, and τ' is its canonical form
$\Delta; X \vdash H \Leftarrow \mathsf{heap}\,[H']$	H is a heap, and H' is its canonical form

In all the judgments of the type system, we always assume that X is a context of heap variables, and that the input variable context Δ is given in canonical form. As conventional, we assume that the contexts contain only distinct variables. We present the rules of the judgments next.

Variable and assertion context formation. Here we define the judgment $\vdash \Delta\ \mathsf{ctx}\,[\Delta']$ for variable context formation, and the judgment $\Delta \vdash \Gamma\ \mathsf{pctx}$ for assertion context formation. In the second judgment, Δ is a variable context which is implicitly assumed well-formed and canonical (i.e. $\vdash \Delta\ \mathsf{ctx}\,[\Delta]$).

$$\frac{}{\vdash \cdot\ \mathsf{ctx}\,[\cdot]} \qquad \frac{\vdash \Delta\ \mathsf{ctx}\,[\Delta'] \quad \Delta' \vdash A \Leftarrow \mathsf{type}\,[A']}{\vdash (\Delta, x{:}A)\ \mathsf{ctx}\,[\Delta', x{:}A']} \qquad \frac{\vdash \Delta\ \mathsf{ctx}\,[\Delta']}{\vdash (\Delta, \alpha)\ \mathsf{ctx}\,[\Delta', \alpha]}$$

$$\frac{}{\Delta; X \vdash \cdot\ \mathsf{pctx}} \qquad \frac{\Delta; X \vdash \Gamma\ \mathsf{pctx} \quad \Delta; X \vdash P \Leftarrow \mathsf{prop}\,[P]}{\Delta; X \vdash (\Gamma, P)\ \mathsf{pctx}}$$

We write $\Delta \vdash \Psi \Leftarrow \mathsf{ctx}\,[\Psi']$ as a shorthand for $\vdash \Delta, \Psi\ \mathsf{ctx}\,[\Delta, \Psi']$.

Type formation. The judgment for type formation is $\Delta \vdash A \Leftarrow \mathsf{type}\,[A']$. It is assumed that $\vdash \Delta\ \mathsf{ctx}\,[\Delta]$. The rules are self-explanatory, except perhaps in the case of Hoare triples, where we need to account for the logic variables abstracted in Ψ. We note that this rule allows Ψ to appear only in the pre- and postconditions, but not in the type of the return result. This reflects the nature of logic variables which can be used only in specifications, but not in the programs (i.e., terms or computations). In addition, the assertions in Hoare types are allowed to depend on an additional heap variable mem, which denotes the current heap of reference

that the assertions are relative to.

$$\frac{}{\Delta, \alpha, \Delta' \vdash \alpha \Leftarrow \mathsf{type}\,[\alpha]}$$

$$\frac{}{\Delta \vdash \mathsf{bool} \Leftarrow \mathsf{type}\,[\mathsf{bool}]} \qquad \frac{}{\Delta \vdash \mathsf{nat} \Leftarrow \mathsf{type}\,[\mathsf{nat}]} \qquad \frac{}{\Delta \vdash 1 \Leftarrow \mathsf{type}\,[1]}$$

$$\frac{\Delta \vdash A \Leftarrow \mathsf{type}\,[A'] \quad \Delta, x{:}A' \vdash B \Leftarrow \mathsf{type}\,[B']}{\Delta \vdash \Pi x{:}A.\ B \Leftarrow \mathsf{type}\,[\Pi x{:}A'.\ B']}$$

$$\frac{\Delta \vdash \Psi \Leftarrow \mathsf{ctx}\,[\Psi'] \quad \Delta, \Psi'; X, \mathsf{mem} \vdash P \Leftarrow \mathsf{prop}\,[P'] \quad \Delta \vdash A \Leftarrow \mathsf{type}\,[A'] \quad \Delta, \Psi', x{:}A'; X, \mathsf{mem} \vdash Q \Leftarrow \mathsf{prop}\,[Q']}{\Delta \vdash \Psi.X.\{P\}x{:}A\{Q\} \Leftarrow \mathsf{type}\,[\Psi'.\{P'\}x{:}A'\{Q'\}]}$$

$$\frac{\Delta, \alpha \vdash A \Leftarrow \mathsf{type}\,[A']}{\Delta \vdash \forall\alpha.\ A \Leftarrow \mathsf{type}\,[\forall\alpha.\ A']}$$

The judgment for monotypes is completely analogous, with the obvious omission of the rule for $\forall\alpha.\ A$.

Assertion formation. The judgment for assertion formation is $\Delta; X \vdash P \Leftarrow \mathsf{prop}\,[P']$, where Δ is a canonical context, and X is a heap context. The assertion P' is the canonical form of P, and is returned as output. The rules describe formation of the primitive assertions Id and $\mathsf{seleq}_\tau(H, M, N)$, the standard propositional connectives of classical logic, and the quantification over term, heap and type variables.

$$\frac{\Delta \vdash A \Leftarrow \mathsf{type}\,[A'] \quad \Delta \vdash M \Leftarrow A'\,[M'] \quad \Delta \vdash N \Leftarrow A'\,[N']}{\Delta; X \vdash \mathsf{Id}_A(M, N) \Leftarrow \mathsf{prop}\,[\mathsf{Id}_{A'}(M', N')]}$$

$$\frac{\Delta \vdash \tau \Leftarrow \mathsf{mono}\,[\tau'] \quad \Delta \vdash H \Leftarrow \mathsf{heap}\,[H'] \quad \Delta \vdash M \Leftarrow \mathsf{nat}\,[M'] \quad \Delta \vdash N \Leftarrow \tau'\,[N']}{\Delta; X \vdash \mathsf{seleq}_\tau(H, M, N) \Leftarrow \mathsf{prop}\,[\mathsf{seleq}_{\tau'}(H', M', N')]}$$

$$\frac{}{\Delta; X \vdash \top \Leftarrow \mathsf{prop}\,[\top]} \qquad \frac{}{\Delta; X \vdash \bot \Leftarrow \mathsf{prop}\,[\bot]}$$

$$\frac{\Delta; X \vdash P \Leftarrow \mathsf{prop}\,[P'] \quad \Delta; X \vdash Q \Leftarrow \mathsf{prop}\,[Q']}{\Delta; X \vdash P \wedge Q \Leftarrow \mathsf{prop}\,[P' \wedge Q']} \qquad \frac{\Delta; X \vdash P \Leftarrow \mathsf{prop}\,[P'] \quad \Delta; X \vdash Q \Leftarrow \mathsf{prop}\,[Q']}{\Delta; X \vdash P \vee Q \Leftarrow \mathsf{prop}\,[P' \vee Q']}$$

$$\frac{\Delta; X \vdash P \Leftarrow \mathsf{prop}\,[P'] \quad \Delta; X \vdash Q \Leftarrow \mathsf{prop}\,[Q']}{\Delta; X \vdash P \supset Q \Leftarrow \mathsf{prop}\,[P' \supset Q']} \qquad \frac{\Delta; X \vdash P \Leftarrow \mathsf{prop}\,[P']}{\Delta; X \vdash \neg P \Leftarrow \mathsf{prop}\,[\neg P']}$$

$$\frac{\Delta \vdash A \Leftarrow \mathsf{type}\,[A'] \quad \Delta, x{:}A'; X \vdash P \Leftarrow \mathsf{prop}\,[P']}{\Delta; X \vdash \forall x{:}A.\ P \Leftarrow \mathsf{prop}\,[\forall x{:}A'.\ P']} \qquad \frac{\Delta \vdash A \Leftarrow \mathsf{type}\,[A'] \quad \Delta, x{:}A'; X \vdash P \Leftarrow \mathsf{prop}\,[P']}{\Delta; X \vdash \exists x{:}A.\ P \Leftarrow \mathsf{prop}\,[\exists x{:}A'.\ P']}$$

$$\frac{\Delta; X, h \vdash P \Leftarrow \mathsf{prop}\,[P']}{\Delta; X \vdash \forall h{:}\mathsf{heap}.\ P \Leftarrow \mathsf{prop}\,[\forall h{:}\mathsf{heap}.\ P']} \qquad \frac{\Delta; X, h \vdash P \Leftarrow \mathsf{prop}\,[P']}{\Delta; X \vdash \exists h{:}\mathsf{heap}.\ P \Leftarrow \mathsf{prop}\,[\exists h{:}\mathsf{heap}.\ P']}$$

$$\frac{\Delta, \alpha; X \vdash P \Leftarrow \mathsf{prop}\,[P']}{\Delta; X \vdash \forall\alpha.\ P \Leftarrow \mathsf{prop}\,[\forall\alpha.\ P']} \qquad \frac{\Delta, \alpha; X \vdash P \Leftarrow \mathsf{prop}\,[P']}{\Delta; X \vdash \exists\alpha.\ P \Leftarrow \mathsf{prop}\,[\exists\alpha.\ P']}$$

Heap formation. The judgment for heap formation is $\Delta; X \vdash H \Leftarrow \mathsf{heap}\,[H']$. It is assumed that $\vdash \Delta\ \mathsf{ctx}$ and X is a context of heap variables. The output of the judgment is H' which is the canonical form of the heap H.

$$\frac{h \in X}{\Delta; X \vdash h \Leftarrow \mathsf{heap}\,[h]} \qquad \frac{}{\Delta; X \vdash \mathsf{empty} \Leftarrow \mathsf{heap}\,[\mathsf{empty}]}$$

$$\frac{\Delta \vdash \tau \Leftarrow \mathsf{mono}\,[\tau'] \quad \Delta; X \vdash H \Leftarrow \mathsf{heap}\,[H'] \quad \Delta \vdash M \Leftarrow \mathsf{nat}\,[M'] \quad \Delta \vdash N \Leftarrow \tau'\,[N']}{\Delta; X \vdash \mathsf{upd}_\tau(H, M, N) \Leftarrow \mathsf{heap}\,[\mathsf{upd}_{\tau'}(H', M', N')]}$$

Sequents. The sequents of the assertion logic are formalized by the judgment $\Delta; X; \Gamma_1 \Longrightarrow \Gamma_2$. Here we assume that Δ is a canonical context, X is a list of heap variables, $\Delta; X \vdash \Gamma_1\ \mathsf{pctx}$ and $\Delta; X \vdash \Gamma_2\ \mathsf{pctx}$, i.e. Γ_1, Γ_2 are well-formed and canonical lists of assertions. In order to simplify the notation somewhat, we implicitly allow that assertions be permuted within Γ_1 and Γ_2.

The presentation of the first-order classical fragment is standard, with left and right sequent rules for each of the connectives and quantifiers. We start with structural fragment, which includes the initial sequents (limited to primitive assertion p, which in our logic includes only Id and seleq), and cut

$$\frac{}{\Delta; X; \Gamma_1, p \Longrightarrow p, \Gamma_2}\ \mathsf{init} \qquad \frac{\Delta; X; \Gamma_1 \Longrightarrow P, \Gamma_2 \quad \Delta; X; \Gamma_1, P \Longrightarrow \Gamma_2}{\Delta; X; \Gamma_1 \Longrightarrow \Gamma_2}\ \mathsf{cut}$$

We have the structural rules of weakening and contraction.

$$\frac{\Delta; X; \Gamma_1 \Longrightarrow \Gamma_2}{\Delta; X; \Gamma_1, P \Longrightarrow \Gamma_2} \qquad \frac{\Delta; X; \Gamma_1 \Longrightarrow \Gamma_2}{\Delta; X; \Gamma_1 \Longrightarrow P, \Gamma_2} \qquad \frac{\Delta; X; \Gamma_1, P, P \Longrightarrow \Gamma_2}{\Delta; X; \Gamma_1, P \Longrightarrow \Gamma_2} \qquad \frac{\Delta; X; \Gamma_1 \Longrightarrow P, P, \Gamma_2}{\Delta; X; \Gamma_1 \Longrightarrow P, \Gamma_2}$$

The propositional connectives do not require much comment either.

$$\frac{}{\Delta; X; \Gamma_1, \bot \Longrightarrow \Gamma_2} \qquad \frac{}{\Delta; X; \Gamma_1 \Longrightarrow \top, \Gamma_2}$$

$$\frac{\Delta; X; \Gamma_1, P, Q \Longrightarrow \Gamma_2}{\Delta; X; \Gamma_1, P \wedge Q \Longrightarrow \Gamma_2} \qquad \frac{\Delta; X; \Gamma_1 \Longrightarrow P, \Gamma_2 \quad \Delta; X; \Gamma_1 \Longrightarrow Q, \Gamma_2}{\Delta; X; \Gamma_1 \Longrightarrow P \wedge Q, \Gamma_2}$$

$$\frac{\Delta; X; \Gamma_1, P \Longrightarrow \Gamma_2 \quad \Delta; X; \Gamma_1, Q \Longrightarrow \Gamma_2}{\Delta; X; \Gamma_1, P \vee Q \Longrightarrow \Gamma_2} \qquad \frac{\Delta; X; \Gamma_1 \Longrightarrow P, Q, \Gamma_2}{\Delta; X; \Gamma_1 \Longrightarrow P \vee Q, \Gamma_2}$$

$$\frac{\Delta; X; \Gamma_1 \Longrightarrow P, \Gamma_2 \quad \Delta; X; \Gamma_1, Q \Longrightarrow \Gamma_2}{\Delta; X; \Gamma_1, P \supset Q \Longrightarrow \Gamma_2} \qquad \frac{\Delta; X; \Gamma_1, P \Longrightarrow Q, \Gamma_2}{\Delta; X; \Gamma_1 \Longrightarrow P \supset Q, \Gamma_2}$$

$$\frac{\Delta; X; \Gamma_1 \Longrightarrow P, \Gamma_2}{\Delta; X; \Gamma_1, \neg P \Longrightarrow \Gamma_2} \qquad \frac{\Delta; X; \Gamma_1, P \Longrightarrow \Gamma_2}{\Delta; X; \Gamma_1 \Longrightarrow \neg P, \Gamma_2}$$

The quantification over term, type and heap variables follows, and is also standard.

$$\frac{\Delta \vdash M \Leftarrow A\,[M] \quad \Delta; X; \Gamma_1, \forall x{:}A.\ P, [M/x]^p_A(P) \Longrightarrow \Gamma_2}{\Delta; X; \Gamma_1, \forall x{:}A.\ P \Longrightarrow \Gamma_2} \qquad \frac{\Delta, x{:}A; \Psi; \Gamma_1 \Longrightarrow P, \Gamma_2}{\Delta; X; \Gamma_1 \Longrightarrow \forall x{:}A.\ P, \Gamma_2}$$

$$\frac{\Delta, x{:}A; \Psi; \Gamma_1, P \Longrightarrow \Gamma_2}{\Delta; \Psi; \Gamma_1, \exists x{:}A.\ P \Longrightarrow \Gamma_2} \qquad \frac{\Delta \vdash M \Leftarrow A\,[M] \quad \Delta; \Psi; \Gamma_1 \Longrightarrow [M/x]^p_A(P), \exists x{:}A.\ P, \Gamma_2}{\Delta; \Psi; \Gamma_1 \Longrightarrow \exists x{:}A.\ P, \Gamma_2}$$

$$\frac{\Delta \vdash \tau \Leftarrow \mathsf{mono}\,[\tau] \quad \Delta; X; \Gamma_1, [\tau/\alpha](P), \forall \alpha.\ P \Longrightarrow \Gamma_2}{\Delta; X; \Gamma_1, \forall \alpha.\ P \Longrightarrow \Gamma_2} \qquad \frac{\Delta, \alpha; X; \Gamma_1 \Longrightarrow P, \Gamma_2}{\Delta; X; \Gamma_1 \Longrightarrow \forall \alpha.\ P, \Gamma_2}$$

$$\frac{\Delta, \alpha; X; \Gamma_1, P \Longrightarrow \Gamma_2}{\Delta; X; \Gamma_1, \exists \alpha.\ P \Longrightarrow \Gamma_2} \qquad \frac{\Delta \vdash \tau \Leftarrow \mathsf{mono}\,[\tau] \quad \Delta; X; \Gamma_1 \Longrightarrow [\tau/\alpha](P), \exists \alpha.\ P, \Gamma_2}{\Delta; X; \Gamma_1 \Longrightarrow \exists \alpha.\ P, \Gamma_2}$$

$$\frac{\Delta; X \vdash H \Leftarrow\ \mathsf{heap}\,[H] \quad \Delta; X; \Gamma_1, \forall h{:}\mathsf{heap}.\ P, [H/h]P \Longrightarrow \Gamma_2}{\Delta; X; \Gamma_1, \forall h{:}\mathsf{heap}.\ P \Longrightarrow \Gamma_2} \qquad \frac{\Delta; X, h; \Gamma_1 \Longrightarrow P, \Gamma_2}{\Delta; X; \Gamma_1 \Longrightarrow \forall h{:}\mathsf{heap}.\ P, \Gamma_2}$$

$$\frac{\Delta; X, h; \Gamma_1 \Longrightarrow P, \Gamma_2}{\Delta; X; \Gamma_1, \exists h{:}\mathsf{heap}.\ P \Longrightarrow \Gamma_2} \qquad \frac{\Delta; X \vdash H \Leftarrow\ \mathsf{heap}\,[H] \quad \Delta; X; \Gamma_1 \Longrightarrow [H/h]P, \exists h{:}\mathsf{heap}.\ P, \Gamma_2}{\Delta; X; \Gamma_1 \Longrightarrow \exists h{:}\mathsf{heap}.\ P, \Gamma_2}$$

Of course, we assume the usual proviso that the variables x, α and h abstracted in the conclusion of the $\exists$-left rules and $\forall$-right rules, do not appear free in other expressions of the rule. This constraint can always be satisfied by alpha-renaming.

Next we need the rules expressing reflexivity and substitutability of equality, which are standard as well. The rules are restricted to primitive assertions q.

$$\frac{}{\Delta; X; \Gamma_1 \Longrightarrow \mathsf{Id}_A(M, M), \Gamma_2} \qquad \frac{\Delta; X; \Gamma_1, \mathsf{Id}_A(M, N) \Longrightarrow [M/x]^p_A(q), [N/x]^p_A(q), \Gamma_2}{\Delta; X; \Gamma_1, \mathsf{Id}_A(M, N) \Longrightarrow [M/x]^p_A(q), \Gamma_2}$$

It is well-known [13], that the equality rules above do not admit extensional equality of functions. The terms M and N must depend only on the variables in Δ and X, while extensional equality of functions require extending the context with an additional variable. We hence require a separate rule for function extensionality, and similarly, a separate rule for equality of type abstractions.

$$\frac{\Delta, x{:}A; X; \Gamma_1 \Longrightarrow \mathsf{Id}_B(M, N), \Gamma_2}{\Delta; X; \Gamma_1 \Longrightarrow \mathsf{Id}_{\Pi x{:}A.\ B}(\lambda x.\ M, \lambda x.\ N), \Gamma_2} \qquad \frac{\Delta, \alpha; X; \Gamma_1 \Longrightarrow \mathsf{Id}_B(M, N), \Gamma_2}{\Delta; X; \Gamma_1 \Longrightarrow \mathsf{Id}_{\forall \alpha.\ B}(\Lambda \alpha.\ M, \Lambda \alpha.\ N), \Gamma_2}$$

In the above rules, it is assumed that the bound variables x and α do not appear free in the involved contexts.

