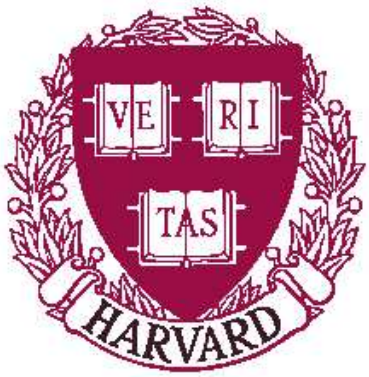
Dependent Type Theory of Stateful Higher-Order Functions

Aleksandar Nanevski and Greg Morrisett

TR-24-05



Computer Science Group
Harvard University
Cambridge, Massachusetts

Dependent Type Theory of Stateful Higher-Order Functions

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

January 6, 2006

Abstract

In this paper we investigate a logic for reasoning about programs with higher-order functions and effectful features like non-termination and state with aliasing. We propose a dependent type theory HTT (short for Hoare Type Theory), where types serve as program specifications. In case of effectful programs, the type of Hoare triples $\{P\}x{:}A\{Q\}$ specifies the precondition P, the type of the return result A, and the postcondition Q.

By Curry-Howard isomorphism, a dependent type theory may be viewed as a functional programming language. From this perspective, the type of Hoare triples is a monad, and HTT is a monadic language, whose pure fragment consists of higher-order functions, while the effectful fragment is a full Turing-complete imperative language with conditionals, loops, recursion and commands for stateful operations like allocation, lookup and mutation of location content.

1 Introduction

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

Higher-order functions are one of the most important mechanisms in the development of well-structured, modular and reusable programs. In the absence of side effects, higher-order functional programming as embodied by the typed lambda calculus exhibits a strong connection to logic. Via the Curry-Howard isomorphism, types in the lambda calculus may be viewed as specifications, while the programs themselves are proofs that the specifications are satisfied. The specifications-as-types and proofs-as-programs paradigm is particularly apparent in dependent type theories [27, 11, 18], where types may contain terms and can therefore express arbitrarily complex properties of programs. The connection with logic makes it relatively easy to reason about functional programs, and consequently, makes it relatively easy to write, debug and maintain functional programs. Unfortunately, the connection breaks in the presence of side effects, like state or non-termination.

Reasoning about imperative programs with side effects also has a long tradition in the programming languages research. We single out Hoare logic [19] as the particularly successful and important representative of this line of work. In Hoare logic, programs are annotated with preconditions and postconditions that describe the properties of the state before and after the execution of the program. For example, if the program E has a precondition P and a postcondition Q – abbreviated using the notation of Hoare triples as $\{P\}E\{Q\}$ – then whenever the execution of E is attempted in the state of which P holds, then Q holds of the state in which E terminates, under the proviso that E actually terminates[1]. Hoare logic has been most successfully employed to reason about first-order imperative programs. Extensions to programs with procedures and more generally higher-order functions have been known to be subtle [7, 9, 8, 1]. As discussed by Apt in his survey of Hoare Logic [3] (page 462), the problems usually appear because "... the semantics of

[1]This is the definition of Hoare triples for partial correctness. To establish total correctness one must also prove that E terminates. We do not discuss total correctness in this paper.

parameter passing is always modeled syntactically by some form of variable substitution in a program, and this leads to various subtle problems concerning variable clashes. These difficulties are especially acute in the presence of recursion and static scoping". The complications mentioned by Apt are related to the treatment of variables as l-values, and often result in restricting of the programming language or the assertion logic in some way, or in sacrificing the modularity by requiring whole-program reasoning.

In this paper, we describe a system, which we call HTT (short for Hoare Type Theory) that can be viewed as a Hoare-like logic of imperative programs with higher-order functions. The system builds on the classical work of Cartwright and Oppen [8] in that state is represented as a functional array with the usual primitives for element access and update, as axiomatized by McCarthy [28]. Our contribution is the formulation of HTT as a dependent type theory, and this has several important consequences. First, variables in type theories are immutable and have static scope, which significantly simplifies the considerations and avoids the difficulties mentioned by Apt. Second, HTT re-establishes the Curry-Howard isomorphism between logic and higher-order functional programming with side effects, in the important special case when the side effects in question concern non-termination and state with first-class locations and aliasing. The Curry-Howard property of the system is essential, because it leads to modularity in the design and specification of programs.

The critical component of every type theory is the equational reasoning about terms. Usually, the equations admitted in this reasoning are a selected set of beta and eta rules for the various connectives of the lambda calculus underlying the type theory, in addition to the rules expressing congruence and equivalence of equality. In the absence of side effects, these equations are sound and complete axiomatization of program equivalence [29]. But, with side effects, and in particular, with non-termination, the lambda calculus is Turing complete; equality between terms is not recursively enumerable, and therefore not finitely axiomatizable. Hence, rather than adding more equations to relate imperative programs, we only relate their inputs and outputs via a system of pre- and postconditions. This is the relationship of our Hoare type theory with the standard Hoare logic.