Heaps are axiomatized as partial functions, using the following extra-logical sequents. First, we need to state that an empty heap has no assignments.

$$\frac{}{\Delta; X; \Gamma_1, \mathsf{seleq}_\tau(\mathsf{empty}, M, N) \Longrightarrow \Gamma_2}$$

The McCarthy axioms.

$$\frac{}{\Delta; X; \Gamma_1 \Longrightarrow \mathsf{seleq}_\tau(\mathsf{upd}_\tau(H, M, N), M, N), \Gamma_2}$$

$$\frac{}{\Delta; X; \Gamma_1, \mathsf{seleq}_\tau(\mathsf{upd}_\sigma(H, M_1, N_1), M_2, N_2) \Longrightarrow \mathsf{Id}_{\mathsf{nat}}(M_1, M_2), \mathsf{seleq}_\tau(H, M_2, N_2), \Gamma_2}$$

We also need to state that heaps are functional, i.e. that each location can point to at most one value (and at most one type). In other words, given a location M, and pairs (τ_1, N_1) and (τ_2, N_2) to which M points to, we should be able to conclude that τ_1 and τ_2 are equal monotypes, and N_1 and N_2 are equal terms of monotype $\tau_1 = \tau_2$. In order to state this property, we require a proposition for type equality, and a proposition for equality of terms at *different* types. While these concepts are standard in type theory (e.g. McBride's "John Major" equality [22]), exploiting them fully in HTT requires extensions to higher-order logic. For example, it seems that even if we added a proposition for type equality, it would not be possible in the first-order setting to conclude that the types nat and bool are different. This is analogous to the situation in Martin-Löf type theory extended with the inductive set of natural numbers, where it is not possible to prove $z \neq \mathsf{s}\ \mathsf{z}$ without recourse to higher type universes [39]. Thus, we leave propositional type equality for future work, and instead admit the following, slightly restricted, axiom, which admits functionality of heaps at each given monotype.

$$\frac{}{\Delta; X; \Gamma_1, \mathsf{seleq}_\tau(H, M, N_1), \mathsf{seleq}_\tau(H, M, N_2) \Longrightarrow \mathsf{Id}_\tau(N_1, N_2), \Gamma_2}$$

We believe that this axiom should suffice for all practical purposes. Our monadic judgments will never generate pre- and postconditions in which the same location is considered at the same heap fragment under two different monotypes, unless such propositions are injected into the system by a programmer-written assertions. But even then, unsoundness does not arise, as such assertions will not be satisfiable, and the computation whose specifications contains them will not be executable.

We now proceed with the rules for the primitive types. Rules for integers implement the Peano axioms. We specify that z has no successor, and that s function is injective. Of course, we also require a rule for the induction principle. The primitive operations like $+$ and $\times$ do not require specific rules, as their definition is already implemented by the reduction rules on normal forms in Section 3.

$$\frac{}{\Delta; X; \Gamma_1, \mathsf{Id}_{\mathsf{nat}}(\mathsf{s}\ M, \mathsf{z}) \Longrightarrow \Gamma_2}$$

$$\frac{}{\Delta; X; \Gamma_1, \mathsf{Id}_{\mathsf{nat}}(\mathsf{s}\ M, \mathsf{s}\ N) \Longrightarrow \mathsf{Id}_{\mathsf{nat}}(M, N), \Gamma_2}$$

$$\frac{\Delta \vdash M \Leftarrow \mathsf{nat}\,[M] \quad \Delta; X; \Gamma_1, P \Longrightarrow [\mathsf{s}\ x/x]^p_{\mathsf{nat}}(P), \Gamma_2}{\Delta; X; \Gamma_1, [z/x]^p_{\mathsf{nat}}(P) \Longrightarrow [M/x]^p_{\mathsf{nat}}(P), \Gamma_2}$$

Rules for Booleans state that true and false are not equal – this is similar to the axiom on integers that states how z is not a successor of any integer. After that, we need an extensionality principle for booleans.

$$\frac{}{\Delta; X; \Gamma_1, \mathsf{Id}_{\mathsf{bool}}(\mathsf{true}, \mathsf{false}) \Longrightarrow \Gamma_2}$$

$$\frac{\Delta \vdash M \Leftarrow \mathsf{bool}\,[M]}{\Delta; X; \Gamma_1, [\mathsf{true}/x]^p_{\mathsf{bool}}(P), [\mathsf{false}/x]^p_{\mathsf{bool}}(P) \Longrightarrow [M/x]^p_{\mathsf{bool}}(P), \Gamma_2}$$

Terms. The judgment for type checking of intro terms is $\Delta \vdash K \Rightarrow A\,[N']$, and the judgment for inferring the type of elim terms is $\Delta \vdash K \Rightarrow A\,[N']$. It is assumed that $\vdash \Delta\ \mathsf{ctx}$ and $\Delta \vdash A \Leftarrow \mathsf{type}\,[A]$. In other words, Δ and A are well formed and canonical.

The rules for the primitive operations are self-explanatory, and we present them first. We use the auxiliary functions plus, times and equals defined in Section 3 in order to compute canonical forms of expressions involving primitive operations.

$$\frac{}{\Delta \vdash \mathsf{true} \Leftarrow \mathsf{bool}\,[\mathsf{true}]} \qquad \frac{}{\Delta \vdash \mathsf{false} \Leftarrow \mathsf{bool}\,[\mathsf{false}]}$$

$$\frac{}{\Delta \vdash \mathsf{z} \Leftarrow \mathsf{nat}\,[\mathsf{z}]} \qquad \frac{\Delta \vdash M \Leftarrow \mathsf{nat}\,[M']}{\Delta \vdash \mathsf{s}\ M \Leftarrow \mathsf{nat}\,[\mathsf{s}\ M']}$$

$$\frac{\Delta \vdash M \Leftarrow \mathsf{nat}\,[M'] \quad \Delta \vdash N \Leftarrow \mathsf{nat}\,[N']}{\Delta \vdash M + N \Leftarrow \mathsf{nat}\,[\mathsf{plus}(M', N')]} \qquad \frac{\Delta \vdash M \Leftarrow \mathsf{nat}\,[M'] \quad \Delta \vdash N \Leftarrow \mathsf{nat}\,[N']}{\Delta \vdash M \times N \Leftarrow \mathsf{nat}\,[\mathsf{times}(M', N')]}$$

$$\frac{\Delta \vdash M \Leftarrow \mathsf{nat}\,[M'] \quad \Delta \vdash N \Leftarrow \mathsf{nat}\,[N']}{\Delta \vdash \mathsf{eq}(M, N) \Leftarrow \mathsf{bool}\,[\mathsf{equals}(M', N')]}$$

Before we can state the rules for the composite types, we need two auxiliary functions $\mathsf{apply}_A(M, N)$, and $\mathsf{spec}(M, \tau)$. In $\mathsf{apply}_A(M, N)$, A is a canonical type, the arguments M and N are canonical intro terms. The function normalizes the application $M\ N$. If M is a lambda abstraction, the redex $M\ N$ is immediately normalized by substituting N hereditarily in the body of the lambda expression. If M is an elim term, there is no redex, and the application is returned unchanged. In $\mathsf{spec}(M, \tau)$, τ is a canonical monotype, and M is a canonical intro term. If M is a type abstraction, the function *specializes* M with the monotype τ. Otherwise, it just returns the term $M\tau$. In other cases, apply and spec are not defined, but such cases cannot arise during typechecking, where these functions are only applied to well-typed arguments.

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

$\mathsf{spec}(K, \tau)$	$=$	$K\ \tau$	if K is an elim term
$\mathsf{spec}(\Lambda\alpha.\ M, \tau)$	$=$	$[\tau/\alpha](M)$	
$\mathsf{spec}(N, \tau)$		fails	otherwise

Now we can present the rest of the typing rules for terms.

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

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

$$\frac{\Delta, \alpha \vdash M \Leftarrow A\,[M']}{\Delta \vdash \Lambda\alpha.\ M \Leftarrow \forall\alpha.\ A\,[\Lambda\alpha.\ M']}\ \forall\mathsf{I} \qquad \frac{\Delta \vdash K \Rightarrow \forall\alpha.\ B\,[N'] \quad \Delta \vdash \tau \Leftarrow \mathsf{mono}\,[\tau']}{\Delta \vdash K\ \tau \Rightarrow [\tau'/\alpha](B)\,[\mathsf{spec}(N', \tau')]}\ \forall\mathsf{E}$$

$$\frac{\Delta \vdash K \Rightarrow A\,[N'] \quad A = B}{\Delta \vdash K \Leftarrow B\,[\mathsf{expand}_B(N')]}\ \Rightarrow\Leftarrow \qquad \frac{\Delta \vdash A \Leftarrow \mathsf{type}\,[A'] \quad \Delta \vdash M \Leftarrow A'\,[M']}{\Delta \vdash M : A \Rightarrow A'\,[M']}\ \Leftarrow\Rightarrow$$

$$\frac{\Delta \vdash K \Rightarrow \alpha\,[K]}{\Delta \vdash \mathsf{eta}_\alpha\ K \Leftarrow \alpha\,[\mathsf{eta}_\alpha\ K]}\ \mathsf{eta}$$

The majority of rules are unchanged from our previous formulation of HTT [26]. For example, in $\Pi\mathsf{I}$ we check that term $\lambda x.\ M$ has the given function type, and if so, return the canonical form $\lambda x.\ M'$. In $\Pi\mathsf{E}$ we first synthesize 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. Finally, we compute the canonical form of the whole application, using the auxiliary function apply. Similar description applies to the rules for polymorphic quantification.

In the rule $\Leftarrow\!\Rightarrow$, we need to determine if the term M checks against the type A, of course if A is a well-formed type to start with. If M and A match, we return the canonical form A' as a type synthesized for M.

In the rule $\Rightarrow\!\Leftarrow$, we are checking an elim term K against a type B. But K can already synthesize its type A, so we simply need to check that A and B are actually equal canonical types. The canonical form synthesized from K in the premise, may not be an intro form (because it is generated by a judgment for elim forms), so we may need to eta expand it. Notice that if A is a type variable α, this expansion cannot be carried out until α is actually instantiated with a concrete monotype. We must record this fact by returning $\mathsf{eta}_\alpha\ N'$ as the expanded variant of N'.

As a consequence, we need a rule to typecheck the constructor eta. Notice that this rule is restricted to canonical terms K only (as apparent in the premise), because eta_α is a constructor of canonical terms, but not of general terms.

Computations. The computation fragment of HTT formalizes the reasoning by strongest postconditions in a small footprint approach. We have two judgments: $\Delta; P \vdash E \Rightarrow x{:}A.\ Q\,[E']$ and $\Delta; P \vdash E \Leftarrow x{:}A.\ Q\,[E']$.

The first judgment generates the strongest postcondition Q of E with respect to the precondition P, and thus formalizes the verification condition generator [28]. The postcondition may depend on the variable $x{:}A$ that binds the result returned by E. We emphasize that the type A is not synthesized, but is supplied as an input, and is assumed canonical. The canonical version of E is returns as an output.

The second judgment checks if Q (which is this time supplied as in input), is a postcondition for E with respect to P. Essentially, this is ensured by verifying that the strongest postcondition of E implies P.

It is important to mention that, syntactically, the assertions P and Q may depend on two heap variables init and mem. Formally, $\Delta; \mathsf{init}, \mathsf{mem} \vdash P \Leftarrow \mathsf{prop}\,[P]$ and $\Delta, x{:}A; \mathsf{init}, \mathsf{mem} \vdash Q \Leftarrow \mathsf{prop}\,[Q]$. The variable init stands for the unknown heap in which the computation is supposed to start the execution. The variable mem stands for the heap that is currently under consideration.

Semantically, the role of the assertion Q is to establish the relationship between the initial and the ending heap of the computation E. It is this assertion that logically captures the semantics of E by describing all of the heaps through which the execution of E passes. On the other hand, P as a precondition serves to guarantee that executing E will never get stuck, but it does not say anything about the results of the computation.

Before we can describe how the computation judgments express the small footprint specifications, we need several additional constructs. We first introduce the auxiliary function $\mathsf{reduce}_A(M, x.\ E)$ which normalizes the term $\mathsf{let\ dia}\ x = M\ \mathsf{in}\ E$. If M is a monadic encapsulation $\mathsf{dia}\ F$, we have a redex which is immediately reduced by composing F and E via a monadic hereditary substitution. If M is an elim term, there is no redex and the term is returned unchanged. Other possibilities cause the function to fail, but they do not arise on well-typed arguments.

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

We also require new assertion connectives. First *sequential composition* of assertions:

$$P \circ Q \quad = \quad \exists h{:}\mathsf{heap}.\ [h/\mathsf{mem}]P \wedge [h/\mathsf{init}]Q$$

This connective defines a version of temporal sequencing of heaps. The informal reading of $P \circ Q$ states that Q holds of the current heap, which is itself obtained from another past heap of which P holds. This connective will be used extensively in the strongest postconditions to record the sequence of heaps through which a computation passes.

We also need the following *difference operator*, which is applied over assertions with a free heap variable mem:

$$R_1 \multimap R_2 \quad = \quad \forall h_1, h{:}\mathsf{heap}.\ \mathsf{splits}(\mathsf{init}, h_1, h) \wedge [h_1/\mathsf{mem}]R_1 \supset \exists h_2{:}\mathsf{heap}.\ \mathsf{splits}(\mathsf{mem}, h_2, h) \wedge [h_2/\mathsf{mem}]R_2$$

The informal reading of $R_1 \multimap R_2$ is that the current heap mem is obtained from the initial heap init by replacing a fragment satisfying R_1 in the heap init with a new fragment of which satisfies R_2. The rest of the heaps init and mem agrees. It is not specified, however, which particular fragment of init is changed. If there are several fragments proving R_1, then each of them could have been replaced, but the replacement is always such that the result satisfies R_2. The $\multimap$ operator is used in the strongest postconditions of the various stateful commands, as it is can describe a difference between two successive heaps of the computation.

We can now describe how the judgments capture the semantics of the type of Hoare triples $\Psi.X.\{P\}x{:}A\{Q\}$. Intuitively, a suspended computation $\mathsf{dia}\ E$ should have the type $\Psi.X.\{P\}x{:}A\{Q\}$ if the following two requirements are satisfied:

1. Assuming that the initial heap can be split into two disjoint parts h_1 and h_2 such that P holds of h_1, then E does not get stuck if executed in this initial heap. Moreover, E never touches h_2 (not even for a lookup); in other words, h_2 is not in the footprint of E.

2. Upon termination of E, the fragment h_1 is replaced with a new fragment which satisfies Q, while h_2 remains unchanged.

Notice that the split of the initial heap into h_1 and h_2 is not decided upon before E executes, and need not be unique. We only know that if a split is possible, then the execution of E defines one such split, but which split is chosen may depend on the run-time conditions.

We argue next that the above two requirements are satisfied by E if we can establish that $\Delta; P' \vdash E \Leftarrow x{:}A.\ Q'$, where $P' = \mathsf{this}(\mathsf{init}) \wedge \exists\Psi.X.(P * \top)$ and $Q' = \forall\Psi.X.P \multimap Q$. We call P' and Q' *elaborated* pre- and postconditions for E, respectively.

The requirement (1) is related to the assertion P'. Indeed, P' states that the initial heap can be split into h_1 and h_2 so that h_1 satisfies P and h_2 satisfies $\top$, as required. In order to ensure progress, the typing judgment will allow E to touch only locations whose existence can be proved. Because there is no information available about h_2 and its locations (knowing $\top$ amounts to knowing nothing), E will be restricted to working with h_1 only.

The requirement (2) is related to the assertion Q'. After unraveling the definition of $\multimap$, Q' essentially states that any split into h_1 and h_2 that E may have induced on init results in a final heap where h_1 is replaced with a fragment satisfying Q, while h_2 remains unchanged. But this is precisely what (2) requires.

We can now present the typing rules for computations. We start with the general rules that correspond

to the monadic fragment, and than proceed with the rules for the individual effectful commands.

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

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

$$\frac{\Delta; \mathsf{this}(\mathsf{init}) \wedge \exists \Psi.X.(P * \top) \vdash E \Leftarrow x{:}A.\ \forall \Psi.X.P \multimap Q\,[E']}{\Delta \vdash \mathsf{dia}\ E \Leftarrow \Psi.X.\{P\}x{:}A\{Q\}\,[\mathsf{dia}\ E']}\ \{\ \}\mathsf{I}$$

$$\frac{\begin{array}{c}\Delta \vdash K \Rightarrow \Psi.X.\{R_1\}x{:}A\{R_2\}\,[N'] \\ \Delta; \mathsf{init}, \mathsf{mem}; P \Longrightarrow \exists \Psi.X.(R_1 * \top) \\ \Delta, x{:}A; P \circ (\forall \Psi.X.R_1 \multimap R_2) \vdash E \Rightarrow y{:}B.\ Q\,[E']\end{array}}{\Delta; P \vdash \mathsf{let\ dia}\ x = K\ \mathsf{in}\ E \Rightarrow y{:}B.\ (\exists x{:}A.\ Q)\,[\mathsf{reduce}_A(N', x.\ E')]}\ \{\ \}\mathsf{E}$$

The rule consequent 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 strongest postcondition into an arbitrary postcondition.

The rule comp types the trivial computation that immediately returns the result $x = M$ and performs no changes to the heap. The postcondition simply asserts the equality of the canonical forms of x and M in addition to the precondition P.

The rule $\{\ \}\mathsf{I}$ is the most important rule of the system, as it defines the type of Hoare triples, and provides it with a small footprint semantics. As discussed before, this is achieved by requiring that the premise $\Delta; P' \vdash E \Leftarrow x{:}A.\ Q'$ is satisfied.

The rule $\{\ \}\mathsf{E}$ describes how a suspended computation $K \Rightarrow \{R_1\}x{:}A\{R_2\}$ can be sequentially composed with another computation E. The two can be composed if the following are satisfied. First, the the assertion logic must establish that the precondition P of the composite computation ensures that the current heap contains a fragment satisfying the precondition R_1 for K. In other words, we need to show that $P \Longrightarrow \exists \Psi.X.(R_1 * \top)$. Second, the computation E needs to check against the postcondition obtained after executing K. The later is taken to be $P \circ \forall \Psi.X.R_1 \multimap R_2$, because the computation encapsulated by K is executed in the heap of which P holds, and we know that the change it imposes on this heap can be described as $\forall \Psi.X.R_1 \multimap R_2$. The normal form of the whole computation is obtained by invoking the auxiliary function reduce.

We notice here that 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, and is by assumption well-formed in the context Δ. In particular, B does not depend on the variable x, so 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, which is an output of the judgment, so we need to existentially abstract x in the postcondition of the conclusions, or otherwise x will be a dangling variable. Similar remark applies to the rules for the specific effectful constructs for allocation, lookup, strong update and deallocation that we present next.

$$\frac{\begin{array}{l}\Delta \vdash \tau \Leftarrow \mathsf{mono}\,[\tau'] \\ \Delta \vdash M \Leftarrow \tau'\,[M'] \quad \Delta, x{:}\mathsf{nat}; P * (x \mapsto_{\tau'} M') \vdash E \Rightarrow y{:}B.\ Q\,[E']\end{array}}{\Delta; P \vdash x = \mathsf{alloc}_\tau(M); E \Rightarrow y{:}B.\ (\exists x{:}\mathsf{nat}.\ Q)\,[x = \mathsf{alloc}_{\tau'}(M'); E']}\ \mathsf{alloc}$$

$$\frac{\begin{array}{ll}\Delta \vdash \tau \Leftarrow \mathsf{mono}\,[\tau'] & \Delta; \mathsf{init}, \mathsf{mem}; P \Longrightarrow M' \hookrightarrow_{\tau'} - \\ \Delta \vdash M \Leftarrow \mathsf{nat}\,[M'] & \Delta, x{:}\tau'; P \wedge (M' \hookrightarrow_{\tau'} \mathsf{expand}_{\tau'}(x)) \vdash E \Rightarrow y{:}B.\ Q\,[E']\end{array}}{\Delta; P \vdash x = [M]_\tau; E \Rightarrow y{:}B.\ (\exists x{:}\tau'.\ Q)\,[x = [M']_{\tau'}; E']}\ \mathsf{lookup}$$

$$\frac{\begin{array}{ll}\Delta \vdash \tau \Leftarrow \mathsf{mono}\,[\tau'] & \\ \Delta \vdash M \Leftarrow \mathsf{nat}\,[M'] & \Delta; \mathsf{init}, \mathsf{mem}; P \Longrightarrow M' \hookrightarrow - \\ \Delta \vdash N \Leftarrow \tau'\,[N'] & \Delta; P \circ ((M' \mapsto -) \multimap (M' \mapsto_{\tau'} N')) \vdash E \Rightarrow y{:}B.\ Q\,[E']\end{array}}{\Delta; P \vdash [M]_\tau = N; E \Rightarrow y{:}B.\ Q\,[[M']_{\tau'} = N'; E']}\ \mathsf{update}$$

$$\frac{\begin{array}{ll} & \Delta; \mathsf{init}, \mathsf{mem}; P \Longrightarrow M' \hookrightarrow - \\ \Delta \vdash M \Leftarrow \mathsf{nat}\,[M'] & \Delta; P \circ ((M' \mapsto -) \multimap \mathsf{emp}) \vdash E \Rightarrow y{:}B.\ Q\,[E']\end{array}}{\Delta; P \vdash \mathsf{dealloc}(M); E \Rightarrow y{:}B.\ Q\,[\mathsf{dealloc}(M'); E']}\ \mathsf{dealloc}$$

In the case of allocation, E is checked against a precondition $P * (x \mapsto_{\tau'} M')$, which is the postcondition obtained from P after the allocation. Notice how this assertion states that the newly allocated memory whose address is stored in the variable x is disjoint from any already allocated memory that is described by P.

In the case of lookup, the strongest postcondition states that the heap is not changed (i.e., P still holds) but we have the additional knowledge that the variable x stores the looked up value. Thus, the precondition for checking E is $P \wedge (M' \hookrightarrow_{\tau'} \mathsf{expand}_{\tau'}(x))$. We need to expand x, because the input assertions to the typing judgments must be in canonical form. In order to ensure progress, we must also establish that the location M' actually exists in the current heap, and points to a value of type τ'. In other words, we must prove that $P \Longrightarrow M' \hookrightarrow_{\tau'} -$. Here the proposition $M' \hookrightarrow_{\tau'} -$ is an abbreviation for $\exists x{:}\tau'.\ (M \hookrightarrow_{\tau'} \mathsf{expand}_{\tau'}(x))$, which is in canonical form.

It is important to notice that proving this sequent may be postponed, as it is not essential for the other premises of the rule. The sequent can simply be collected as part of the verification condition for the computation, and attempted later. This property will be true of all the sequents involved in the computation judgments.

In the case of update, the precondition for E must state that the old value of the location M' is replaced with a new one given by N'. Hence the precondition $P \circ ((M' \mapsto -) \multimap (M' \mapsto_{\tau'} N'))$. $M' \mapsto -$ is an abbreviation for the canonical proposition $\exists \alpha.\ \exists x{:}\alpha.\ M' \mapsto_\alpha \mathsf{eta}_\alpha\ x$, which states that the location M' is allocated, but is not specific about the type or the value that M' points to. To ensure progress, it must be shown that P implies $M' \hookrightarrow -$. Because the type of the old value pointed to by M' is existentially abstracted, this rule implements strong updates.

In the case of deallocation, the precondition for E must state that the location M' has been removed from the heap. This is equivalent to saying that the heap fragment containing M' has been replaced by an empty heap fragment. Hence the precondition $P \circ ((M' \mapsto -) \multimap \mathsf{emp})$. Again, progress can be made only if M' is actually allocated, which must be provable in the assertion logic.

It is important to observe that all of the rules for the primitive commands are independent of

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.

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

We notice here that P_1 and P_2 may overlap significantly, because they both contain the proposition P as a subexpression. Thus, HTT typing rules currently do not generate postconditions that are optimal for space. We leave improvements in this direction for future work.

Finally, we present the rule for recursion. 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 $\Pi x{:}A.\ \Psi.X.\{R_1\}y{:}B\{R_2\}$, expressing that f is a function whose range is a computation with precondition R_1 and postcondition R_2.

$$\frac{\begin{array}{c}\Delta \vdash T \Leftarrow \mathsf{type}\,[\Pi x{:}A.\Psi.X.\{R_1\}y{:}B\{R_2\}] \\ \Delta \vdash M \Leftarrow A\,[M'] \\ \Delta; \mathsf{init}, \mathsf{mem}; P \Longrightarrow [M'/x]^p_A(\exists \Psi.X.(R_1 * \top)) \\ \Delta, f{:}\Pi x{:}A.\ \Psi.X.\{R_1\}y{:}B\{R_2\}, x{:}A; \mathsf{this}(\mathsf{init}) \wedge \exists \Psi.X.(R_1 * \top) \vdash E \Leftarrow y{:}B.\ (\forall \Psi.X.R_1 \multimap R_2)\,[E'] \\ \Delta, y{:}[M'/x]^p_A(B); P \circ [M'/x]^p_A(\forall \Psi.X.R_1 \multimap R_2) \vdash F \Rightarrow z{:}C.\ Q\,[F']\end{array}}{\Delta; P \vdash y = \mathsf{fix}_T(M, f.x.E); F \Rightarrow z{:}C.\ (\exists y{:}[M'/x]^p_A(B).Q)\,[y = \mathsf{fix}_{\Pi x:A.\Psi.X.\{R_1\}y:B\{R_2\}}(M', f.x.E'); F']}$$

Before M can be applied to the recursive function, and the obtained computation executed, we need to check that the main precondition P implies $\exists \Psi.X.(R_1 * \top)$, so that the heap contains a fragment that satisfies R_1. After the recursive call we are in a heap that is changed according to the proposition $\forall \Psi.X.R_1 \multimap R_2$, the computation F following the recursive call is checked with a precondition $P \circ (\forall \Psi.X.R_1 \multimap R_2)$. Of course, because the recursive calls are started using M for the argument x, we need to substitute the canonical M' for x everywhere.

Example. As a second example, consider the function sumfunc that takes an argument n and computes the sum $1 + \cdots + n$. The function first allocates a which will store the partial sums, then increments the contents of a with successive nats in a loop, until n is reached. Then a is deallocated before its contents is returned as the final result.

We present the code for sumfunc below, and annotate it with assertions (enclosed in braces and labeled) that are generated during typechecking at the various control points. In the code, we assumed given the ordering $\leq$, and introduced the following abbreviations: (1) if M then E else F is short for $\mathsf{if}(M, E, F)$; (2) $\mathsf{sum}(r, n) = \mathsf{Id}_{\mathsf{nat}}(2 \times r, n \times n + 1)$ denoting that $r = 1 + \cdots + n$; (4) $I = i \leq n \wedge \exists t{:}\mathsf{nat}.\ a \mapsto_{\mathsf{nat}} t \wedge \mathsf{sum}(t, i)$ will be the loop invariant during the summation; (5) $Q = a \mapsto_{\mathsf{nat}} - \wedge \mathsf{sum}(x, n)$ asserts what holds upon the exit from the loop.

```
sumfunc : Πn:nat. {emp} r : nat {emp ∧ sum(r, n)} =
 λn. dia(a = alloc_nat(0);
         P0:{this(init) * (a ↦_nat 0)}
         x = fix(0, f. i.
                   P1:{this(init) ∧ (I * ⊤)}
                   s = [a]_nat;
                   P2:{P1 ∧ a ↪_nat s}
                   t = if eq(i, n) then
                         P3:{P2 ∧ Id_nat(i, n)}
                         s
                       else
```

```
                    P4:{P2 ∧ ¬ Id_nat(i, n)}
                    [a]_nat = s+i+1;
                    P5:{P4 ∘ (a ↦_nat - ⊸ a ↦_nat s+i+1)}
                    let dia x = f (i+1)
                    in
                       P6:{P5 ∘ ([i+1/i]I ⊸ Q)}
                       x
                    end;
                P7:{(P3 ∧ Id_nat(t, s)) ∨
                     (∃x:nat. P6 ∧ Id_nat(t, x))}
                t);
P8:{P0 ∘ ([0/i]I ⊸ Q)}
dealloc(a);
P9:{P8 ∘ (a ↦_nat - ⊸ emp)}
x);
```

The specification for sumfunc states that the function starts and ends with an empty heap. The most interesting part of the code is the recursive loop. It introduces the fixpoint variable f, whose type we take to be $f{:}\Pi i{:}\mathsf{nat}.\ \{I\}x{:}\mathsf{nat}\{Q\}$, giving the loop invariant in the precondition. The variable i is the counter which drives the loop. The initial value for i is 1, as specified in the first argument of the fixpoint construct, and the loop terminates when i reaches n.

The verification condition consists of the following sequents: (1) $P_1 \Longrightarrow a \hookrightarrow_{\mathsf{nat}} -$, so that a can be looked up, (2) $P_4 \Longrightarrow a \hookrightarrow -$ so that a can be updated, (3) $P_5 \Longrightarrow [i+1/i]I * \top$, so that the computation obtained from $f(i+1)$ can be executed, (4) $P_7 \wedge \mathsf{Id}_{\mathsf{nat}}(x,t) \Longrightarrow I \multimap Q$, so that the fixpoint satisfies the prescribed postcondition, (5) $P_8 \Longrightarrow a \hookrightarrow -$ so that a can be deallocated, and (6) $P_9 \wedge \mathsf{Id}_{\mathsf{nat}}(r,x) \Longrightarrow \mathsf{emp} \multimap \mathsf{emp} \wedge \mathsf{sum}(r,n)$, so that sumfunc has the required postcondition. It is not too hard to see that all these sequents are valid.

5 Properties

In this section we present the basic properties of HTT, leading up to the substitution principles. Somewhat surprisingly, the development has the exact same structure as our previous HTT proposal [26], and the addition of polymorphism and the small footprint property do not require significant new lemmas and theorems. This is explained by noticing that the new HTT and the old HTT both work in much the same way by generating strongest postconditions of the computations, and the main input of the theorems in this section is to establish that this process respects the semantics of computations.

While we have the same theorems as before, each of them has new cases that arise from the additions of this paper. In the following text, we present these theorems and the selected cases of their proofs; the presentation is self-contained.

We start with several auxiliary lemmas and theorems expressing general properties of the typing judgments, computations and substitutions. We proceed to study the properties of canonical forms, and establish substitution principles for the fragment of HTT consisting of canonical forms only. Then the results are lifted to general (i.e., non-canonical forms).

First, we prove that typechecking in HTT can be reduced to proving in the assertion logic.

Theorem 7 (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.

Proof: The typing judgments of HTT are syntax directed; their premises always involve typechecking smaller expressions, or deciding syntactic equality of types, or computing hereditary substitutions, or deciding sequents of the assertion logic. 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. ■

The above theorem assumes an oracle that decides the sequents of the assertion logic. As customary in the Proof-Carrying Code architecture [28], it should be possible to replace the oracle with a certificate that serves as a checkable witness of the sequents' validity. Then the above theorem will loose the attribute "relative"; typechecking will include proof checking for sequents, and thus become decidable. With this extensions, an HTT computation judgment will contain all the information needed to establish its own derivation, as the derivation process is completely guided by the syntax of the computation. In the terminology of Martin-Löf [21], the judgments become *analytic*. An alternative view of this property is that an HTT computation can be seen as a proof of its own specification. In other words, the effectful fragment of HTT satisfies the Curry-Howard correspondence between computations and specification proofs [15].

There is another way to interpret Theorem 7. As explained in Section 4, the collection of assertion logic sequents encountered during typechecking may be considered as the verification condition for the expression being typechecked. Thus, the theorem may be seen as stating that verification conditions for HTT are computable.

The HTT judgments satisfy the usual structural properties of weakening and contraction.

Lemma 8 (Structural properties)
Let $\Delta \vdash J$ range over the judgments of HTT type theory which depend on a variable context Δ, and let $\Delta; \Psi \vdash J$ range over judgments which depend on both Δ and a heap context Ψ. Then the following holds

1. ***Variable context weakening.*** *If $\Delta \vdash J$ and $\Delta \vdash A \Leftarrow \mathsf{type}\,[A]$, then $\Delta, x{:}A \vdash J$*
2. ***Heap context weakening.*** *If $\Delta; \Psi \vdash J$ then $\Delta; \Psi, h \vdash J$*
3. ***Variable context contraction.*** *If $\Delta, x{:}A, \Delta_1, y{:}A, \Delta_2 \vdash J$ then $\Delta, x{:}A, \Delta_1, [x/y]\Delta_2 \vdash [x/y]J$*
4. ***Heap context contraction.*** *If $\Delta; \Psi, h, \Psi_1, g, \Psi_2 \vdash J$ then $\Delta; \Psi, h, \Psi_1, \Psi_2 \vdash [h/g]J$.*

Proof: By straightforward induction on the derivation of J. ■

The closed closed canonical forms of type nat are numerals, and closed canonical forms of type bool are the constants true and false, as we show below.

Lemma 9 (Closed canonical forms of primitive type)
1. *If $\cdot \vdash M \Leftarrow \mathsf{nat}\,[M]$ then $M = \mathsf{s}^n\mathsf{z}$ for some natural number n.*
2. *If $\cdot \vdash M \Leftarrow \mathsf{bool}\,[M]$ then $M = \mathsf{true}$ or $M = \mathsf{false}$.*

Proof: By induction on the structure of the involved expressions. For the first statement, M can be z or $\mathsf{s}\ N$ or an arithmetic expression $N_1 + N_2$ or $N_1 \times N_2$. The first two cases are trivial to show. On the other hand, the last two cases cannot be canonical. Indeed, by induction hypothesis, N_1 and N_2 must be numerals, and so the primitive functions $\mathsf{plus}(N_1, N_2)$ and $\mathsf{times}(N_1, N_2)$ which are used to compute the canonical forms of addition and multiplication cannot return $N_1 + N_2$ and $N_1 \times N_2$, respectively. The proof of the second statement is similar, so we omit it. ■

The next lemma formalizes the property of heap substitutions. Heap substitutions are not hereditary; they do not introduce any redexes and are total as functions on expressions. Thus, the lemma can be proved prior to any properties of hereditary substitutions.

Lemma 10 (Heap substitution principles)
Let $\Delta; \Psi \vdash H \Leftarrow \mathsf{heap}\,[H']$. Then:

1. *If $\Delta; \Psi, h, \Psi_1 \vdash H_1 \Leftarrow \mathsf{heap}\,[H'_1]$, then $\Delta; \Psi, \Psi_1 \vdash [H/h]H_1 \Leftarrow \mathsf{heap}\,[[H'/h]H'_1]$.*

2. *If* $\Delta; \Psi, h, \Psi_1 \vdash P \Leftarrow \mathsf{prop}\,[P']$, *then* $\Delta; \Psi, \Psi_1 \vdash [H/h]P \Leftarrow \mathsf{prop}\,[[H'/h]P']$.

3. *If* $\Delta; \Psi, h, \Psi_1; \Gamma_1 \Longrightarrow \Gamma_2$, *then* $\Delta; \Psi, \Psi_1, [H'/h]\Gamma_1 \Longrightarrow [H'/h]\Gamma_2$.

Proof: By straightforward mutual induction on the structure of expressions being substituted into. We also use the property that heap substitutions commute with hereditary substitutions, without stating it explicitly. ■

The next several lemmas explore the properties of canonical forms. First we note that type substitution commutes with expansions.

Lemma 11 (Type substitution and expansion)
For canonical τ, A *and* N, *we have:*

$$[\tau/\alpha]\mathit{expand}_A(N) = \mathit{expand}_{[\tau/\alpha]A}([\tau/\alpha]N)$$

Proof: By straightforward induction on A. ■

Then we establish the properties of hereditary substitution of variable expansions, as a first step towards proving the substitution principle. If $x{:}A$ is a free variable in a well-typed expression N, then hereditarily substituting $\mathsf{expand}_A(x)$ for x does not change N. Intuitively, this holds because the typing ensures that x is correctly used in N, so that when $\mathsf{expand}_A(x)$ is substituted, the redexes that are created and hereditarily reduced do not influence the result of the substitution.

Lemma 12 (Properties of variable expansion)

1. *If* $\Delta, x{:}A, \Delta_1 \vdash K \Rightarrow B\,[K]$, *then* $[\mathit{expand}_A(x)/x]^k_A(K)$ *exists, and*

   (a) *if* $[\mathit{expand}_A(x)/x]^k_A(K) = K'$ *is an elim term, then* $K' = K$

   (b) *if* $[\mathit{expand}_A(x)/x]^k_A(K) = N' :: S$ *is an intro term, then* $N' = \mathit{expand}_B(K)$, *and* $S = B^-$.

2. *If* $\Delta, x{:}A, \Delta_1 \vdash N \Leftarrow B\,[N]$, *then* $[\mathit{expand}_A(x)/x]^m_A(N) = N$.

3. *If* $\Delta, x{:}A, \Delta_1; \Psi.X.P \vdash E \Leftarrow y{:}B.\ Q\,[E]$, *then* $[\mathit{expand}_A(x)/x]^e_A(E) = E$.

4. *If* $\Delta, x{:}A, \Delta_1 \vdash B \Leftarrow \mathsf{type}\,[B]$, *then* $[\mathit{expand}_A(x)/x]^a_A(B) = B$.

5. *If* $\Delta, x{:}A, \Delta_1; \Psi \vdash P \Leftarrow \mathsf{prop}\,[P]$, *then* $[\mathit{expand}_A(x)/x]^p_A(P) = P$.

6. *If* $\Delta, x{:}A, \Delta_1; \Psi \vdash H \Leftarrow \mathsf{heap}\,[H]$, *then* $[\mathit{expand}_A(x)/x]^h_A(H) = H$.

7. *If* $\Delta \vdash M \Leftarrow A\,[M]$, *then* $[M/x]^m_A(\mathit{expand}_A(x)) = M$.

8. *If* $\Delta; \Psi.X.P \vdash E \Leftarrow x{:}A.\ Q\,[E]$, *then* $\langle E/x\rangle_A(\mathit{expand}_A(x)) = E$.