This development suggests that the terms in HTT should split into two fragments. The first fragment consist of constructs for which we can employ Hoare-like reasoning by pre- and postconditions. This includes the stateful commands, loops, conditionals and recursion. The second fragment consists only of pure, dependent, higher-order functions, and the equational reasoning about this fragment follows the traditional approach of type theory. Stateful computations can be internalized into the pure fragment, by means of a type constructor $\{P\}x{:}A\{Q\}$. This type classifies suspended computations that when executed in a state satisfying the proposition P, produce a value of type A and a new state satisfying Q, if they terminate. The variable x names the return value of the computation, and Q may depend on x. The propositions P and Q are drawn from a classical multi-sorted first-order logic, whose sorts include heaps and all the types of our type theory, i.e. booleans, natural numbers (which also stand for locations), dependent functions, and effectful computations themselves.

The described distinction between the reasoning methods for pure and effectful terms – while at the same time internalizing effectful computations into the pure fragment via a type constructor for Hoare triples – is what separates our approach from the recent related work.

For example, Honda et al. [21, 5] present a succession of increasingly powerful logics for reasoning about functional programs with references. The main feature of their logics is a proposition asserting that a given function application terminates and maps a state satisfying the precondition into a state satisfying the postcondition. In contrast, our system does not provide pre- and postconditions for functions or function applications, per-se. We use a Hoare-like treatment only for the effectful fragment of the language. Higher-order functions are pure, and are thus subject to the usual type-theoretic equational reasoning. We believe that this leads to a significant conceptual simplifications over the logics of [21, 5].

The work of Shao et al. [42] on type systems for Proof-carrying code and the work of Xi et al. [46, 47] on Applied type systems are approaches to dependent type systems for effectful programs. Both of these systems consist of an effectful language for writing programs, and a pure language of types and kinds for writing specifications. The connection between the two language levels is established by singleton types. However, the specification language cannot contain effects, which restricts the specifications to talk only about pure values from the language of programs. In HTT, all terms can be used in the specifications, which allows us

to avoid the indirectness of the singleton types. In addition, because effectful computations are first-class values, specifications in HTT can depend on effectful computations, and can thus conceivably express a variety of relations on computations (including safety, liveness, or contextual equivalence properties). Such an approach will require user-defined and inductive predicates, which we do not consider in the current paper, but we believe that HTT is consistent with such an extension.

Mandelbaum et al. [26] develop a theory of type refinements for reasoning about behavior of effectful programs. Like HTT, their system consists of two levels. The pure level corresponds to the customary ML-style type system, while the effectful level essentially facilitates reasoning by pre- and postconditions. The pre- and postconditions are drawn from a substructural assertion logic, and can be parametrized with respect to various effectful commands. For example, one presented instantiation can enforce that every file must be open before it is read. This assertion logic, however, is rather restricted in order for typechecking to be decidable, so it is not clear whether it can be extended with equational reasoning about programs with state and aliasing. The language supports a variant of dependent typing via singleton types, somewhat similar in nature to the systems discussed in the previous paragraph [46, 42].

We also single out the works of Hamid and Shao [17] and Ni and Shao [32] on Certified assembly programming. These papers consider reasoning frameworks for assembly programs written in a continuation passing style (CPS), and [32] allows embedded code pointers (and thus higher-order functions) to appear in the assembly programs. The embedded code pointers are added by introducing a proposition that establishes the safety of jumping to a given code pointer if a certain precondition is satisfied. All such proposition are later interpreted and proved correct with respect to the whole-program heap. In HTT, we use a type, rather than a proposition to capture the semantics of Hoare triples, and the semantics of Hoare triples is established not by interpretation, but directly as a meta theorem expressing the substitution properties for computations. Also, HTT does not require that the programs are written in CPS style.

Finally, we mention Separation logic [33, 41, 34] as a variant of Hoare logic particularly suited for reasoning about state with aliasing. There are two main characteristics of Separation logic. First, the separation propositional connectives allow very concise statements about disjointness of heaps. Second, Separation logic facilitates local reasoning about programs, in the sense that pre- and postconditions need only describe the heap that the program actually touches. The untouched parts of the heap are automatically considered invariant.

In this paper we use a classical logic of assertions, and thus our propositions about disjointness clearly cannot be as concise as those of Separation logic. We have decided on classical logic because it has a well-studied proof theory, and has been employed in several practical verification systems, like ESC/Java [25, 12] and Cyclone [22]. It is quite plausible that separating connectives may be added to HTT, and we discuss this possibility in Section 7. The focus of the current paper, however, is not on any particular assertion logic, but is rather on the type theory of Hoare triples and the interaction of state and non-termination with higher-order functions and dependent types. Having said this, it is important to mention that HTT satisfies a similar property of local reasoning as Separation logic, and we prove this fact in Section 5.