Proof: By mutual nested induction, first on the structure of A^-, and then on the structure of the involved examples. We omit the particular cases, as they are same as in our earlier proposal [26]. The new cases arising from the addition of polymorphism are completely straightforward. ■

The next lemma establishes the identity principle of the assertion logic sequent calculus; that is from an assumption P, we can prove a conclusion P, where P is an arbitrary proposition. In Section 4, initial sequents were restricted to primitive propositions, so now the identity principle needs to be explicitly proved when P is not primitive. Simultaneously, we must show that substitutability of equality holds for arbitrary propositions, and that expansions of well-typed elimination terms are well-typed themselves.

Lemma 13 (Identity principles)

1. *If* $\Delta; \Psi \vdash P \Leftarrow \mathsf{prop}\,[P]$, *then* $\Delta; \Psi; \Gamma_1, P \Longrightarrow P, \Gamma_2$.

2. *If* $\Delta;\Psi;\Gamma_1, \mathsf{Id}_B(M,N) \Longrightarrow [M/x]^p_B(P), \Gamma_2$ *and* $[N/x]^p_B(P)$ *is well-formed and canonical (i.e.,* $\Delta;\Psi \vdash [N/x]^p_B(P) \Leftarrow \mathsf{prop}\,[[N/x]^p_B(P)]$*), then* $\Delta;\Psi;\Gamma_1, \mathsf{Id}_B(M,N) \Longrightarrow [N/x]^p_B(P), \Gamma_2$.

3. *If* $\Delta;\Psi;\Gamma_1, \mathsf{HId}(H_1,H_2) \Longrightarrow [H_1/h]P, \Gamma_2$ *then* $\Delta;\Psi;\Gamma_1, \mathsf{HId}(H_1,H_2) \Longrightarrow [H_2/h]P, \Gamma_2$.

4. *If* $\Delta \vdash K \Rightarrow A\,[K]$, *then* $\Delta \vdash \mathsf{expand}_A(K) \Leftarrow A\,[\mathsf{expand}_A(K)]$.

Proof: While the overall structure of this proof remains the same as in [26], there are changes that arise because the correspondence between Hoare types and the computation judgments is now different due to the small footprint approach.

Whereas before, the proof was by simultaneous induction on the structures of P and A, now we work with a "translation" of A in which any Hoare triple $\Psi.X.\{P\}x{:}B\{Q\}$ of A is replaced with $\{P'\}x{:}B\{Q'\}$, where $P' = \exists\Psi.X.(P * \top)$ and $Q' = \forall\Psi.X.P \multimap Q$. The reader may recognize the translated assertions P' and Q' as the elaborated pre- and postcondition from Section 4.

We next present a proof of a case of statement 4 when $A = \Psi.X.\{P\}x{:}B\{Q\}$. This case is obviously specific to the small footprint approach. In this case, we have $\mathsf{expand}_A(K) = \mathsf{dia}\,(\mathsf{let\ dia}\ y = K\ \mathsf{in}\ \mathsf{expand}_B(y))$. In order for this term to check against A, the typing rules require that the following sequents be proved, where we assume that Ψ' is an α-renaming of the context Ψ and X' is an alpha renaming of the heap context X.

1. $\Delta;\mathsf{init},\mathsf{mem};\mathsf{this}(\mathsf{init}) \wedge \exists\Psi.X.(P * \top) \Longrightarrow \exists\Psi.X.(P * \top)$

2. $\Delta, x{:}B;\mathsf{init},\mathsf{mem};\exists y{:}B.\mathsf{this}(\mathsf{init}) \wedge \exists\Psi.X.(P * \top) \circ \forall\Psi.X.(P \multimap [y/x]Q) \wedge$
$\wedge \mathsf{Id}_B(\mathsf{expand}_B(x), \mathsf{expand}_B(y)) \Longrightarrow \forall\Psi.X.(P \multimap Q)$

The first sequent shows that the precondition for K is satisfied at the point in the computation where K is executed; it is trivial to derive by applying the left rule for conjunction, and then appealing inductively to statement 1.

The second sequent shows that the strongest postcondition generated for $\mathsf{let\ dia}\ y = K\ \mathsf{in}\ \mathsf{expand}_B(y)$ with respect to the precondition P' actually implies Q'. Notice that the sequent is well-formed because, by induction hypothesis on B, $\mathsf{expand}_B(x)$ and $\mathsf{expand}_B(y)$ are canonical. To prove this sequent, we first remove the existential quantification over y, and then inductively apply the statement 2 of this lemma, and the properties of variable expansion to deal with the substitution $[x/y]Q$ and obtain a simpler

$$\Delta, x{:}B, y{:}B;\mathsf{init},\mathsf{mem};\mathsf{this}(\mathsf{init}) \wedge \exists\Psi.X.(P * \top) \circ \forall\Psi.X.(P \multimap Q) \Longrightarrow \forall\Psi.X.(P \multimap Q)$$

The inductive steps are justified because Q is a strict subexpression of the translation of A.

Now we expand the definition of $\circ$ to introduce a new heap variable h, but then immediately use the property 3 to equate h and init, thus obtaining.

$$\Delta, x{:}B, y{:}B;\mathsf{init},\mathsf{mem};\mathsf{this}(\mathsf{init}) \wedge \exists\Psi.X.(P * \top), \forall\Psi.X.(P \multimap Q) \Longrightarrow \forall\Psi.X.(P \multimap Q)$$

Now, we simply invoke inductively statement 1 on $Q' = \forall\Psi.X.(P \multimap Q)$ to derive the conclusion. This step is justified, because Q' is a subexpression of the translation of A. ■

The next lemma restates in the context of HTT the usual properties of Hoare Logic, like weakening of the consequent and strengthening of the precedent. Also included is the property on the preservation of history, which states that a computation does not depend on how the heap in which it executes has been obtained. Thus, if the computation has an elaborated precondition P and a postcondition Q, these can be composed with an arbitrary proposition R into a new precondition $R \circ P$ and a new postcondition $R \circ Q$.

Lemma 14 (Properties of computations)
Suppose that $\Delta; P \vdash E \Leftarrow x{:}A.\ Q\,[E']$. *Then:*

1. Weakening Consequent. *If* $\Delta, x{:}A;\mathit{init},\mathit{mem};Q \Longrightarrow R$, *then* $\Delta; P \vdash E \Leftarrow x{:}A.\ R\,[E']$.

2. Strengthening Precedent. *If* $\Delta; \mathit{init}, \mathit{mem}; R \Longrightarrow P$, *then* $\Delta; R \vdash E \Leftarrow x{:}A.\ Q\,[E']$.

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

Proof: The structure of the proof is generally the same as in [26], but many cases look different because of the small footprint. We present some of the cases here.

Weakening of consequent is proved in the same way as before. From $\Delta; P \vdash E \Leftarrow x{:}A.\ Q\,[E']$ we know that there exists a proposition S, such that $\Delta; P \vdash E \Rightarrow x{:}A.\ S$ where $\Delta, x{:}A; \mathsf{init}, \mathsf{mem}; S \Longrightarrow Q$. Applying the rule of cut, we get $\Delta, x{:}A; \mathsf{init}, \mathsf{mem}; S \Longrightarrow R$, and thus $\Delta; P \vdash E \Leftarrow x{:}A.\ R\,[E']$.

Strengthening precedent and preservation of history are proved by induction on the structure of E. In both statements, the characteristic case is $E = \mathsf{let\ dia}\ y = K\ \mathsf{in}\ F$. In this case, from the typing of E we obtain: $\Delta \vdash K \Rightarrow \Psi.X.\{R_1\}y{:}B\{R_2\}\,[N']$ where $\Delta; \mathsf{init}, \mathsf{mem}; P \Longrightarrow \exists\Psi.X.(R_1 * \top)$, and $\Delta, y{:}B; P \circ \forall\Psi.X.(R_1 \multimap R_2) \vdash F \Rightarrow x{:}A.\ S\,[F']$ where also $\Delta, x{:}A; \mathsf{init}, \mathsf{mem}; \exists y{:}B.\ S \Longrightarrow Q$, and $E' = \mathsf{reduce}_B(N', y.\ F')$.

For strengthening precedent, $(\Delta; R \vdash E \Leftarrow x{:}A.\ Q\,[E'])$, we need to establish that:

1. $\Delta; \mathsf{init}, \mathsf{mem}; R \Longrightarrow \exists\Psi.X.(R_1 * \top)$, and

2. $\Delta, y{:}B; R \circ \forall\Psi.X.(R_1 \multimap R_2) \vdash F \Rightarrow x{:}A.\ S'\,[F']$ for some proposition S' such that $\Delta, x{:}A; \exists y{:}B.\ S' \Longrightarrow Q$.

The sequent (1) follows by the rule of cut, from the assumption $R \Longrightarrow P$ and the sequent $P \Longrightarrow \exists\Psi.X.(R_1 * \top)$ obtained from the typing of E. To derive (2), we first observe that $\Delta, y{:}B; P \circ \forall\Psi.X.(R_1 \multimap R_2) \vdash F \Rightarrow x{:}A.\ S\,[F']$ implies $\Delta, y{:}B; P \circ \forall\Psi.X.(R_1 \multimap R_2) \vdash F \Leftarrow x{:}A.\ S\,[F']$, by the inference rule consequent, and using the identity principle (Lemma 13) to establish $S \Longrightarrow S$. It is also easy to show that the sequent $\Delta; \mathsf{init}, \mathsf{mem}; R \circ \forall\Psi.X.(R_1 \multimap R_2) \Longrightarrow P \circ \forall\Psi.X.(R_1 \multimap R_2)$ is derivable, after first expanding the definition of the propositional connective "$\circ$". Now, by induction hypothesis on F, we have $\Delta, y{:}B; R \circ \forall\Psi.X.(R_1 \multimap R_2) \vdash F \Leftarrow x{:}A.\ S\,[F']$.

The later means that there exists a proposition S' such that $\Delta, y{:}B; R \circ \forall\Psi.X.(R_1 \multimap R_2) \vdash F \Rightarrow x{:}A.\ S'\,[F']$ where $\Delta, y{:}B, x{:}A; \mathsf{init}, \mathsf{mem}; S' \Longrightarrow S$. But then we can clearly also have the sequent $\Delta, x{:}A; \mathsf{init}, \mathsf{mem}; \exists y{:}B.\ S' \Longrightarrow \exists y{:}B.\ S$. Now, by the rule of cut applied to the sequent $\exists y{:}B.\ S \Longrightarrow Q$ (which was derived from the typing of E), we obtain $\Delta, x{:}A; \exists y{:}B.\ S' \Longrightarrow Q$, which finally shows the derivability of (2).

In order to show preservation of history $(\Delta; R \circ P \vdash E \Leftarrow x{:}A.\ (R \circ Q)\,[E'])$, we need to establish that:

3. $\Delta; \mathsf{init}, \mathsf{mem}; R \circ P \Longrightarrow \exists\Psi.X.(R_1 * \top)$, and

4. $\Delta, y{:}B; (R \circ P) \circ \forall\Psi.X.(R_1 \multimap R_2) \vdash F \Rightarrow x{:}A.\ S'\,[F']$ where $\Delta, x{:}A; \mathsf{init}, \mathsf{mem}; \exists y{:}B.\ S' \Longrightarrow (R \circ Q)$.

Sequent (3) follows by cut from the sequents $(R \circ P) \Longrightarrow [h/\mathsf{init}]P$ and $[h/\mathsf{init}]P \Longrightarrow \forall\Psi.X.(R_1 * \top)$. The first sequent is trivially obtained after expanding the definition of "$\circ$". The second sequent follows from $P \Longrightarrow \forall\Psi.X.(R_1 * \top)$ by heap substitution principles and the fact that R_1 does not depend on the heap variable init (as evident from the formation rule for the type $\Psi.X.\{R_1\}y{:}B\{R_2\}$). To derive (4), we apply the induction hypothesis on the typing derivation for F, to obtain $\Delta, y{:}B; R \circ (P \circ (\forall\Psi.X.(R_1 \multimap R_2))) \vdash F \Leftarrow x{:}A.\ (R \circ S)$. This gives us $\Delta, y{:}B; (R \circ P) \circ \forall\Psi.X.(R_1 \multimap R_2) \vdash F \Leftarrow x{:}A.\ (R \circ S)$ by using strengthening of precedent and associativity of "$\circ$", i.e. the fact that $R \circ (P \circ X) \Longrightarrow (R \circ P) \circ X$ (for any R, P and X), which is easy to show.

The last derivation means that $\Delta, y{:}B; (R \circ P) \circ \forall\Psi.X.(R_1 \multimap R_2) \vdash F \Rightarrow x{:}A.\ S'$ for some proposition S' for which $\Delta, y{:}B; \mathsf{init}, \mathsf{mem}; S' \Longrightarrow (R \circ S)$. By the rules of the assertion logic, and the fact that $y \notin \mathsf{FV}(R)$, we now have $\exists y{:}B.\ S' \Longrightarrow \exists y{:}B.\ (R \circ S) \Longrightarrow R; \exists y{:}B.\ S \Longrightarrow (R \circ Q)$. By cut, $\exists y{:}B.\ S' \Longrightarrow (R \circ Q)$, thus proving the derivability of (4).

The other cases of Preservation of History are proved in a similar way relying on the properties that $R \circ (P * X) = (R \circ P) * X$ (in the case of append) and $R \circ (P \wedge X) = (R \circ P) \wedge X$ (in the case of lookup). Both of these equations are easy to prove; the action of X is on the current heap mem, but after expanding the definition of $\circ$ all the occurrences of mem in R are substituted with a fresh heap variable, which is thus not influence by X. ■

Preservation of History is important because of the way HTT computes strongest postconditions. It essentially states that it is irrelevant what kind of computation lead to the creation of the current heap (that is, which sequence of strongest postcondition calculations is lead to the creation of the proposition representing the current heap). Rather, what matters is only the what is true of the heap at the moment.

A similar properties does not seem to have been considered by, say, Separation Logic, but we believe that may be because, to the best of our knowledge, no work on Separation Logic has based the semantics of the Hoare triples on the notion of strongest postconditions. It is usually the other way around; triples are defined semantically, and then the strongest postconditions are derived as admissible rules.

One of the main properties of Separation Logic is the frame rule, which captures the essence of small footprints. Of course, the frame rule is admissible in HTT as well, as shown by the next lemma.

Lemma 15 (Frame)
If $\Delta \vdash \mathit{dia}\ E \Leftarrow \Psi.X.\{P\}x{:}A\{Q\}\,[E']$, *and* $\Delta, \Psi; X, \mathit{mem} \vdash R \Leftarrow \mathsf{prop}\,[R]$, *then* $\Delta \vdash \mathit{dia}\ E \Leftarrow \Psi.X.\{P * R\}x{:}A\{Q * R\}\,[E']$.

Proof: From the assumption on the typing of $\mathsf{dia}\ E$, we obtain $\Delta; \mathsf{this(init)} \wedge \exists\Psi.X.(P * \top) \vdash E \Leftarrow x{:}A.\ \forall\Psi.X.(P \multimap Q)$.

Notice that that following sequents are derivable.

1. $\mathsf{this(init)} \wedge \exists\Psi.X.(P * R * \top) \Longrightarrow \mathsf{this(init)} \wedge \exists\Psi.X.(P * \top)$, and
2. $\forall\Psi.X.(P \multimap Q) \Longrightarrow \forall\Psi.X.(P * R \multimap Q * R)$.

Both are proved easily, in the second case after expanding the definition of $\multimap$. Now the result follows from the typing of E, by strengthening the precedent using (1) and weakening the consequent using (2).

■

The next lemma formulates the properties of the monotype substitution into canonical forms. The monotype substitution is not hereditary, so the lemma can easily be established by simple induction.

Lemma 16 (Canonical monotype substitution principles)
Suppose that $\Delta \vdash \tau \Leftarrow \mathsf{mono}\,[\tau]$, *and* $\vdash \Delta, \alpha, \Delta_1\ \mathsf{ctx}$. *Denote by* $(-)'$ *the operation of monotype substitution* $[\tau/\alpha](-)$, *and suppose that the context* $\Delta_1' = [\tau/\alpha](\Delta_1)$ *is well-formed (i.e.* $\vdash \Delta, \Delta_1'\ \mathsf{ctx}$*). Then the following holds.*

1. *If* $\Delta, \alpha, \Delta_1 \vdash K \Rightarrow B\,[K]$, *then the type* B' *is well-formed (i.e.* $\Delta, \Delta_1' \vdash B' \Leftarrow \mathsf{type}\,[B']$*), and* $\Delta, \Delta_1' \vdash K' \Rightarrow B'\,[K']$.
2. *If* $\Delta, \alpha, \Delta_1 \vdash N \Leftarrow B\,[N]$, *and the type* B' *is well-formed (i.e.,* $\Delta, \Delta_1' \vdash B' \Leftarrow \mathsf{type}\,[B']$*), then* $\Delta, \Delta_1' \vdash N' \Leftarrow B'\,[N']$.
3. *If* $\Delta, \alpha, \Delta_1; P \vdash E \Leftarrow y{:}B.\ Q\,[E]$, *and* $y \notin \mathit{FV}(M)$, *and the propositions* P' *and* Q' *and the type* B' *and are well-formed (i.e.,* $\Delta, \Delta_1'; \mathit{init}, \mathit{mem} \vdash P' \Leftarrow \mathsf{prop}\,[P']$, $\Delta, \Delta_1' \vdash B' \Leftarrow \mathsf{type}\,[B']$ *and* $\Delta, \Delta_1', y{:}B'; \mathit{init}, \mathit{mem} \vdash Q' \Leftarrow \mathsf{prop}\,[Q']$*), then* $\Delta, \Delta_1'; P' \vdash E' \Leftarrow y{:}B'.\ Q'\,[E']$.
4. *If* $\Delta, \alpha, \Delta_1 \vdash B \Leftarrow \mathsf{type}\,[B]$, *then* $\Delta, \Delta_1' \vdash B' \Leftarrow \mathsf{type}\,[B']$.
5. *If* $\Delta, \alpha, \Delta_1; X \vdash P \Leftarrow \mathsf{prop}\,[P]$, *then* $\Delta, \Delta_1'; X \vdash P' \Leftarrow \mathsf{prop}\,[P']$.
6. *If* $\Delta, \alpha, \Delta_1; X \vdash H \Leftarrow \mathsf{heap}\,[H]$, *then* $\Delta, \Delta_1'; X \vdash H' \Leftarrow \mathsf{heap}\,[H']$.
7. *If* $\Delta, \alpha, \Delta_1; X; \Gamma_1 \Longrightarrow \Gamma_2$, *and the proposition context* Γ_1' *and* Γ_2' *and are well-formed (i.e.,* $\Delta, \Delta_1' \vdash \Gamma_1'\ \mathsf{pctx}$, *and* $\Delta, \Delta_1' \vdash \Gamma_2'\ \mathsf{pctx}$*), then* $\Delta, \Delta_1'; X; \Gamma_1' \Longrightarrow \Gamma_2'$.

Proof: By straightforward induction on the structure of the given expressions, using the properties of composition of hereditary and monotype substitution to establish equality of the types when necessary. The essential property that simplifies the considerations is that monotype substitution is a total function, and that it cannot create redexes in the result term.

The only somewhat unusual cases arise in statement 2, when $N = K$ or $N = \mathsf{eta}_\alpha\ K$. In the first case, the result follows immediately by Lemma 11. In the second case, when $N = \mathsf{eta}_\alpha\ K$, we know by the typing derivation that $\Delta, \alpha, \Delta_1 \vdash K \Rightarrow \alpha\,[K]$, and by induction hypothesis on K, $\Delta, \Delta_1' \vdash K' \Rightarrow \tau\,[K']$. But then by the Identity principles (Lemma 13), $\Delta, \Delta_1' \vdash \mathsf{expand}_\tau(K') \Leftarrow \tau\,[\mathsf{expand}_\tau(K')]$. Because $[\tau/\alpha](\mathsf{eta}_\alpha\ K)$ equals $\mathsf{expand}_\tau(K')$, this is precisely what we needed prove. ■

The canonical substitution principle for terms and computations remains the same as in [26].

Lemma 17 (Canonical term substitution principles)
Suppose that $\Delta \vdash M \Leftarrow A\,[M]$, *and* $\vdash \Delta, x{:}A, \Delta_1\ \mathsf{ctx}$ *and that the context* $\Delta_1' = [M/x]_A(\Delta_1)$ *exists and is well-formed (i.e.* $\vdash \Delta, \Delta_1'\ \mathsf{ctx}$*). Then the following holds.*

1. *If* $\Delta, x{:}A, \Delta_1 \vdash K \Rightarrow B\,[K]$, *then* $[M/x]^k_A(K)$ *and* $B' = [M/x]^a_A(B)$ *exist and is well-formed (i.e.* $\Delta, \Delta_1' \vdash B' \Leftarrow \mathsf{type}\,[B']$*) and*
   (a) *if* $[M/x]^k_A(K) = K'$ *is an elim term, then* $\Delta, \Delta_1' \vdash K' \Rightarrow B'\,[K']$
   (b) *if* $[M/x]^k_A(K) = N' :: S$ *is an intro term, then* $\Delta, \Delta_1' \vdash N' \Leftarrow B'\,[N']$, *and* $S = B^-$.
2. *If* $\Delta, x{:}A, \Delta_1 \vdash N \Leftarrow B\,[N]$, *and the type* $B' = [M/x]^a_A(B)$ *exists and is well-formed (i.e.,* $\Delta, \Delta_1' \vdash B' \Leftarrow \mathsf{type}\,[B']$*), then* $\Delta, \Delta_1' \vdash [M/x]^m_A(N) \Leftarrow B'\,[[M/x]^m_A(N)]$.
3. *If* $\Delta, x{:}A, \Delta_1; P \vdash E \Leftarrow y{:}B.\ Q\,[E]$, *and* $y \notin \mathsf{FV}(M)$, *and the propositions* $P' = [M/x]^p_A(P)$ *and* $Q' = [M/x]^p_A(Q)$ *and the type* $B = [M/x]^a_A(B)$ *exist and are well-formed (i.e.,* $\Delta, \Delta_1'; \mathit{init}, \mathit{mem} \vdash P' \Leftarrow \mathsf{prop}\,[P']$, $\Delta, \Delta_1' \vdash B' \Leftarrow \mathsf{type}\,[B']$ *and* $\Delta, \Delta_1', y{:}B'; \mathit{init}, \mathit{mem} \vdash Q' \Leftarrow \mathsf{prop}\,[Q']$*), then* $\Delta, \Delta_1'; P' \vdash [M/x]^e_A(E) \Leftarrow y{:}B'.\ Q'\,[[M/x]^e_A(E)]$.
4. *If* $\Delta, x{:}A, \Delta_1 \vdash B \Leftarrow \mathsf{type}\,[B]$, *then* $\Delta, \Delta_1' \vdash [M/x]^a_A(B) \Leftarrow \mathsf{type}\,[[M/x]^a_A(B)]$.
5. *If* $\Delta, x{:}A, \Delta_1; X \vdash P \Leftarrow \mathsf{prop}\,[P]$, *then* $\Delta, \Delta_1'; \Psi \vdash [M/x]^p_A(P) \Leftarrow \mathsf{prop}\,[[M/x]^p_A(P)]$.
6. *If* $\Delta, x{:}A, \Delta_1; X \vdash H \Leftarrow \mathsf{heap}\,[H]$, *then* $\Delta, \Delta_1'; \Psi \vdash [M/x]^h_A(H) \Leftarrow \mathsf{heap}\,[[M/x]^h_A(H)]$.
7. *If* $\Delta, x{:}A, \Delta_1; X; \Gamma_1 \Longrightarrow \Gamma_2$, *and the proposition context* $\Gamma_1' = [M/x]_A(\Gamma_1)$ *and* $\Gamma_2' = [M/x]_A(\Gamma_2)$ *exist and are well-formed (i.e.,* $\Delta, \Delta_1' \vdash \Gamma_1'\ \mathsf{pctx}$, *and* $\Delta, \Delta_1' \vdash \Gamma_2'\ \mathsf{pctx}$*), then* $\Delta, \Delta_1'; \Psi; \Gamma_1' \Longrightarrow \Gamma_2'$.
8. *If* $\Delta; P \vdash E \Leftarrow x{:}A.\ Q\,[E]$, *and* $\Delta, x{:}A; Q \vdash F \Leftarrow y{:}B.\ R\,[F]$, *where* $x \notin \mathsf{FV}(B, R)$, *then* $\Delta; P \vdash \langle E/x\rangle_A(F) \Leftarrow y{:}B.\ R\,[\langle E/x\rangle_A(F)]$.

Proof: By nested induction, first on the structure of the shape of A, and then on the derivation of the first typing or sequent judgment in each case. We present only the proof of the statement 3, when $E = \mathsf{let\ dia}\ z = K\ \mathsf{in}\ F$ and $[M/x]K$ is an introduction term $\mathsf{dia}\ E_1$, as this is the most involved case. This case also differs from the analogous one in the earlier formulation of HTT, because of the small footprints. To abbreviate the notation, we write $(-)'$ instead of $[M/x]^*_A(-)$.

In this case, by the typing derivation of E, we know that $\Delta, x{:}A, \Delta_1 \vdash K \Rightarrow \Psi.X.\{R_1\}z{:}C\{R_2\}\,[K]$, and $\Delta, x{:}A, \Delta_1; \mathsf{init}, \mathsf{mem}; P \Longrightarrow \exists\Psi.X.(R_1 * \top)$, and $\Delta, x{:}A, \Delta_1, z{:}C; P \circ \forall\Psi.X.(R_1 \multimap R_2) \vdash F \Leftarrow y{:}B.\ Q\,[F]$ and $\Delta, \Delta_1'; \mathsf{this}(\mathsf{init}) \wedge \exists\Psi'.X.(R_1' * \top) \vdash E_1 \Leftarrow z{:}C'.\ \forall\Psi'.X.R_1' \multimap R_2'\,[E_1]$. We also know by Theorem 1 that $(\Psi.X.\{R_1\}z{:}C\{R_2\})^- \leq A^-$, and in particular $C^- < A^-$. And, of course, by definition $E' = \langle E_1/z\rangle_{C^-}(F')$.

From the typing of E_1, by preservation of history (Lemma 14), $\Delta, \Delta_1'; P' \circ (\mathsf{this}(\mathsf{init}) \wedge \exists\Psi'.X.(R_1' * \top)) \vdash E_1 \Leftarrow z{:}C'.\ P' \circ \forall\Psi'.X.(R_1' \multimap R_2')\,[E_1]$. From the typing of F, by induction hypothesis, $\Delta, \Delta_1', z{:}C'; P' \circ \forall\Psi'.X.(R_1' \multimap R_2') \vdash F' \Leftarrow y{:}B'.\ Q'\,[F']$. By induction hypothesis on $C'^- = C^- < A^-$, and from the above

two judgments, by monadically substituting E_1 for z in F', we obtain $\Delta, \Delta_1'; P' \circ (\mathsf{this}(\mathsf{init}) \wedge \exists \Psi'.X.(R_1' * \top)) \vdash E' \Leftarrow y{:}B'.\ Q'\,[E']$.

Finally, by induction hypothesis on the derivation of the sequent $P \Longrightarrow \exists \Psi.X.(R_1 * \top)$ we obtain $P' \Longrightarrow \exists \Psi'.X.(R_1' * \top)$, and therefore also $P' \Longrightarrow P' \circ (\mathsf{this}(\mathsf{init}) \wedge \exists \Psi'.X.(R_1' * \top))$. Now we can apply strengthening of the precedent (Lemma 14) to derive the required $\Delta, \Delta_1'; P' \vdash E' \Leftarrow y{:}B'.\ Q'\,[E']$. ■

The following lemma shows that canonical forms of expressions obtained as output of the typing judgments, are indeed canonical in the sense that they are well-typed and invariant under further normalization. In other words, the process of obtaining canonical forms is an involution. The lemma will be important subsequently in the proof of the substitution principles. It will establish that the various intermediate expressions produced by the typing are canonical, and thus subject to the canonical substitution principles from Lemma 17.

Lemma 18 (Involution of canonical forms)

1. *If* $\Delta \vdash K \Rightarrow A\,[K']$, *and* K' *is an elim term, then* $\Delta \vdash K' \Rightarrow A\,[K']$.
2. *If* $\Delta \vdash K \Rightarrow A\,[N']$ *and* N' *is an intro term, then* $\Delta \vdash N' \Leftarrow A\,[N']$.
3. *If* $\Delta \vdash N \Leftarrow A\,[N']$, *then* $\Delta \vdash N' \Leftarrow A\,[N']$.
4. *If* $\Delta; P \vdash E \Leftarrow x{:}A.\ Q\,[E']$, *then* $\Delta; P \vdash E' \Leftarrow x{:}A.\ Q\,[E']$.
5. *If* $\Delta \vdash A \Leftarrow \mathsf{type}\,[A']$, *then* $\Delta \vdash A' \Leftarrow \mathsf{type}\,[A']$.
6. *If* $\Delta; \Psi \vdash P \Leftarrow \mathsf{prop}\,[P']$, *then* $\Delta; \Psi \vdash P' \Leftarrow \mathsf{prop}\,[P']$.
7. *If* $\Delta; \Psi \vdash H \Leftarrow \mathsf{heap}\,[H']$, *then* $\Delta; \Psi \vdash H' \Leftarrow \mathsf{heap}\,[H']$.

Proof: By straightforward simultaneous induction on the structure of the given typing derivations. We discuss here the statement 3. The cases for the introduction forms are trivial, and so is the case for the eta rule. Notice that in the case of the eta rule, by the form of the rule, we already know that $N = N'$, so there is nothing to prove.

The only remaining case is when the last rule in the judgment derivation is $\Rightarrow\Leftarrow$, and correspondingly, we have $N = K$ is an elimination term.

In this case, by the typing derivation, we know that $\Delta \vdash K \Rightarrow B\,[M']$ and $A = B$ and $N' = \mathsf{expand}_A(M')$. Now, if M' is an introduction term, then $N' = M'$ and the result immediately follows by induction hypothesis 2. On the other hand, if M' is an elimination term, then by induction hypothesis 1, $\Delta \vdash M' \Rightarrow A\,[M']$, and then by the identity principles (Lemma 13), $\Delta \vdash \mathsf{expand}_A(M') \Leftarrow A\,[\mathsf{expand}_A(M')]$.

■

Finally, we can state and prove the substitution principles on general, rather than only on canonical terms. In this lemma, we avoid the statement about the existence and well-formedness of the various expressions, because all of these can be shown satisfied by the canonical term substitution principles. We also remind the reader that the general forms do not contain the term $\mathsf{eta}_\alpha\ K$, which can only appear in the canonical fragment.

Lemma 19 (General monotype substitution principles)
Suppose that $\Delta \vdash \tau \Leftarrow \mathsf{mono}\,[\tau']$. *Then the following holds.*

1. *If* $\Delta, \alpha, \Delta_1 \vdash K \Rightarrow B\,[N']$, *then* $\Delta, [\tau'/\alpha](\Delta_1) \vdash [\tau/\alpha]K \Rightarrow [\tau'/\alpha](B)\,[[\tau'/\alpha](N')]$.
2. *If* $\Delta, \alpha, \Delta_1 \vdash N \Leftarrow B\,[N']$, *then* $\Delta, [\tau'/\alpha](\Delta_1) \vdash [\tau/\alpha]N \Leftarrow [\tau'/\alpha](B)\,[[\tau'/\alpha](N')]$.
3. *If* $\Delta, \alpha, \Delta_1; P \vdash E \Leftarrow y{:}B.\ Q\,[E']$, *and* $y \notin \mathsf{FV}(M)$, *then* $\Delta, [\tau'/\alpha](\Delta_1); [\tau'/\alpha](P) \vdash [\tau/\alpha]E \Leftarrow y{:}[\tau'/\alpha](B).\ [\tau'/\alpha](Q)\,[[\tau'/\alpha](E')]$.

4. *If* $\Delta, \alpha, \Delta_1 \vdash B \Leftarrow \mathsf{type}\,[B']$, *then* $\Delta, [\tau'/\alpha](\Delta_1) \vdash [\tau/\alpha]B \Leftarrow \mathsf{type}\,[[\tau'/\alpha](B')]$.

5. *If* $\Delta, \alpha, \Delta_1; X \vdash P \Leftarrow \mathsf{prop}\,[P]$, *then* $\Delta, [\tau'/\alpha](\Delta_1); X \vdash [\tau/\alpha]P \Leftarrow \mathsf{prop}\,[[\tau'/\alpha](P')]$.

6. *If* $\Delta, \alpha, \Delta_1; X \vdash H \Leftarrow \mathsf{heap}\,[H]$, *then* $\Delta, [\tau'/\alpha](\Delta_1); X \vdash [\tau/\alpha]H \Leftarrow \mathsf{heap}\,[[\tau'/\alpha](H')]$.

Proof: Straightforward by simultaneous induction of the principal derivations. ■

The general term substitution principle is the same as in [26].

Lemma 20 (General term substitution principles)
Suppose that $\Delta \vdash A \Leftarrow \mathsf{type}\,[A']$ *and* $\Delta \vdash M \Leftarrow A'\,[M']$. *Then the following holds.*

1. *If* $\Delta, x{:}A', \Delta_1 \vdash K \Rightarrow B\,[N']$, *then* $\Delta, [M'/x]_A(\Delta_1) \vdash [M : A/x]K \Rightarrow [M'/x]^a_A(B)\,[[M'/x]^m_A(N')]$.

2. *If* $\Delta, x{:}A', \Delta_1 \vdash N \Leftarrow B\,[N']$, *then* $\Delta, [M'/x]_A(\Delta_1) \vdash [M : A/x]N \Leftarrow [M'/x]^a_A(B)\,[[M'/x]^m_A(N')]$.

3. *If* $\Delta, x{:}A', \Delta_1; P \vdash E \Leftarrow y{:}B.\ Q\,[E']$, *and* $y \notin \mathsf{FV}(M)$, *then* $\Delta, [M'/x]_A(\Delta_1); [M'/x]^p_A(P) \vdash [M : A/x]E \Leftarrow y{:}[M'/x]^a_A(B).\ [M/'x]^p_A(Q)\,[[M'/x]^e_A(E')]$.

4. *If* $\Delta, x{:}A', \Delta_1 \vdash B \Leftarrow \mathsf{type}\,[B']$, *then* $\Delta, [M'/x]_A(\Delta_1) \vdash [M : A/x]B \Leftarrow \mathsf{type}\,[[M'/x]^a_A(B')]$.

5. *If* $\Delta, x{:}A', \Delta_1; \Psi \vdash P \Leftarrow \mathsf{prop}\,[P']$, *then* $\Delta, [M'/x]_A(\Delta_1) \vdash [M : A/x]P \Leftarrow \mathsf{prop}\,[[M'/x]^p_A(P)]$.

6. *If* $\Delta, x{:}A', \Delta_1; \Psi \vdash H \Leftarrow \mathsf{heap}\,[H']$, *then* $\Delta, [M'/x]_A(\Delta_1) \vdash [M : A/x]H \Leftarrow \mathsf{heap}\,[[M'/x]^h_A(H)]$.

7. *If* $\Delta; P \vdash E \Leftarrow x{:}A'.\ Q\,[E']$ *and* $\Delta, x{:}A'; Q \vdash F \Leftarrow y{:}B.\ R\,[F']$, *where* $x \notin \mathsf{FV}(B, R)$, *then* $\Delta; P \vdash \langle E/x : A\rangle F \Leftarrow y{:}B.\ R\,[\langle E'/x\rangle_A(F')]$.

Proof: The proofs do not change significantly. The new cases involving the polymorphic abstraction are straightforward. The cases involving computations are different due to the small footprints, but the approach to proving them is largely analogous to the corresponding cases in the canonical fragment.

It is somewhat interesting that the case $N = \mathsf{eta}_\alpha\ K$ does not arise here, unlike in the canonical version of the theorem, because N is a general, rather than canonical term, and thus may not contain the constructor eta. ■

6 Operational semantics

In this section we define the call-by-value, left-to-right structured operational semantics for HTT. This provides the constructive interpretation of HTT proof terms, and shows that they can be viewed as programs.

Our approach is completely analogous with the previous proposal in [26], and the only addition in this section are the new and changed cases of the Preservation and Progress theorems, which arise due to the extensions with polymorphism and small footprints. We also have an extended definition of the concept of heap soundness (see below), which is needed to account for strong update and deallocation (neither of which was present in the old proposal).

We note that Preservation and Progress theorems together establish that HTT is sound with respect to evaluation. The Progress theorem is proved under the assumption that HTT assertion logic is Heap Sound, but we establish this Heap Soundness subsequently in Section 7, using denotational semantics.

The syntactic domains used in the operational semantics are the following.

Values	v, l	$::=$	$(\,) \mid \lambda x.\ M \mid \Lambda\alpha.\ M \mid \mathsf{dia}\ E \mid \mathsf{true} \mid \mathsf{false} \mid \mathsf{z} \mid \mathsf{s}\ v$
Value heaps	χ	$::=$	$\cdot \mid \chi, l \mapsto_\tau v$
Continuations	κ	$::=$	$\cdot \mid x{:}A.\ E; \kappa$
Control expressions	ρ	$::=$	$\kappa \triangleright E$
Abstract machines	μ	$::=$	$\chi, \kappa \triangleright E$

Values. The definition of values is standard, and includes the intro forms for each of the composite type constructors. As usual, we will admit function values whose bodies are unreduced. We use v to range over values, and l to range over integers when they are used as pointers into the heap.

Value heaps. Value heaps are assignments from integers to values, where each assignment is indexed by a type. They are a run-time concept, unlike heaps from Section 2 which are expressions used for reasoning in the assertion logic. That the two notions actually correspond to each other is the statement of our definition of heap soundness, that will be given later in this section.

A value heap χ is well-formed if the judgment $\vdash \chi : \mathsf{heapval}$ is satisfied. The rules of this judgment are as follows.

$$\frac{}{\vdash \cdot : \mathsf{heapval}} \qquad \frac{\vdash \chi : \mathsf{heapval} \quad \vdash \tau \Leftarrow \mathsf{mono}\,[\tau] \quad \vdash l \Leftarrow \mathsf{nat}\,[l] \quad \vdash v \Leftarrow \tau\,[M] \quad l \notin \mathsf{dom}(\chi)}{\vdash (\chi, l \mapsto_\tau v) : \mathsf{heapval}}$$

Notice how the judgment requires that the index monotype τ is canonical, and that the location l is a canonical integer with no free variables – and is thus a numeral, rather than an arithmetic expression, as shown in Lemma 9 – so that l can be checked for membership in $\mathsf{dom}(\chi)$. Value heaps are considered equal up to the reordering of their assignments.