Recently, a type theoretic approach to Separation logic has been advocated by Birkedal et al. [6]. This work is similar to ours – at least in spirit – in the sense that it contains dependent types and a type of stateful computations. However, it is significantly restricted when compared to HTT; for example, the integer expressions that can appear in the dependent types seem to be strictly second-class in the sense that they cannot appear as function arguments or be returned as function results. No such restrictions exist in HTT.

The dichotomy between pure and effectful computations outlined above, and type systems that exploit it, are well-known in functional programming. In particular, this is the foundation behind the idea of monads [30, 44, 23], and monadic functional languages like Haskell [35]. Indeed, our system is a dependently typed, monomorphic, monadic lambda calculus, whose effectful fragment comprises a full Turing-complete first-order imperative language. The type constructor for Hoare triples $\{P\}x{:}A\{Q\}$ is a monad indexed with P and Q. The HTT calculus is based on the judgmental presentation of monads by Pfenning and Davies [38], and we also employ a dependently typed formulation with canonical forms and hereditary substitutions as proposed by Watkins et al. [45], with significant extensions that are particular to our application. The main

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

From the logical standpoint, monads correspond to lax logic [13, 4, 2, 38], which is a particularly simple intuitionistic modal logic. In terms of Kripke semantics of possible worlds, the modal connective of lax logic is an existential quantifier over possible worlds. The possible worlds semantics provides an intuitive logical reading of our computation type $\{P\}x{:}A\{Q\}$ as a bounded existential quantifier over possible worlds: a term of this type is a proof that starting in a possible world in which P holds, there *exists* a possible target world in which A is true and in which Q holds as well. The constructive content of this proof is to actually modify the starting world until the target world is obtained. Non-terminating computations describe how to reach an inconsistent world.

Of course, via the Curry-Howard isomorphism, the proof terms in our type theory may be seen as programs, and we make this explicit by providing a call-by-value operational semantics. The operational semantics is based on the beta reductions for functions and monadically encapsulated computations, and on the world-modification semantics outlined above for the impure commands. We also prove the progress and preservation theorems, thus establishing the soundness of HTT with respect to evaluation (relative to the soundness of the underlying logic of assertions used in the Hoare triples).

2 Overview

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

We start by defining the HTT syntax in the table below.

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

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

The type $\{P\}x{:}A\{Q\}$ classifies computations that execute under a precondition P and, if they terminate, return a value $x{:}A$ and a postcondition Q. Here, P and Q are propositions that describe and relate the

properties of the memory (i.e., the heap) at the beginning and the end of the computation. The variable x is bound in the type, and its scope extends through the postcondition Q.

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

The dependent function type $\Pi x{:}A.\ B$ in which B does not depend on x is abbreviated as $A \to B$, and the computation type $\{\top\}x{:}A\{\top\}$ is abbreviated as $\Diamond A$.

We will occasionally use p, q and variants, to range over primitive propositions.

Locations, heaps and the memory model. The semantics of memory that we assume in this paper can best be summarized by the following informal equations.

$$\text{Heaps} \quad = \quad \bigcup_{\alpha \overset{\text{fin}}{\subseteq} \text{Types}} \prod_{A \in \alpha} \mathsf{nat} \overset{\text{fin}}{\to} A$$

In other words, each heap is a finite collection of tuples (A, M, N) where A is a type, M is an integer denoting an address in the heap, and N is an element of type A. The tuple (A, M, N) is called an *assignment* of H. If (A, M, N) is an assignment in the heap H, we say that the location M *points to* N, or that N is the *content* of location M, or that the heap H *assigns* N to the location M. Given type A, the A-*section* of the heap H is the set of pairs (M, N) where (A, M, N) is an assignment in H. Each heap is a *function* in the sense that given an address M, there is *at most* one assignment (A, M, N) in the heap H.

Syntactically, we adopt the following constructs to represent heaps: emp denotes the empty heap, and $\mathsf{upd}_A(H, M, N)$ denotes a heap in which the location M points to the element N from the type A, but all the other assignments equal those of the heap H. If the heap H already assigns something to the location M, that assignment is replaced with (A, M, M) in the updated heap $\mathsf{upd}_A(H, M, N)$. We also allow heap variables, and use h to vary over them.

Now we can define the following propositions which will frequently appear in our assertions.

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

Terms. Terms form the pure fragment of HTT, and contain the basic operations on primitive types (e.g. equality, arithmetic), as well as the constructs for introduction and elimination of non-primitive types like 1, $\Pi x{:}A.\ B$ and $\{P\}x{:}A\{Q\}$.

For example, true and false are the constructors of Boolean type, as is he equality operator on integers $\mathsf{eq}(M, N)$. The operators z and $\mathsf{s}\ M$ are the standard constructors for zero and the successor function in Peano arithmetic, and we also include the primitive operations of addition and multiplication on integers.

Following the approach of Watkins et al. [45], we separate the terms into the syntactic categories of introduction terms (or intro terms) and elimination terms (or elim terms). For example, the introduction term for the function type is the lambda abstraction $\lambda x.\ M$, while the corresponding elimination term is the application $K\ M$. Similarly, the introduction term for the unit type is ( ) (with no corresponding elimination

constructs). The introduction term for the computation type $\{P\}x{:}A\{Q\}$ is $\mathsf{dia}\ E$, which encapsulates the computation E and coerces it into the pure fragment. The corresponding elimination form is not a term constructor, but a computation constructor, and we will discuss it subsequently in the paragraph describing computations.

Because of the syntactic distinction between intro and elim terms, a significant amount of type information may be omitted from the terms with no consequences for typechecking. Intuitively, elimination terms decrease the type information, and hence their types can be *inferred* from the components of the elimination term. The introduction forms increase the type information and hence the type against which they are *checked* must be somehow supplied from the surrounding context. This leads to a bidirectional typechecking system [39] where terms are minimally decorated with types, because the type information can appropriately be inferred and propagated as necessary. The types must be stated explicitly only when an introduction term appears in an elimination construct. In this situation, the elimination construct suggests that a type should be inferred, but this is not possible as the introduction term admits only checking, but not inference. This motivates the HTT constructor $M : A$ which allows that an intro term M be used as an elim term, but only if it is decorated with a type A.

An introduction term followed by an elimination term is precisely a definition of a beta redex. Hence, the separation of terms into introduction and elimination categories requires that types be explicitly provided only at beta redexes, but nowhere else. Moreover, it naturally leads to the following syntactic criterion for normality of terms. A term is in *normal form* iff it does not contain a beta redex iff it does not contain the term constructor $M : A$. In the subsequent sections we will phrase the process of normalization (i.e. reducing a term to its normal form) essentially as an algorithm that seeks to eliminate the appearances of the constructor $M : A$.

We conclude the overview of HTT terms by noting that the stated purity of terms means not only that the terms do not contain any effectful constructs, but also that no free heap variable may appear in a term. The same restriction on free heap variables holds of HTT types as well.

Computations. Computations form the effectful fragment of HTT. They correspond rather closely to programs in a generic imperative and sequential programming language. Intuitively, each computations is a semicolon-separated sequence of commands that perform effects and return with a result that is subsequently bound to a designated variable. Thus we have:

1. $x = \mathsf{alloc}_A(M); E$, which allocates a portion of the heap, and initializes it with the value of M. The address of the heap segment is bound to x, and the evaluation proceeds with the computation E.

2. $x = [M]_A; E$, looks up the content of the location M, and stores the value in x. Before the lookup can be performed, we must prove that M points to a value of type A.

3. $[M]_A = N; E$, mutates the content of the heap space assigned to the location M, by writing N (of type A) into it.

4. $x = \mathsf{if}_A(M, E_1, E_2); E$, executes the computation E_1 or the computation E_2 depending whether the Boolean term M denotes true or false, respectively. The computations E_1 and E_2 both return a result of type A, which is then bound to x.

5. $x = \mathsf{loop}^I_A(M, y.N, y.F); E$ is a looping construct with an *accumulator*. The term M is the initial value of the accumulator with which the loop starts the computation. N is a Boolean guard: if N evaluates to true when the free variable y is bound to the accumulator, then the iteration continues. Otherwise, the loop exits, returning the accumulator as the result to be bound to x. The computation F is the loop body, and is executed in each iteration as follows: first the free variable y is bound to the current value of the accumulator, and then the result of evaluating F is used as the new accumulator value. The proposition I is the loop invariant, and the type A is the type of the accumulator, and hence also the type of M and of the return value x.

6. $x = \mathsf{fix}_A(f.y.F, M); E$ is a recursion construct, adapted for the monadic presentation of HTT. Here A is a type of the form $A = \Pi z{:}B.\{R_1\}x{:}C\{R_2\}$. Intuitively, the semantics of fix is as follows: first it defines the function f of type A, which is the least fixed point of the equation $f = \lambda y.\ \mathsf{dia}\ F$. The function f is immediately applied to $M : B$ to obtain a computation that is subsequently executed and its result (of type $[M/z]C$, where $[M/z]$ is a capture-avoiding substitution of M for z) is bound to x.