For the purposes of the Preservation and Progress theorems, we will need to convert a value heap into a heap canonical form, so we introduce the following conversion function.

$$\begin{array}{rcll} [\![\cdot]\!] & = & \mathsf{empty} & \\ [\![\chi, l \mapsto_\tau v]\!] & = & \mathsf{upd}_\tau([\![\chi]\!], l, M), & \text{where } \cdot \vdash v \Leftarrow \tau\,[M] \end{array}$$

We will also write $\Delta; \chi \vdash P$ as short for $\Delta; \mathsf{mem}; \mathsf{this}([\![\chi]\!]) \Longrightarrow P$. This judgment will essentially establish that the proposition P holds of the value heap χ. Of course, here we assumed that $\vdash \Delta\ \mathsf{ctx}$, and $\vdash \chi : \mathsf{heapval}$ and $\Delta; \mathsf{mem} \vdash P \Leftarrow \mathsf{prop}\,[P]$.

Continuations and control expressions. 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. In fact, the control expression $x_1.E_1; \ldots; x_n.E_n \triangleright E$ (we omit the types for the moment) is just a different syntactic way of writing the computation

$$\mathsf{let\ dia}\ x_n = \mathsf{dia}\ (\mathsf{let\ dia}\ x_{n-1} = \cdots \mathsf{dia}\ (\mathsf{let\ dia}\ x_1 = \mathsf{dia}\ E\ \mathsf{in}\ E_1) \cdots\ \mathsf{in}\ E_{n-1})\ \mathsf{in}\ E_n$$

This expression has a special place in the operational semantics, because its call-by-value evaluation cannot be described simply by means of the monadic substitution used in the equational theory of HTT. In the equational theory of HTT, the composition of computations E and $x.\ F$ is defined by the monadic substitution $\langle E/x : A\rangle F$. But in a call-by-value semantics, the composition must evaluate E before substituting the value into F. This is formally described by creating the control expression $x.\ F; \cdot \triangleright E$; or in other words, first push $x.\ F$ onto the continuation, and proceed to evaluate E.

The described distinction between the equational theory and operational semantics of HTT is analogous to the well-known difference in the simple lambda calculus between the rules for beta reduction and the beta-value reduction which is used in the call-by-value operational semantics. However, because the terms in HTT are pure, call-by-value semantics is adequate for the equational theory, as we prove in the Preservation theorem below.

We require a typing judgment for control expressions. It has the form $\Delta; P \vdash \kappa \triangleright E \Leftarrow x{:}A.\ Q$, and, not surprisingly, 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. The judgment does not compute the canonical form of $\kappa \triangleright E$, because control expressions are used for purposes of operational semantics, and not for equational reasoning. The judgment assumes the that Δ, P, A and Q are canonical. Moreover, P and Q are allowed to depend on a heap variable mem, but unlike in the computation judgments, no dependence on init is allowed. In the computation judgments, the heap variable init is used in the Hoare triples to to denote the yet unknown heap in which a suspended computation can be executed. But when working with control expressions, we do not have a constructor for suspension, so init is not needed.

$$\frac{\Delta; P \vdash E \Leftarrow x{:}A.\ Q\,[E']}{\Delta; P \vdash \cdot \triangleright E \Leftarrow x{:}A.\ Q}$$

$$\frac{\Delta \vdash B \Leftarrow \mathsf{type}\,[B'] \quad \Delta; P \vdash \kappa \triangleright E \Leftarrow y{:}B'.\ R \quad \Delta, y{:}B'; R \vdash F \Leftarrow x{:}A.\ Q\,[F'] \quad y \notin \mathsf{FV}(A, Q)}{\Delta; P \vdash \kappa; (y{:}B.\ F; \cdot) \triangleright E \Leftarrow x{:}A.\ Q}$$

The control expression with the empty continuation $\cdot \triangleright E$ is well-typed if E is well-typed as a computation. If the continuation is not empty, we can split it into its *last* computation $y{:}B.F$, and use the variable κ to name the continuation consisting of all the preceding computations. Then control expression is well-typed if F is well-typed under the precondition R where R is some postcondition for $\kappa \triangleright E$.

We next prove a lemma that will allow us to replace the computation E in the control expression $\kappa \triangleright E$ with another computation F, as long as E and F have the same postconditions, and thus both provide the same precondition for the execution of κ. The lemma is slightly more general, and instead of a computation F it considers a control expression $\kappa_1 \triangleright E_1$. This is not problematic, because of the close correspondence between computations and control expressions.

Lemma 21 (Replacement)

1. *If* $\Delta; P \vdash \kappa \triangleright E \Leftarrow x{:}A.\ Q$, *then* $\Delta; P \vdash E \Leftarrow y{:}B.\ R\,[E']$ *for some* y, B, R, E' *and if* $\Delta_1; P_1 \vdash \kappa_1 \triangleright E_1 \Leftarrow y{:}B.\ R$, *for some* Δ_1 *extending* Δ, *then* $\Delta_1; P_1 \vdash \kappa_1; \kappa \triangleright E_1 \Leftarrow x{:}A.\ Q$.

2. *If* $\Delta; P \vdash y{:}B.\ F; \kappa \triangleright E \Leftarrow x{:}A.\ Q$, *then* $\Delta; P \vdash \kappa \triangleright \langle E/y : B\rangle F \Leftarrow x{:}A.\ Q$.

Proof: By straightforward induction on the structure of κ. The proof is completely analogous to the one we presented in [26], so we omit it here. ■

Abstract machines. An abstract machine μ is a pair of a value heap χ and a control expression $\kappa \triangleright E$. The control expression is evaluated against the heap, to eventually produce a result and possibly change the starting heap.

The type information for an abstract machine μ specifies the type A of the return result and the description Q of the ending heap of the machine. The judgment has the form $\chi, \kappa \triangleright E \Leftarrow x{:}A.\ Q$. We assume the usual conventions about canonicity of A and Q.

By definition, the judgment $\vdash \chi, \kappa \triangleright E \Leftarrow x{:}A.\ Q$ is equivalent to $\cdot; P \vdash \kappa \triangleright E \Leftarrow x{:}A.\ Q$, where $P = \mathsf{this}([\![\chi]\!])$. In other words, we first convert the heap χ into a canonical proposition P which uniquely defines χ (up to the normalization of values stored in the heap), 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 inference rules of the evaluation judgments are straightforward, and completely analogous to the one we presented in [26]. For completeness, we repeat the rules here, with the addition of the new terms for type polymorphism.

We start by presenting the rules for evaluating elimination terms.

$$\frac{K \hookrightarrow_k K'}{K\ N \hookrightarrow_k K'\ N} \qquad \frac{N \hookrightarrow_m N'}{(v : A)\ N \hookrightarrow_k (v : A)\ N'}$$

$$\frac{K \hookrightarrow_k K'}{K\ \tau \hookrightarrow_k K'\ \tau}$$

$$\frac{}{(\lambda x.\ M : \Pi x{:}A_1.\ A_2)\ v \hookrightarrow_k [v : A_1/x]M : [v : A_1/x]A_2}$$

$$\frac{}{(\Lambda\alpha.\ M : \forall\alpha.\ A)\ \tau \hookrightarrow_k [\tau/\alpha]M : [\tau/\alpha]A}$$

$$\frac{M \hookrightarrow_m M'}{M : A \hookrightarrow_k M' : A}$$

Evaluation of introduction terms follows. If the introduction term is obtained by coercion from an elimination term, we invoke the judgment for elimination terms. If the returned result is of the form $v : A$, we remove the type annotation. This prevents accumulation of type annotations, as in $v : A_1 : \cdots : A_n$.

$$\frac{K \hookrightarrow_k K' \quad K' \neq v : A}{K \hookrightarrow_m K'} \qquad \frac{K \hookrightarrow_k v : A}{K \hookrightarrow_m v}$$

Of course, we also need evaluation rules for primitive operations.

$$\frac{M \hookrightarrow_m M'}{\mathsf{s}\ M \hookrightarrow_m \mathsf{s}\ M'}$$

$$\frac{M \hookrightarrow_m M'}{M + N \hookrightarrow_m M' + N} \qquad \frac{N \hookrightarrow_m N'}{v + N \hookrightarrow_m v + N'} \qquad \frac{}{v_1 + v_2 \hookrightarrow \mathsf{plus}(v_1, v_2)}$$

$$\frac{M \hookrightarrow_m M'}{M \times N \hookrightarrow_m M' \times N} \qquad \frac{N \hookrightarrow_m N'}{v \times N \hookrightarrow_m v \times N'} \qquad \frac{}{v_1 \times v_2 \hookrightarrow \mathsf{times}(v_1, v_2)}$$

$$\frac{M \hookrightarrow_m M'}{\mathsf{eq}(M, N) \hookrightarrow_m \mathsf{eq}(M', N)} \qquad \frac{N \hookrightarrow_m N'}{\mathsf{eq}(v, N) \hookrightarrow_m \mathsf{eq}(v, N')}$$

$$\frac{}{\mathsf{eq}(v_1, v_2) \hookrightarrow_m \mathsf{equals}(v_1, v_2)}$$

In the evaluation of abstract machines, we occasionally 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 outcome of the evaluation, however, does not depend on type information, and the Progress theorem proved below shows that type checking is unnecessary (i.e., it always succeeds) if the evaluation starts with well-typed

abstract machines.

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

$$\frac{K \hookrightarrow_k K'}{\chi, \kappa \triangleright \mathsf{let\ dia}\ x = K\ \mathsf{in}\ E \hookrightarrow_e \chi, \kappa \triangleright \mathsf{let\ dia}\ x = K'\ \mathsf{in}\ E}$$

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

$$\frac{M \hookrightarrow_m M'}{\chi, \kappa \triangleright x = \mathsf{alloc}_\tau(M); E \hookrightarrow_e \chi, \kappa \triangleright x = \mathsf{alloc}_\tau(M'); E} \qquad \frac{\cdot \vdash \tau \Leftarrow \mathsf{mono}\,[\tau'] \quad l \notin \mathsf{dom}(\chi)}{\chi, \kappa \triangleright x = \mathsf{alloc}_\tau(v); E \hookrightarrow_e (\chi, l \mapsto_{\tau'} v), \kappa \triangleright [l{:}\mathsf{nat}/x]E}$$

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

$$\frac{M \hookrightarrow_m M'}{\chi, \kappa \triangleright [M]_\tau = N; E \hookrightarrow_e \chi, \kappa \triangleright [M']_\tau = N; E} \qquad \frac{N \hookrightarrow_m N'}{\chi, \kappa \triangleright [v]_\tau = N; E \hookrightarrow_e \chi, \kappa \triangleright [v]_\tau = N'; E}$$

$$\frac{\cdot \vdash \tau \Leftarrow \mathsf{mono}\,[\tau']}{(\chi_1, l \mapsto_\sigma v', \chi_2), \kappa \triangleright [l]_\tau = v; E \hookrightarrow_e (\chi_1, l \mapsto_{\tau'} v, \chi_2), \kappa \triangleright E}$$

$$\frac{M \hookrightarrow_m M'}{\chi, \kappa \triangleright \mathsf{dealloc}(M); E \hookrightarrow_e \chi, \kappa \triangleright \mathsf{dealloc}(M'); E} \qquad \frac{}{(\chi_1, l \mapsto_\sigma v, \chi_2), \kappa \triangleright \mathsf{dealloc}(l); E \hookrightarrow_e (\chi_1, \chi_2), \kappa \triangleright E}$$

$$\frac{M \hookrightarrow_m M'}{\chi, \kappa \triangleright x = \mathsf{if}_A(M, E_1, E_2); E \hookrightarrow_e \chi, \kappa \triangleright x = \mathsf{if}_A(M', E_1, E_2); E}$$

$$\frac{}{\chi, \kappa \triangleright x = \mathsf{if}_A(\mathsf{true}, E_1, E_2); E \hookrightarrow_e \chi, x{:}A.\ E; \kappa \triangleright E_1} \qquad \frac{}{\chi, \kappa \triangleright x = \mathsf{if}_A(\mathsf{false}, E_1, E_2); E \hookrightarrow_e \chi, x{:}A.\ E; \kappa \triangleright E_2}$$

$$\frac{M \hookrightarrow_m M'}{\chi, \kappa \triangleright y = \mathsf{fix}_A(M, f.x.E); F \hookrightarrow_e \chi, \kappa \triangleright y = \mathsf{fix}_A(M', f.x.E); F}$$

$$\frac{N = \lambda z.\ \mathsf{dia}\ (y = \mathsf{fix}_{\Pi x:A.\Psi.X.\{R_1\}y:B\{R_2\}}(z, f.x.E); y)}{\chi, \kappa \triangleright y = \mathsf{fix}_{\Pi x:A.\Psi.X.\{R_1\}y:B\{R_2\}}(v, f.x.E); F \hookrightarrow_e \chi, (y{:}[v : A/x]B.\ F; \kappa) \triangleright [v : A/x, N : \Pi x{:}A.\Psi.X.\{R_1\}y{:}B\{R_2\}/f]E}$$

The preservation theorem 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. In other words, Preservation also shows the adequacy of evaluation for canonical forms. The critical case here are expressions involving addition and multiplication of natural numbers. However, because canonical forms for these operations perform the simplifications described in Section 3, natural number values correspond to closed canonical forms (Lemma 9).

Theorem 22 (Preservation)

1. *if* $K_0 \hookrightarrow_k K_1$ *and* $\cdot \vdash K_0 \Rightarrow A\,[N']$, *then* $\cdot \vdash K_1 \Rightarrow A\,[N']$.
2. *if* $M_0 \hookrightarrow_m M_1$ *and* $\cdot \vdash M_0 \Leftarrow A\,[M']$, *then* $\cdot \vdash M_1 \Leftarrow A\,[M']$.
3. *if* $\mu_0 \hookrightarrow_e \mu_1$ *and* $\vdash \mu_0 \Leftarrow x{:}A.\ Q$, *then* $\vdash \mu_1 \Leftarrow x{:}A.\ Q$.

Proof: The first two statements are proved by simultaneous induction on the evaluation judgment, using inversion on the typing derivation, and substitution principles. The third statement is proved by case analysis on the evaluation judgment, using the first two statements and the replacement lemma (Lemma 21). The cases can roughly be split into the ones in which the command changes the continuation of the abstract machine, and the ones in which the command changes the heap of the abstract machine. For the first category, we present the case of $\mathsf{let\ dia}$ (the case of fixpoints is similar, but a bit more involved). For the second category, we present the case for alloc.

First the case $\mu_0 = \chi_0, \kappa_0 \triangleright \mathsf{let\ dia}\ y = \mathsf{dia}\ F : \Psi.\{R_1\}y{:}B\{R_2\}\ \mathsf{in}\ E$. In this case, $\mu_1 = \chi_0, (y{:}B.E; \kappa_0) \triangleright F$. Let $P = \mathsf{this}([\![\chi_0]\!])$. From the typing of μ_0, we know $P \vdash \kappa_0 \triangleright \mathsf{let\ dia}\ y = \mathsf{dia}\ F : \Psi.\{R_1\}y{:}B\{R_2\}\ \mathsf{in}\ E \Leftarrow x{:}A.\ Q$. By the replacement lemma, $P \vdash \mathsf{let\ dia}\ y = \mathsf{dia}\ F : \Psi.\{R_1\}y{:}B\{R_2\}\ \mathsf{in}\ E \Leftarrow z{:}C.\ S$, for some type C and proposition S (we omit normal forms of computations here, as they are not needed). Therefore $\cdot \vdash B \Leftarrow \mathsf{type}\,[B']$ and $\Psi; \mathsf{mem} \vdash R_1 \Leftarrow \mathsf{prop}\,[R_1']$ and $\Psi; \mathsf{mem} \vdash R_2 \Leftarrow \mathsf{prop}\,[R_2']$ and

1. $\mathsf{init}, \mathsf{mem}; P \Longrightarrow \exists\Psi.X.(R_1' * \top)$
2. $\mathsf{this}(\mathsf{init}) \wedge \exists\Psi.X.(R_1' * \top) \vdash F \Leftarrow y{:}B'.\ \forall\Psi.X.(R_1' \multimap R_2')$
3. $y{:}B'; P \circ \forall\Psi.X.R_1' \multimap R_2' \vdash E \Leftarrow z{:}C.\ S$.

It now suffices to show that $P \vdash y{:}B.E \triangleright F \Leftarrow z{:}C.\ S$, and then the result follows by replacement. To establish the last judgment, we first observe that from (2), by Preservation of History (Lemma 14), $P \circ \mathsf{this}(\mathsf{init}) \wedge \exists\Psi.X.(R_1' * \top) \vdash F \Leftarrow y{:}B'.\ (P \circ \forall\Psi.X.R_1' \multimap R_2')$. Then, because (1) implies $P \Longrightarrow P \circ \mathsf{this}(\mathsf{init}) \wedge \exists\Psi.X.(R_1' * \top)$, we can strengthen the precedent and obtain $P \vdash F \Leftarrow y{:}B'.\ (P \circ \forall\Psi.X.R_1' \multimap R_2')$. From the last derivation, and (3), by the typing rules for control expressions, we obtain $P \vdash y{:}B.E \triangleright F \Leftarrow z{:}C.\ S$, which proves the case.

Let us now consider the case for alloc, i.e. when $\mu_0 = \chi_0, \kappa_0 \triangleright y = \mathsf{alloc}_\tau(v); E$. In this case, $\mu_1 = (\chi_0, l \mapsto_\tau v), \kappa \triangleright [l : \mathsf{nat}/y]E$ (here we assume that τ is canonical for simplicity). Let $P = \mathsf{this}([\![\chi_0]\!])$. From the typing of μ_0, we know that $P \vdash \kappa_0 \triangleright y = \mathsf{alloc}_\tau(v); E \Leftarrow x{:}A.\ Q$. By replacement, $P \vdash y = \mathsf{alloc}_\tau(v); E \Leftarrow z{:}C.\ S$ for some z, C, S. Let $\vdash v \Leftarrow \tau\,[N']$. By the typing rules for alloc, $y{:}\mathsf{nat}; P * y \mapsto_\tau N' \vdash E \Leftarrow z{:}C.\ S$. Thus, for any numeral l, and in particular for any numeral $l \notin \mathsf{dom}(\chi_0)$, we have $P * l \mapsto_\tau N' \vdash [l : \mathsf{nat}/y]E \Leftarrow z{:}C.\ S$.

To establish the typing for μ_1, it suffices to show that $\mathsf{this}([\![\chi_0, l \mapsto_\tau v]\!]) \vdash [l : \mathsf{nat}/y]E \Leftarrow z{:}C.\ S$. But this clearly holds by strentghtening precedent, as $\mathsf{this}([\![\chi_0, l \mapsto_\tau v]\!]) \Longrightarrow P * (l \mapsto_\tau N')$. ■

The last theorem in this section is the Progress theorem. Progress states that the evaluation of well-typed expressions cannot get stuck. In this sense, it establishes the *soundness* of typing with respect to evaluation. But before we can state and prove the progress theorem, we need to define the property of the assertion logic which we call *heap soundness*.