7. The computation that simply consist of the term M is the trivial computation that immediately returns M, without performing any effects.

8. The construct $\mathsf{let\ dia}\ x = K\ \mathsf{in}\ E$ sequentially composes the computation encapsulated in the term K, with the computation E. This construct is the elimination form for the computation types.

In the usual presentation of the monadic lambda calculus [30, 44, 23], the last two constructs above correspond to the monadic *unit* and *bind*, respectively. Our formulation has a better proof theory and is directly adopted from the work of Pfenning and Davies [38].

The computations of HTT implement an imperative programming language, but we emphasize here that the treatment of variables in HTT computations follows the functional, rather than the imperative programming paradigm. In all the computation constructs discusses above, the variable x is considered bound, with the scope extending through E, and is subject to alpha renaming.

Reductions, expansions and substitutions. In this segment we illustrate some aspects of the equational theory of HTT. In particular, we present the beta reductions and eta expansions that correspond to the non-primitive types of HTT.

The unit type 1 and the function types are standard. There are no beta redexes corresponding to 1, because there are no elimination terms for 1, but the eta expansion is as follows:

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

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

$$(\lambda x.\ M : \Pi x{:}A.\ B)\ N \implies_\beta [N : A/x]M$$

and the eta expansion is:

$$M : \Pi x{:}A.\ B \implies_\eta \lambda y.\ (M : \Pi x{:}A.\ B)\ y \qquad \text{where } y \notin \mathsf{FV}(M : \Pi x{:}A.\ B)$$

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 situation is slightly more involved with the computation types. Beta reduction on computations should implement a sequential composition of two computations. To this end, we define an appropriate auxiliary operation of *monadic substitution* $\langle E/x : A\rangle F$, which composes E and F sequentially. The operation is defined by induction on the structure of E, and we again follow the presentation of Pfenning and Davies [38].

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

In the above definition, we used c to range over the computation constructs for allocation, lookup, mutation, conditionals, loops and recursion, as all of these cases are treated in the same way by the monadic substitution.

With the monadic substitution defined, we can specify the beta reduction for computation types as:

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

The eta expansion does not require any special treatment:

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

where $y \notin \mathsf{FV}(M : \{P\}x{:}A\{Q\})$.

We conclude this 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 propositions. The substitution simply commutes with most of the constructors, except that it leaves terms and types invariant. Recall that terms and types are pure and thus do not depend on free heap variables.

$$\begin{array}{lcll}
[H/h](h) & = & H & \\
[H/h](g) & = & g & \text{if } g \neq h \\
[H/h](\mathsf{emp}) & = & \mathsf{emp} & \\
[H/h](\mathsf{upd}_A(G, M, N)) & = & \mathsf{upd}_A([H/h]G, M, N) & \\
 & & & \\
[H/h](\mathsf{Id}_A(M, N)) & = & \mathsf{Id}_A(M, N) & \\
[H/h](\mathsf{HId}(G_1, G_2)) & = & \mathsf{HId}([H/h]G_1, [H/h]G_2) & \\
[H/h](\mathsf{indom}(G, M)) & = & \mathsf{indom}([H/h]G, M) &
\end{array}$$

We omit the rest of the propositional cases from the definition, as they follow the same pattern, and do not contribute any new insights.

Examples. Our first example presents the HTT version of a function for swapping the content of two (possibly aliased) locations. Here and in the rest of the paper, we use the following concrete syntax: `[x:A]B` stands for the dependent function type $\Pi x{:}A.\ B$, `fn x. M` stands for $\lambda x.\ M$, `??` and `!!` are existential and universal quantifiers, and `&&`, `||`, `==>` and `~~` stand for propositional conjunction, disjunction, implication and negation, respectively. We also omit the type annotation from the imperative commands and from the primitive propositions, when that does not result in ambiguities.

```
swap : [x:nat] [y:nat] {seleq(mem, x, -) && seleq(mem, y, -)} r : 1
                          {!!v1:A. !!v2:A. seleq(init, x, v1) &&
                                           seleq(init, y, v2) ==>
                                        HId(mem, upd(upd(init, x, v2), y, v1))} =
       fn x. fn y. dia(v1 = [x];
                       v2 = [y];
                       [y] = v1;
                       [x] = v2; ())
```

The function `swap` takes the locations x and y, and then returns a (suspended) computations. This computation, when activated, binds the content of x and y to variables v_1 and v_2 respectively, and writes them back into the memory, but in a swapped order. The computation is interesting only for its effects, and thus its return value is ( ). We have implicitly assumed that all the seleq and upd constructs out of x and y are indexed by some type A (which we omit for brevity).

The type of `swap` deserves further comments. The precondition of the computation produced by `swap` requires that x and y are locations that are actually allocated when the computation starts. If this condition is not satisfied, the computation will get stuck because it dereferences x and y. The precondition is allowed to depend on the heap variable mem, which denotes the heap at the beginning of the computation. This variable is considered bound by the precondition.