Definition 23 (Heap soundness)
The assertion logic of HTT is heap sound iff for every value heap χ,

1. *the existence of a derivation for the sequent $\cdot$; mem; $\mathsf{this}([\![\chi]\!]) \Longrightarrow l \hookrightarrow_\tau -$ implies that $l \mapsto_\tau v \in \chi$, for some value v, and*
2. *the existence of a derivation for the sequent $\cdot$; mem; $\mathsf{this}([\![\chi]\!]) \Longrightarrow l \hookrightarrow -$ implies that $l \mapsto_\tau v \in \chi$ for some monotype τ and a value v.*

The clauses of the definition of heap soundness correspond to the side conditions that need to be derived in the typing rules for the primitive commands of lookup, update and deallocation. Heap soundness essentially shows that the assertion logic soundly reasons about value heaps, so that facts established in the assertion logic will be true during evaluation. If the assertion logic proves that $l \hookrightarrow_\tau -$, then the evaluation will be able to associate a value v with this location, which is needed, for example, in the evaluation rule for lookup. If the assertion logic proves that $l \hookrightarrow -$, then the evaluation will be able to associate a monotype τ and a value $v{:}\tau$, which is needed in the evaluation rules for update and deallocation.

We now state the Progress theorem, which can be seen as a statement of soundness of the type system of HTT with respect to evaluation, relative to the heap soundness of the assertion logic. Heap soundness is established in Section 7.

Theorem 24 (Progress)
Suppose that the assertion logic of HTT is heap sound. Then the following holds.

1. *If* $\cdot \vdash K_0 \Rightarrow A\,[N']$, *then either* $K_0 = v : A$ *or* $K_0 \hookrightarrow_k K_1$, *for some* K_1.
2. *If* $\cdot \vdash M_0 \Leftarrow A\,[M']$, *then either* $M_0 = v$ *or* $M_0 \hookrightarrow_m M_1$, *for some* M_1.
3. *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.

Proof: The proofs are by straightforward case analysis on the involved expressions, employing inversion on the typing derivations, using heap soundness in the cases of third statement involving the primitive effectful commands for allocation, lookup and update. ■

7 Heap Soundness

In this section we prove that the assertion logic of HTT is heap sound. We do so by means of a simple denotational semantics of HTT.

Let **pCpo** be the category of ω-complete partially ordered sets (partially ordered sets such that every ω-chain has a least upper bound) and partial continuous functions. Note that the objects do not necessarily have a least element. For a partial continuous function f, write $f(a)\downarrow$ for "$f(a)$ is defined" and write $f(a)\uparrow$ for "$f(a)$ is undefined." For cpo's X and Y, we write $X \rightharpoonup Y$ for the set of partial continous functions from X to Y and $X \to Y$ for the set of (total) continous functions from X to Y.

Let MonoTypes denote the set of mono types of HTT.

Let N denote the discrete cpo of natural numbers, let B denote the discrete cpo of booleans with elements *true* and *false*, and let 1 denote the one-element cpo with element $*$. Finally, let *Loc* be a copy of N. Recall that **pCpo** is bilimit compact and complete. Hence there is a canonical solution to the following recursive domain equation:

$$\begin{array}{lcl} V & \cong & 1 + N + B + (V \to V) + (H \rightharpoonup (V \times H)) + (\Pi_{A \in \mathrm{MonoTypes}}\, V) \\ H & = & \Sigma_{L \in P_{fin}(Loc)}(L \to V), \end{array}$$

where the ordering of $\Sigma_{L \in P_{fin}(Loc)}(L \to V)$ only relates records (heaps) with equal domain; two records with equal domain are ordered pointwise. We write i for the isomorphism $V \to 1 + N + B + (V \to V) + (H \rightharpoonup (V \times H))$ and i^{-1} for its inverse. We write κ for the coproduct injections of 1, N, ..., into the sum $1 + N + B + (V \to V) + (H \rightharpoonup (V \times H)$ (i.e., we do not distinguish notationally between the five different coproduct injections).

We write $\{\}$ for the empty heap $\kappa_\emptyset(\emptyset) \in H$. Further, we abbreviate the update of a heap $h = \kappa_L(h')$ with m mapped to v, which formally is defined by

$$\begin{cases} \kappa_L(h'[m \mapsto v]) & \text{if } m \in L, \\ \kappa_{L \cup \{m\}}(h'[m \mapsto v]) & \text{otherwise,} \end{cases}$$

to simply $h[m \mapsto v]$. Note that the update operation is indeed continuous. For a heap $h = \kappa_L(h') \in H$ and a location l, we write $l \in \mathsf{dom}(h)$ for $l \in L$, and we write $h(l)$ for the value $h'(l)$ (assuming that $l \in L$). Further, for a heap $h = \kappa_L(h') \in H$ we write "choose $x \notin \mathsf{dom}(h)$" to mean that x should be an element of *Loc* not in L (such a number always exists since L is finite).

We write π, possibly with subscripts, for projections out of products of cpo's (the subscript will indicate which projection we are referring to). When presenting the denotational semantics below, we often omit

the isomorphisms i, and i^{-1}, and injections κ. Further, we use a (semantic) strict let (here s and s' are mathematical expressions):

$$\mathsf{let}\ (v,h) = s \mathsf{in}\ s' \equiv \begin{cases} \text{undefined} & \text{if } s \text{ is undefined,} \\ (\lambda(v,h).s')(s) & \text{otherwise.} \end{cases}$$

In the semantics we do not disinguish between type errors, exceptional errors (dereferencing null-pointers), or non-termination. It is straightforward to adapt the semantics to do so, but we do not need it for showing heap soundness.

Note that we only need to define the interpretation on normal forms (c.f., the definition of the sequents of the assertion logic). Hence we omit the canonical forms in the brackets below and simply write, e.g., $\Delta \vdash A \Leftarrow \mathsf{type}\ [\,]$ for $\Delta \vdash A \Leftarrow \mathsf{type}[A]$.

- We let MonoTypeSubst = TyVar $\to$ MonoTypes denote the set of monotype substitutions (here TyVar denotes the set of type variables). We use θ to range over monotype substitutions.
- Types $\Delta \vdash A \Leftarrow \mathsf{type}\ [\,]$ are interpreted by V.
- Contexts $\vdash \Delta\ \mathsf{ctx}$ of length n are interpreted by $[\![\Delta]\!] = V^n$.[3]
- Contexts $\Delta; X$ of the form $\Delta; h_1, \ldots, h_m$ are interpreted by $[\![\Delta]\!] \times H^m$. We often use ρ to range over elements of $[\![\Delta]\!]$. and use μ to range over elements of H^n.
- Intro terms in context $\Delta \vdash M \Leftarrow A\ [\,]$ are interpreted by elements of MonoTypeSubst $\to [\![\Delta]\!] \to V$; the inductive definition is given in Figure 1.
- Elim terms in context $\Delta \vdash K \Rightarrow A\ [\,]$ are interpreted by elements of MonoTypeSubst $\to [\![\Delta]\!] \to V$; the inductive definition is given in Figure 2.
- Computations in context $\Delta; P \vdash E \Rightarrow x{:}A.\, Q\ [\,]$ are interpreted by elements of MonoTypeSubst $\to [\![\Delta]\!] \to (H \rightharpoonup (V \times H))$; the inductive definition is given in Figure 3.
- Computations in context $\Delta; P \vdash E \Leftarrow x{:}A.\, Q\ [\,]$ are interpreted by elements of MonoTypeSubst $\to [\![\Delta]\!] \to (H \rightharpoonup (V \times H))$; the inductive definition is given in Figure 4.
- Heaps in context $\Delta; X \vdash H \Leftarrow \mathsf{heap}\ [\,]$ are interpreted by MonoTypeSubst $\to [\![\Delta; X]\!] \to H$; the inductive definition is given in Figure 5.
- Propositions in context $\Delta; X \vdash P \Leftarrow \mathsf{prop}\ [\,]$ are interpreted by MonoTypeSubst $\to \mathcal{P}\,[\![\Delta; X]\!]$; the inductive definition is given in Figure 6. Here we implicitly apply the forgetful function from **pCpo** to **Set** and then use the powerset functor $\mathcal{P}$ of **Set**.

Lemma 25
The denotational semantics is well-defined.

Lemma 26
Substitution into canonical terms is modelled via the environment. We only show two cases below (really, there are cases corresponding to all those in Lemmas 16 and 17):

1. $[\![\Delta \vdash [\tau/\alpha]K \Rightarrow [\tau/\alpha]A\ [\,]]\!]_\theta\, \rho = [\![\Delta, \alpha \vdash K \Rightarrow A\ [\,]]\!]_{\theta[\alpha \mapsto \tau]}\, \rho$
2. $[\![\Delta \vdash [M/x]^m_{A^-}(N) \Leftarrow [M/x]^a_{A^-}(B)\ [\,]]\!]_\theta\, \rho = [\![\Delta, x{:}A \vdash N \Leftarrow B\ [\,]]\!]_\theta(\rho, [\![\Delta \vdash M \Leftarrow A\ [\,]]\!]_\theta\, \rho)$

[3] Here we include a V-factor for type variables, not only for program variables — could equally well have been omitted.

$$
\begin{array}{rcl}
[\![\Delta \vdash \mathsf{true} \Leftarrow \mathsf{bool}\ [\,]]\!]_\theta & = & \lambda\rho.\ \mathit{true} \\
[\![\Delta \vdash \mathsf{false} \Leftarrow \mathsf{bool}\ [\,]]\!]_\theta & = & \lambda\rho.\ \mathit{false} \\
[\![\Delta \vdash \mathsf{z} \Leftarrow \mathsf{nat}\ [\,]]\!]_\theta & = & \lambda\rho.\ 0 \\
[\![\Delta \vdash \mathsf{s}\ M \Leftarrow \mathsf{nat}\ [\,]]\!]_\theta & = & \lambda\rho.\ 1 + [\![\Delta \vdash M \Leftarrow \mathsf{nat}\ [\,]]\!]_\theta\ \rho \\
[\![\Delta \vdash M + N \Leftarrow \mathsf{nat}\ [\,]]\!]_\theta & = & \lambda\rho.\ [\![\Delta \vdash M \Leftarrow \mathsf{nat}\ [\,]]\!]_\theta\ \rho + [\![\Delta \vdash N \Leftarrow \mathsf{nat}\ [\,]]\!]_\theta\ \rho \\
[\![\Delta \vdash M \times N \Leftarrow \mathsf{nat}\ [\,]]\!]_\theta & = & \lambda\rho.\ [\![\Delta \vdash M \Leftarrow \mathsf{nat}\ [\,]]\!]_\theta\ \rho \times [\![\Delta \vdash N \Leftarrow \mathsf{nat}\ [\,]]\!]_\theta\ \rho \\
[\![\Delta \vdash \mathsf{eq}(M, N) \Leftarrow \mathsf{bool}\ [\,]]\!]_\theta & = & \lambda\rho.\ [\![\Delta \vdash M \Leftarrow \mathsf{nat}\ [\,]]\!]_\theta\ \rho = [\![\Delta \vdash N \Leftarrow \mathsf{nat}\ [\,]]\!]_\theta\ \rho \\
[\![\Delta \vdash () \Leftarrow 1\ [\,]]\!]_\theta & = & \lambda\rho.\ * \\
[\![\Delta \vdash \lambda x.\ M \Leftarrow \Pi x{:}A.\ B\ [\,]]\!]_\theta & = & \lambda\rho.\ (\lambda v.\ [\![\Delta, x{:}A \vdash M \Leftarrow B\ [\,]]\!]_\theta(\rho, v)) \\
[\![\Delta \vdash \Lambda\alpha.\ M \Leftarrow \forall\alpha.\ A\ [\,]]\!]_\theta & = & \lambda\rho.\ (\lambda\tau \in \mathrm{MonoTypes}.\, [\![\Delta, \alpha \vdash M \Leftarrow A\ [\,]]\!]_{\theta[\alpha \mapsto \tau]}(\rho, *)) \\
[\![\Delta \vdash \mathsf{dia}\ E \Leftarrow \Psi.X.\{P\}x{:}A\{Q\}\ [\,]]\!]_\theta & = & \lambda\rho.\ \lambda h.\ [\![\Delta; \mathsf{this}(\mathsf{init}) \wedge \exists\Psi.X.(P * \top) \vdash E \Leftarrow x{:}A.\ \forall\Psi.X.P \multimap Q\ [\,]]\!]_\theta\ \rho h \\
[\![\Delta \vdash \mathsf{eta}_\alpha\ K \Leftarrow \alpha\ [\,]]\!]_\theta & = & [\![\Delta \vdash K \Rightarrow \alpha\ [\,]]\!]_\theta
\end{array}
$$

Figure 1: Interpretation of Intro Terms

$$
\begin{array}{rcl}
[\![\Delta, x{:}A, \Delta_1 \vdash x \Rightarrow A\ [\,]]\!]_\theta & = & \lambda\rho.\ \pi_x(\rho) \\
[\![\Delta \vdash K\ M \Rightarrow [M'/x]^a_A(B)\ [\,]]\!]_\theta & = & \lambda\rho.\ ([\![\Delta \vdash K \Rightarrow \Pi x{:}A.\ B\ [\,]]\!]_\theta\ \rho)([\![\Delta \vdash M \Leftarrow A\ [\,]]\!]_\theta\ \rho) \\
[\![\Delta \vdash K\ \tau \Rightarrow [\tau/\alpha](B)\ [\,]]\!]_\theta & = & \lambda\rho.\ ([\![\Delta \vdash K \Rightarrow \forall\alpha.\ B\ [\,]]\!]_\theta\ \rho)(\theta(\tau))
\end{array}
$$

Figure 2: Interpretation of Elim Terms

A sequent $\Delta; h_1, \ldots, h_k; P_1, \ldots, P_n \Longrightarrow Q_1, \ldots, Q_m$ of the assertion logic is *valid* if, for all $\rho \in [\![\Delta]\!]$ and all $\mu \in H^k$,

$$[\![\Delta; X \vdash P_1 \wedge \cdots \wedge P_n\ [\,]]\!](\rho, \mu) \subseteq [\![\Delta; X \vdash Q_1 \vee \cdots \vee Q_m\ [\,]]\!](\rho, \mu).$$

Theorem 27 (Soundness of Assertion Logic)
All the axioms and rules of the assertion logic are sound with respect to the semantic notion of validity.

Proof: All the standard rules for classical logic are trivially sound since we interpret the logic as in sets. Thus it just remains to check that the basic axioms for equality are sound. But those are all easy to verify; the only interesting case is extensionality of functions represented by λ-terms. That holds because λ-terms are indeed interpreted by elements in V corresponding to honest functions. ■

Theorem 28 (Heap Soundness)
The assertion logic of HTT is heap sound.

Proof: We only include the argument for item 1 in the definition of heap soundness. Let χ be such that $\vdash \chi : \mathsf{heapval}$. By assumption $\cdot; \mathsf{mem}; \mathsf{Hld}(\mathsf{mem}, [\![\chi]\!]) \Longrightarrow \mathsf{seleq}_A(\mathsf{mem}, l, -)$ is derivable, so by logic also $\cdot; \cdot; \cdot \Longrightarrow \mathsf{seleq}_A([\![\chi]\!], l, -)$ is derivable. By soundness of the assertion logic (Theorem 27) and the definition of the semantics of the assertion logic, we have that $[\![\cdot; \cdot; \cdot \Longrightarrow \mathsf{seleq}_A([\![\chi]\!], l, -)]\!] = \mathsf{true}$. By the definition of the semantics of seleq_A we can calculate that this means that $\exists v \in V.\ [\![[\![\chi]\!]]\!](*)(l) = v$. By the definition of $[\![\chi]\!]$ and the semantics of heaps (Figure 5), we clearly have that $l \mapsto_A v_0 \in \chi$, for some value v_0, as required (and $[\![v_0]\!] *$ is the v that exists). ■

8 Related work

There has been a significant interest recently in systems for reasoning about effectful higher-order functions. Honda et al. [14, 3] present several Hoare Logics for total correctness, where specifications in the form of