Observe that the precondition does not specify in any way the values that x and y point to. It is the job of the postcondition to describe what these values are and how they may be changed by the computation. In fact, more generally, the postcondition needs to relate the old heap that existed at the beginning of the computation, with the new heap obtained after the computation has finished. Thus, the postcondition must depend on two heap variables init and mem which denote the old and the new heap, respectively. The two variables are considered bound by the postcondition.

It will be one of the most important tasks of the HTT typing rules in Section 4 to make sure that the heap variable mem from the precondition, and the heap variable init from the postcondition, refer to one and the same heap. Without establishing this connection, the HTT type theory would not be sound.

At this point it is worth relating our approach to computation typing where the postcondition relates the two heaps, with the classical approach of Hoare logic [19, 21, 5, 41]. In most variants of Hoare logic, the specification for the program like `swap x y` that swaps the values pointed to by x and y would take the form of a Hoare triple very similar to the following (we employ here an informal stylized notation similar to HTT, in order to better illustrate the connections):

$$\{\mathsf{seleq}_A(\mathsf{mem}, x, v_1) \wedge \mathsf{seleq}_A(\mathsf{mem}, y, v_2)\}\ \mathsf{swap\ x\ y}\ \{\mathsf{seleq}_A(\mathsf{mem}, x, v_2) \wedge \mathsf{seleq}_A(\mathsf{mem}, y, v_1)\}$$

The above Hoare triple uses variables v_1 and v_2, which are not bound anywhere and cannot appear in the program, but only in the assertions of the Hoare triple. These variables are *schematic*, also frequently called *logic* variables. Because the scope of schematic variables is global, they can appear simultaneously in the precondition and the postcondition, thus establishing the connection between the state in which the computation started and in which it ended. Unfortunately, schematic variables are not easily and naturally reconciled with functional programming and type theory, because they cannot appear in programs or proof terms. This is why we avoid them in HTT and instead relate the input and output heaps of a computation via the postcondition, as discussed above.

For the second example, let us illustrate how to use computations obtained from function calls. The function below transposes the content of three locations several times, using `swap`. We use the customary ML-like concrete syntax for a nested sequence of let bindings. When the let-bound variable is of no interest, we replace it with the underscore.

```
identity : [x:nat] [y:nat] [u:nat]
               {seleq(mem, x, -) && seleq(mem, y, -) && seleq(mem, u, -)} r : 1
               {Hid(init, mem)} =
      fn x. fn y. fn u. dia(let dia _ = swap x y
                                dia _ = swap y u
                                dia _ = swap x y
                                dia _ = swap x u
                            in
                                ()
                            end)
```

Again, we assume that the seleq's in the precondition are at the same given type A. Each of the nested let-dia's executes a swapping computation on the supplied two variables. The return type of `swap` is 1, so we ignore the values that it computes. After four swaps, the initial heap is restored, as evident from the postcondition.

Our third example is the function `double` adapted from Honda et al. [21]. `double` takes two integer locations and doubles their contents. If the two locations are aliased, then the content of that location is quadrupled. In either case, the function returns the sum of the two original contents. In this example, we use the decimal notation for integers, rather than the unary notation of Peano numerals.

```
double : [x:nat] [y:nat]
           {seleq(mem, x, -) && seleq(mem, y, -)} r : nat
           {!!v1:nat. !!v2:nat.
                (seleq(init, x, v1) && seleq(init, y, v2)) ==>
                  (Id(r, v1+v2) &&
                     (~~Id(x, y) ==> seleq(mem, x, 2*v1) && seleq(mem, y, 2*v2)) &&
                     (Id(x, y) ==> seleq(mem, x, 4*v1)))} =
       fn x. fn y. dia (w1 = [x];
                        w2 = [y];
                        [x] = w1 + w1;
                        w3 = [y];
                        [y] = w3 + w3; w1 + w2)
```

The index types at seleq and upd here are all nat, because the locations x and y point to integers.

If we were interested in writing a function that requires the argument locations to be non-aliased, we could specify that requirement in the precondition, as follows.

```
double' : [x:nat] [y:nat]
           {seleq(mem, x, -) && seleq(mem, y, -) && ~~Id(x, y)} r : nat
           {!!v1:nat. !!v2:nat.
               (seleq(init, x, v1) && seleq(init, y, v2)) ==>
                 Id(r, v1+v2) && seleq(mem, x, 2*v1) && seleq(mem, y, 2*v2)} =
      fn x. fn y. dia (w1 = [x];
                       [x] = w1 + w1;
                       w2 = [y];
                       [y] = w2 + w2; w1 + w2)
```

A more traditional approach to the HTT program above, may employ a type of proofs so that along with the locations x and y, the doubling function is passed a proof that x and y are distinct. If such a proof is not available, the function may not even be invoked. Proofs should be computationally irrelevant, and hence ignored for the purposes of evaluation [20, 37]. We do not pursue that approach here, but we believe that an extension of HTT with the type of proofs and proof irrelevancy should be possible in the future work.

3 Normal forms and hereditary substitutions

Equational reasoning about terms in type theories usually requires that terms be converted into some kind of normal form before they can be compared for equality. The conversion to normal form is usually defined only on well-typed terms, making the equational reasoning and typechecking mutually dependent on each other. For the purposes of HTT, we adopt an approach due to Watkins et al. [45], where equational reasoning and typechecking are disentangled, essentially by allowing normalization of terms that are not necessarily well typed. This leads to significant conceptual simplifications of the system and its meta theory.

The main idea is to consider the syntactic structure of (not necessarily well-typed) normal terms and define substitutions which preserve this structure. For example, in places where ordinary capture-avoiding substitution of a normal term into another normal term creates a redex, like $(\lambda x.\ M)\ N$, we need to continue the substitution process by substituting N for x in M. This may produce another redex, which must be immediately reduced, initiating another substitution, and so on. Following [45], we call this kind of repeated substitution operation *hereditary substitution*.

The difficulty is that hereditary substitutions may fail to terminate, if applied to ill-typed terms. An example is a substitution of $\lambda x.\ x\ x$ for y in $y\ y$. To avoid this problem, hereditary substitutions must be supplied with an additional information that serves as a decreasing metric, so that the substitution may explicitly fail when some condition on the metric is satisfied, rather than diverge. It turns out that an

appropriate metric for this purpose is the type of the substituted variable. In fact, for HTT it suffices to consider only the dependency-free version of this type.

Thus, we start by formally defining the operation of erasing dependencies from a type. Given a type A, its *shape* A^- is the simple type defined as follows.

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

We proceed now to define the syntax of normal terms. As explained in Section 2, normal terms do not contain beta redexes, or equivalently, they do not contain the constructor $M : A$. In this section, we extend the notion further by requiring that terms of a primitive type are in as simple form as possible. For example, an integer expression $\mathsf{s}\ M + N$, is not consider normalized, because it can be reduced to $\mathsf{s}\ (M + N)$. The later form is simpler, because it makes more of the structure apparent. For example, just by looking at the syntax of $\mathsf{s}\ (M + N)$, we know that the result must be larger or equal than one. Similarly, addition $\mathsf{z} + N$ can be reduced to N, multiplication $\mathsf{s}\ M \times N$ can be reduced to $N + (M \times N)$, and multiplication $\mathsf{z} \times N$ can be reduced to z.

The table below summarizes how the syntax of normal forms differs from the general syntax of HTT from Section 2.

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

Here, $M^{-\mathsf{z},\mathsf{s}}$ stands for a normal term syntactically different from z or $\mathsf{s}\ N$ for some N (because, as commented previously, additions involving this kinds of terms may be simplified). Similar comments apply to multiplication and to equality. The normal forms of all the other syntactic categories are build as before, but instead of ordinary intro and elim terms, they hereditarily use the normal intro and elim terms.

Next we define the hereditary substitution operations. Hereditary substitutions are functions that substitute a normal expression for a variable x into a given normal expression. The later expression may be a type, proposition, intro term, elim term, computation or a heap). Each substitution is indexed by a type shape S, which, as we will subsequently show in Theorem 1, is the decreasing metric that prevents the hereditary substitutions from diverging. In the typing judgments in Section 4, S is the shape of the type associated with M and x.

The substitutions are defined by nested induction, first on the structure of the type shape S, and second on the structure of the expression being substituted into. In other words, we either go to a smaller shape (in which case the expression we substitute into may become larger), or the shape remains the same, but the expressions decrease. We write $S_1 \leq S_2$ and $S_1 < S_2$ if the shape S_1 occurs in S_2 (as a proper subexpression in the second case).

If the expressions involved in the hereditary substitution are not well typed, the substitution, though terminating, may fail to return a meaningful term. Hereditary substitutions are *partial functions*, which become total when restricted to well-typed terms (we will establish the later property as a theorem). As conventional when discussing partial functions and 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 both defined and syntactically equal.

We have the following hereditary substitutions, where the superscript indicates the domain of the oper-

ation[2].

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

Substitution $[M/x]^k_S$ into elim terms could either return an elim term K', or an intro term M'. In the later case, we also return a shape $S' \leq S$.

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

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

Notice that the substitution into $K\ N$ 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 associated shape is not smaller than S.