$$
\begin{array}{l}
[\![\Delta; P \vdash M \Rightarrow x{:}A.\ P \wedge \mathsf{Id}_A(\mathsf{expand}_A(x), M')\ [\,]]\!]_\theta \\
\quad = \lambda\rho.\ \lambda h.\ ([\![\Delta \vdash M \Leftarrow A\ [\,]]\!]_\theta\, \rho, h) \\
[\![\Delta; P \vdash \mathsf{let\ dia}\ x = K\ \mathsf{in}\ E \Rightarrow y{:}B.\ (\exists x{:}A.\ Q)\ [\,]]\!]_\theta \\
\quad = \lambda\rho.\ \lambda h.\ [\![\Delta, x{:}A; P \circ (\forall\Psi_1.X_1.R_1 \multimap R_2) \vdash E \Rightarrow y{:}B.\ Q\ [\,]]\!]_\theta(\rho, [\![\Delta \vdash K \Rightarrow \Psi_1.X_1.\{R_1\}x{:}A\{R_2\}\ [\,]]\!]_\theta\, \rho)h \\
[\![\Delta; P \vdash x = \mathsf{alloc}_\tau(M); E \Rightarrow y{:}B.\ (\exists x{:}\mathsf{nat}.\ Q)\ [\,]]\!]_\theta \\
\quad = \lambda\rho.\ \lambda h.\ \ \mathsf{let} \\
\qquad\qquad m = [\![\Delta \vdash M \Leftarrow \tau'\ [\,]]\!]_\theta\, \rho \\
\qquad\qquad x = \text{choose } x \notin \mathsf{dom}(h) \\
\qquad\quad \mathsf{in}\ [\![\Delta, x{:}\mathsf{nat}; P * (x \mapsto_{\tau'} M') \vdash E \Rightarrow y{:}B.\ Q\ [\,]]\!]_\theta(\rho, x)(h[x \mapsto m]) \\
[\![\Delta; P \vdash x = [M]_\tau; E \Rightarrow y{:}B.\ (\exists x{:}\tau'.\ Q)\ [\,]]\!]_\theta \\
\quad = \lambda\rho.\ \lambda h.\ \ \mathsf{let} \\
\qquad\qquad m = [\![\Delta \vdash M \Leftarrow \mathsf{nat}\ [\,]]\!]_\theta\, \rho \\
\qquad\qquad x = h(m) \text{ if } x \in \mathsf{dom}(h) \\
\qquad\quad \mathsf{in}\ [\![\Delta, x{:}\tau'; P \wedge (M' \hookrightarrow_{\tau'} x) \vdash E \Rightarrow y{:}B.\ Q\ [\,]]\!]_\theta(\rho, x)h \\
[\![\Delta; P \vdash [M]_\tau = N; E \Rightarrow y{:}B.\ Q\ [\,]]\!]_\theta \\
\quad = \lambda\rho.\ \lambda h.\ \ \mathsf{let} \\
\qquad\qquad n = [\![\Delta \vdash N \Leftarrow \tau'\ [\,]]\!]_\theta\, \rho \\
\qquad\qquad m = [\![\Delta \vdash M \Leftarrow \mathsf{nat}\ [\,]]\!]_\theta\, \rho \\
\qquad\qquad h' = h[m \mapsto n] \text{ if } m \in \mathsf{dom}(h) \\
\qquad\quad \mathsf{in}\ [\![\Delta; P \circ ((M' \mapsto -) \multimap (M' \mapsto_{\tau'} N')) \vdash E \Rightarrow y{:}B.\ Q\ [\,]]\!]_\theta\, \rho h' \\
[\![\Delta; P \vdash x = \mathsf{if}_A(M, E_1, E_2); E \Rightarrow y{:}B.\ (\exists x{:}A'.\ Q)\ [\,]]\!]_\theta \\
\quad = \lambda\rho.\ \lambda h.\ \ \mathsf{let} \\
\qquad\qquad m = [\![\Delta \vdash M \Leftarrow \mathsf{bool}\ [\,]]\!]_\theta\, \rho \\
\qquad\quad \mathsf{in} \\
\qquad\quad \begin{cases}
\mathsf{let} & \text{if } m \\
\qquad (x, h') = [\![\Delta; P \wedge \mathsf{Id}_{\mathsf{bool}}(M', \mathsf{true}) \vdash E_1 \Rightarrow x{:}A'.\ P_1\ [\,]]\!]_\theta\, \rho h \\
\mathsf{in}\ [\![\Delta, x{:}A'; P_1 \vee P_2 \vdash E \Rightarrow y{:}B.\ Q\ [\,]]\!]_\theta(\rho, x)h' & \\
\mathsf{let} & \text{if not } m \\
\qquad (x, h') = [\![\Delta; P \wedge \mathsf{Id}_{\mathsf{bool}}(M', \mathsf{false}) \vdash E_2 \Rightarrow x{:}A'.\ P_2\ [\,]]\!]_\theta\, \rho h \\
\mathsf{in}\ [\![\Delta, x{:}A'; P_1 \vee P_2 \vdash E \Rightarrow y{:}B.\ Q\ [\,]]\!]_\theta(\rho, x)h' &
\end{cases} \\
[\![\Delta; P \vdash y = \mathsf{loop}^I_A(M, x.\ N, x.\ F); E \Rightarrow z{:}C.\ (\exists y{:}A'.\ Q)\ [\,]]\!]_\theta \\
\quad = \ldots \text{type in later} \\
[\![\Delta; P \vdash y = \mathsf{fix}_{\Pi x:A.\Psi.X.\{R_1\}y:B\{R_2\}}(f.x.E, M); F \Rightarrow z{:}C.\ (\exists y{:}[M/x]^p_{A'}(B').Q)\ [\,]]\!]_\theta \\
\quad = \lambda\rho.\ \lambda h.\ \ \mathsf{let} \\
\qquad\qquad \phi = \mathit{fix}\ \ \big(\lambda f{:}V \to (H \rightharpoonup V \times H).\ \lambda x.\ \lambda h. \\
\qquad [\![\Delta, f{:}\Pi x{:}A.\ \Psi.X.\{R_1\}y{:}B\{R_2\}, x{:}A; \mathsf{this(init)} \wedge \exists\Psi.X.(R_1 * \top) \vdash E \Leftarrow y{:}B.\ (\forall\Psi.X.R_1 \multimap R_2)\ [\,]]\!]_\theta(\rho, f, x)h\big) \\
\qquad\qquad m = [\![\Delta \vdash M \Leftarrow A\ [\,]]\!]_\theta\, \rho \\
\qquad\qquad (y, h') = \phi m h \\
\qquad\quad \mathsf{in}\ [\![\Delta, y{:}[M'/x]^p_A(B); P \circ [M'/x]^p_A(\forall\Psi.X.R_1 \multimap R_2) \vdash F \Rightarrow z{:}C.\ Q\ [\,]]\!]_\theta(\rho, y)h'
\end{array}
$$

Figure 3: Interpretation of Computations, I

$$
\begin{array}{l}
[\![\Delta; P \vdash E \Leftarrow x{:}A.\ Q\ [\,]]\!]_\theta \\
\qquad = [\![\Delta; P \vdash E \Rightarrow x{:}A.\ R\ [\,]]\!]_\theta
\end{array}
$$

Figure 4: Interpretation of Computations, II

$$
\begin{array}{rcl}
[\![\Delta; h_1, \ldots, h_n \vdash h_i \Leftarrow \mathsf{heap}\ [\,]]\!]_\theta & = & \pi_i : [\![\Delta]\!] \times H^n \to H \\
[\![\Delta; X \vdash \mathsf{emp} \Leftarrow \mathsf{heap}\ [\,]]\!]_\theta & = & \lambda(\rho,\mu).\{\} \\
[\![\Delta; X \vdash \mathsf{upd}_A(H,M,N) \Leftarrow \mathsf{heap}\ [\,]]\!]_\theta & = & \lambda(\rho,\mu).\mathsf{let} \\
 & & \quad h = [\![\Delta; X \vdash H \Leftarrow \mathsf{heap}\ [\,]]\!]_\theta(\rho,\mu) \\
 & & \quad m = [\![\Delta \vdash M \Leftarrow \mathsf{nat}\ [\,]]\!]_\theta(\rho) \\
 & & \quad n = [\![\Delta \vdash N \Leftarrow A'\ [\,]]\!]_\theta(\rho) \\
 & & \mathsf{in}\ h[m \mapsto n]
\end{array}
$$

Figure 5: Interpretation of Heaps

$$
\begin{array}{rcl}
[\![\Delta; X \vdash \mathsf{seleq}_\tau(H,M,N) \Leftarrow \mathsf{prop}\ [\,]]\!]_\theta & = & \{(\rho,\mu) \mid ([\![\Delta; X \vdash H \Leftarrow \mathsf{heap}\ [\,]]\!]_\theta(\rho,\mu))([\![\Delta \vdash M \Leftarrow \mathsf{nat}\ [\,]]\!]_\theta\,\rho) = \\
 & & \qquad [\![\Delta \vdash N \Leftarrow \tau\ [\,]]\!]_\theta\,\rho\} \\
[\![\Delta; X \vdash \mathsf{Id}_A(M,N) \Leftarrow \mathsf{prop}\ [\,]]\!]_\theta & = & \{(\rho,\mu) \mid [\![\Delta \vdash M \Leftarrow A\ [\,]]\!]_\theta\,\rho = [\![\Delta \vdash N \Leftarrow A\ [\,]]\!]_\theta\,\rho\} \\
[\![\Delta; X \vdash \top \Leftarrow \mathsf{prop}\ [\,]]\!]_\theta & = & [\![\Delta; X]\!] \\
[\![\Delta; X \vdash \bot \Leftarrow \mathsf{prop}\ [\,]]\!]_\theta & = & \emptyset \\
[\![\Delta; X \vdash P \wedge Q \Leftarrow \mathsf{prop}\ [\,]]\!]_\theta & = & [\![\Delta; X \vdash P \Leftarrow \mathsf{prop}\ [\,]]\!]_\theta \cap [\![\Delta; X \vdash Q \Leftarrow \mathsf{prop}\ [\,]]\!]_\theta \\
[\![\Delta; X \vdash P \vee Q \Leftarrow \mathsf{prop}\ [\,]]\!]_\theta & = & [\![\Delta; X \vdash P \Leftarrow \mathsf{prop}\ [\,]]\!]_\theta \cup [\![\Delta; X \vdash Q \Leftarrow \mathsf{prop}\ [\,]]\!]_\theta \\
[\![\Delta; X \vdash P \supset Q \Leftarrow \mathsf{prop}\ [\,]]\!]_\theta & = & \{(\rho,\mu) \mid (\rho,\mu) \in [\![\Delta; X \vdash P \Leftarrow \mathsf{prop}\ [\,]]\!]_\theta \text{ implies} \\
 & & \qquad (\rho,\mu) \in [\![\Delta; X \vdash Q \Leftarrow \mathsf{prop}\ [\,]]\!]_\theta\} \\
[\![\Delta; X \vdash \neg P \Leftarrow \mathsf{prop}\ [\,]]\!]_\theta & = & [\![\Delta; X]\!] \setminus [\![\Delta; X \vdash P \Leftarrow \mathsf{prop}\ [\,]]\!]_\theta \\
[\![\Delta; X \vdash \forall x{:}A.\ P \Leftarrow \mathsf{prop}\ [\,]]\!]_\theta(\rho,\mu) & \text{iff} & [\![\Delta, x{:}A; X \vdash P \Leftarrow \mathsf{prop}]\!]_\theta((\rho,v),\mu) \quad \text{for all values } v \in V \\
[\![\Delta; X \vdash \exists x{:}A.\ P \Leftarrow \mathsf{prop}\ [\,]]\!]_\theta(\rho,\mu) & \text{iff} & [\![\Delta, x{:}A; X \vdash P \Leftarrow \mathsf{prop}]\!]_\theta((\rho,v),\mu) \quad \text{for some values } v \in V \\
[\![\Delta; X \vdash \forall h{:}\mathsf{heap}.\ P \Leftarrow \mathsf{prop}\ [\,]]\!]_\theta(\rho,\mu) & \text{iff} & [\![\Delta; X, h \vdash P \Leftarrow \mathsf{prop}\ [\,]]\!]_\theta(\rho,(\mu,h)) \quad \text{for all heaps } h \in H \\
[\![\Delta; X \vdash \exists h{:}\mathsf{heap}.\ P \Leftarrow \mathsf{prop}\ [\,]]\!]_\theta(\rho,\mu) & \text{iff} & [\![\Delta; X, h \vdash P \Leftarrow \mathsf{prop}\ [\,]]\!]_\theta(\rho,(\mu,h)) \quad \text{for some heap } h \in H \\
[\![\Delta; X \vdash \forall \alpha.\ P \Leftarrow \mathsf{prop}\ [\,]]\!]_\theta(\rho,\mu) & \text{iff} & [\![\Delta, \alpha; X \vdash P \Leftarrow \mathsf{prop}\ [\,]]\!]_{\theta[\alpha \mapsto \tau]}((\rho,*),\mu) \quad \text{for all monotypes } \tau \\
[\![\Delta; X \vdash \exists \alpha.\ P \Leftarrow \mathsf{prop}\ [\,]]\!]_\theta(\rho,\mu) & \text{iff} & [\![\Delta, \alpha; X \vdash P \Leftarrow \mathsf{prop}\ [\,]]\!]_{\theta[\alpha \mapsto \tau]}((\rho,*),\mu) \quad \text{for some monotype } \tau
\end{array}
$$

Figure 6: Interpretation of Propositions

Hoare triples are taken as propositions. Krishnaswami [18] proposes a version of Separation Logic for a higher-order typed language. Similarly to HTT, Krishnaswami bases his logic on a monadic presentation of the underlying programming language. Both proposals do not support polymorphism, strong updates, deallocation or pointer arithmetic. Both are Hoare-like Logics, rather than type theories, and are thus subject to the criticism we outlined in Section 1.

Shao et al. [37] and Xi et al [43, 44] present dependently typed systems for effectful programs, based on singleton types, but they do not allow effectful terms in the specifications. Both systems encode a notion of pre- and postconditions. In the work of Xi, assertions are drawn from linear logic, and the proofs for pre- and postconditions are embedded within the code. It is interesting that the properties of linear logic actually require the embedding of proofs and code, unlike in HTT where this is optional. For most effectful commands, a precondition must be transformed into a suitable form (usually a linear product) before the postcondition can be computed at all. The proofs are necessary in order to guide this transformation of preconditions.

Mandelbaum et al. [20] develop a theory of type refinements for reasoning about effectful higher-order functions, but their specifications are restricted in order for the type checking to be decidable. In particular, it does not seem possible in that system to reason about state with aliasing. This system allows a form of type dependency via the use of singleton types.

Abadi and Leino [1] describe a logic for object-oriented programs where specifications, like in HTT, are treated as types. One of the problems that authors describe concerns the scoping of variables; certain specifications cannot be proved because the inference rule for $\mathsf{let\ val}\ x = E\ \mathsf{in}\ F$ does not allow sufficient interaction between the specifications of E and F. Such problems do not appear in HTT.

Birkedal et al. [6] describe a dependent type system for well-specified programs in idealized Algol extended with heaps. The type system includes a wide collection of higher-order frame rules, which are shown sound by a denotational model. A serious limitation of the type system compared to HTT is that the heap in *loc. cit.* can only contain simple integer values.

9 Future work

In this section we describe some future work that we plan to carry out, involving higher-order assertion logic, local state, and lifting other applications of Hoare Logic to HTT and higher-order functions.

Higher-order assertion logic. The polymorphic multi-sorted first-order assertion logic presented in the current paper is not enough. For any practical application, HTT needs internal means of defining new predicates, including inductive ones, and new types of data. Often in assertions, one needs to talk about a type of lists and a predicate describing that a heap contains a linked lists. All of these are definable in higher-order logic [8, 31, 40]. For purposes of HTT, the higher-order logic will also require polymorphic quantification over monotypes.

Furthermore, higher-order assertion logic should be the appropriate framework for studying Cook completeness of HTT [9], as with higher-order assertions it should be possible to exactly express the strongest postconditions for any kind of un-annotated looping or recursion construct of HTT.

Applications of Hoare Logic and higher-order functions. One way to view HTT is as a general framework for embedding Hoare Logics into Type Theory and provide higher-order functions. One important prerequisite for this embedding seems to be that the reasoning in the Hoare Logic in question supports strongest postconditions, or dually, weakest preconditions. In this paper, we applied HTT to the problem of reasoning about state with aliasing. But, other applications seem possible as well. For example, Separation Logic has been used recently to reason about concurrent programs [30] and we hope that the small footprint extension that we presented here may be applied to the same problem in a setting with higher-order functions. Another interesting domain is reasoning about information flow and security [2].

Local state. HTT specifications, as presented in this paper can only describe state that is reachable from the variables that are in scope, or from the return result of a computation. Local state, which, by definition, is not reachable in this way, but is implicit, and may be shared by functions or data structures, cannot be described. To enrich HTT types so that local state can be described, we require at least two components.

First, a computation should have more than one result so that it can return the addresses of locally allocated data. Thus, we will require a new type of Hoare triples, with a syntax as in $\Psi.X.\{P\}\Delta, x{:}A\{Q\}$, where Δ is a context of variables that abstracts over the local data of the computation. The variables from Δ can be used in the return type A and in the postcondition Q. This extension may employ some results from the Contextual modal type theory of [27].

Of course, if the local addresses are made explicit as the return result of the computation, they are not local anymore. The second component required for a type system of local state must provide a mechanism for existential abstraction over the context Δ. A related question is how to associate an abstract datatype (e.g. red-black trees) with chunks of local state.

10 Conclusions

In this paper, we extend our previous formulation of Hoare Type Theory (HTT) [26] with predicative polymorphism and small footprints in the style of Separation Logic [29, 35, 30, 36]. We also prove the soundness of the underlying assertion logic – a result that was missing in [26]. With this result, we complete the overall proof of type soundness of HTT with respect to the operational semantics.

HTT is a dependent type theory with a type constructor that captures the specifications of Hoare Logic to express precise statements about correctness of effectful programs with state and aliasing. The specification types of HTT can be used in programs to ensure that each computation is invoked only in contexts in which it is meaningful to do so. This capability is important – indeed it an instance of the general mechanism by which type systems reduce the complexity of program development. But this capability is not available in the usual formulations of Hoare Logics where programs cannot depend on specifications.

Our extension with small footprints improves the modularity of HTT, as it tightly relates a computation with the fragment of the heap which the computation actually uses. This is in contrast to our previous work [26], where the specifications had to describe the properties of the whole program heap.

We further argue that polymorphism and small footprints should be developed together. Polymorphism is, of course, of independent interest for program development and reuse, but it is possible that the small footprint extension could have been achieved separately, perhaps by utilizing axiomatizations based on the Logic of Bunched Implication [34].

In the presence of polymorphism we can already define the spatial connectives of Separation Logic, but more is needed to express the strongest postconditions of higher-order computations. In particular, we need the ability to explicitly name and manage heap fragments in order to assert their invariance across computations. This kind of property does not seem expressible by spatial connectives alone, thus showing that polymorphism can be used as an essential ingredient of small footprints.

References

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

[2] T. Amtoft, S. Bandhakavi, and A. Banerjee. A logic for information flow in object-oriented programs. In *Symposium on Principles of Programming Languages, POPL'06*, pages 91–102, Charleston, South Carolina, 2006.

[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] B. Biering, L. Birkedal, and N. Torp-Smith. BI hyperdoctrines, Higher-Order Separation Logic, and Abstraction. Technical Report ITU-TR-2005-69, IT University of Copenhagen, Copenhagen, Denmark, July 2005.

[5] L. Birkedal, N. Torp-Smith, and J. C. Reynolds. Local reasoning about a copying garbage collector. In *Symposium on Principles of Programming Languages, POPL'04*, pages 220–231, Venice, Italy, 2004.

[6] 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.

[7] 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.

[8] A. Church. A formulation of the simple theory of types. *The Journal of Symbolic Logic*, 5(2):56–68, Jun 1940.

[9] S. A. Cook. Soundness and completeness of an axiom system for program verification. *SIAM Journal of Computing*, 7(1):70–90, 1978.

[10] 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.

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

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

[13] M. Hofmann. *Extensional Concepts in Intensional Type Theory*. PhD thesis, Department of Computer Science, University of Edinburgh, July 1995. Avaliable as Technical Report ECS-LFCS-95-327.

[14] 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.

[15] W. A. Howard. The formulae-as-types notion of construction. In *To H.B.Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 479–490. Academic Press, 1980.

[16] 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.

[17] 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.

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

[19] 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.

[20] 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.

[21] P. Martin-Löf. On the meanings of the logical constants and the justifications of the logical laws. *Nordic Journal of Philosophical Logic*, 1(1):11–60, 1996.

[22] C. McBride. *Dependently Typed Functional Programs and their Proofs*. PhD thesis, University of Edinburgh, 1999.

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

[24] E. Moggi. Computational lambda-calculus and monads. In *Symposium on Logic in Computer Science, LICS'89*, pages 14–23, Asilomar, California, 1989.

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

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

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

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

[29] 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.

[30] 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.

[31] L. C. Paulson. A formulation of the simple theory of types (for Isabelle). In *International Conference in Computer Logic, COLOG'88*, volume 417 of *Lecture Notes in Computer Science*, pages 246–274. Springer, 2000.

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

[33] B. C. Pierce and D. N. Turner. Local type inference. *ACM Transactions on Programming Languages and Systems*, 22(1):1–44, 2000.

[34] D. J. Pym, P. W. O'Hearn, and H. Yang. Possible worlds and resources: The semantics of BI. *Theoretical Computer Science*, 315(1):257–305, 2004.

[35] 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.

[36] J. C. Reynolds. Lecture notes for the course "An introduction to separation logic". Available at `http://www.cs.cmu.edu/~jcr/www15818A4s2005/notes6.ps`, Spring 2005.

[37] 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.

[38] F. Smith, D. Walker, and G. Morrisett. Alias types. In G. Smolka, editor, *European Symposium on Programming, ESOP'00*, volume 1782 of *Lecture Notes in Computer Science*, pages 366–381, Berlin, Germany, 2000.

[39] J. M. Smith. The Independence of Peano's Fourth Axiom from Martin-Löf's Type Theory without Universes. *Journal of Symbolic Logic*, 53(3):840–845, 1988.

[40] SRI International and DSTO. *The HOL System: Description*. University of Cambridge Computer Laboratory, July 1991.

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

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

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

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