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 taken 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

[2]We use k for elim term, m for intro terms, e for computations, a for types, and h for heaps. We will use $*$ to range over all these syntactic categories.

defined as:

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

As can be seen from the definition, all of the cases are compositional, and the substitution may fail to be defined only if the substitutions into the subterms are undefined. The side conditions $y \notin \mathsf{FV}(M)$ and $y \neq x$ do not cause failure, because they can always be satisfied by alpha renaming y.

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. Because of the compositionality, the substitution may fail only if some of the substitutions into subterms fail, or if the side conditions concerning the shape S are not satisfied. When the substitutions are invoked on well-typed terms, the side conditions about S are always satisfied, so the actual implementation of hereditary substitutions does not need to check for them. 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 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. As before, c varies over the primitive commands for allocation, lookup, mutation, conditionals, loops and fixpoints.

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

The substitution operations into types, propositions 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 normal 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 structure of the index A, and second on the structure of the argument we apply the substitution to. In each case of the definition, we either decrease the type index, 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 beginning of the term. More formally,

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

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 5, 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, propositions 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 $[M/x]^m_S(N_1)$, $[M/x]^m_S(N_2)$ *and* $[M/x]^a_S(A)$ *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, N_2 and A. ■

Lemma 5 (Composition of hereditary substitutions)
Suppose that T ranges over expressions of any syntactic category (i.e., elim terms, intro terms, computations, types, propositions, 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 $y \notin \mathsf{FV}(M_0)$ 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 $[M_0/x]^e_A(F_1) = \langle [M_0/x]^e_A(E_1)/y\rangle_B(F_0)$.*
3. *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, F, and E_0, in the respective cases). ■

4 Type system

The most important feature of the type system of HTT is its use of *canonical forms* in the style of Watkins et al. [45] to facilitate equational reasoning. An expression is in a canonical form, if it is normal (i.e., contains no beta redexes), but is also in eta long form.

A term is in eta long form if all of its introduction subterms are eta expanded. For example, in the context $f{:}(\mathsf{nat} \to \mathsf{nat}) \to (\mathsf{nat} \to \mathsf{nat}) \to \mathsf{nat}, g{:}\mathsf{nat} \to \mathsf{nat}$, the term $f\ g$ has type $(\mathsf{nat} \to \mathsf{nat}) \to \mathsf{nat}$ and is normal, but is not eta long, and hence not canonical. The corresponding canonical form is $\lambda h.\ f\ (\lambda y.\ g\ y)\ (\lambda x.\ h\ x)$. With this notion of canonicity, comparing two terms for equality modulo beta reduction and eta expansion can be done by simply comparing the respective canonical forms for alpha equivalence.

The typing judgments of HTT synthesize the canonical forms of terms in parallel with type checking, and the synthesis employs hereditary substitutions. In Section 3, hereditary substitutions were defined on normal forms, but for the purposes of the typing rules, it is possible to restrict them to canonical forms. We will show later that hereditary substitution over canonical forms produces canonical results.

Typing at the level of terms and computations consists of the following judgments.

$\Delta \vdash K \Rightarrow A\,[N']$	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, so that each of the involved expressions is either considered given as input, or is synthesized as output. For example, $\Delta \vdash K \Rightarrow A\,[N']$ is an inference judgments. If an elimination term K is given, the judgment infers its canonical type A, and synthesizes the canonical form N' of K. Of course, K is arbitrary, i.e., it is not necessarily canonical. On the other hand, $\Delta; \Gamma \vdash M \Leftarrow A\,[M']$ *checks* whether the arbitrary term M matches against the canonical type A. Naturally, M and A are inputs. If the two match, the canonical form M' is synthesized as output.

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

In all the HTT judgments, the variable context Δ is an input and is canonical, in the sense that only types in canonical form may appear in it.

The HTT type theory further requires a judgment to define the truth of propositions from the assertion logic. We employ a multi-sorted variant of the classical sequent calculus.

$$\Delta; \Psi; \Gamma_1 \Longrightarrow \Gamma_2 \quad \text{assuming all propositions in } \Gamma_1 \text{ are true, one of the propositions in } \Gamma_2 \text{ is true}$$

Here Ψ is a list of heap variables, and Γ_1, Γ_2 are lists of propositions. The assertion logic is only concerned with canonical forms, so Γ_1 and Γ_2 only contain propositions in canonical form.

The rest of the HTT judgments deals with formation of various syntactic categories.

$\vdash \Delta\ \mathsf{ctx}$	Δ is a canonical variable context
$\Delta; \Psi \vdash \Gamma\ \mathsf{pctx}$	Γ is a canonical proposition context
$\Delta; \Psi \vdash P \Leftarrow \mathsf{prop}\,[P']$	P is a proposition, and P' is its canonical form
$\Delta \vdash A \Leftarrow \mathsf{type}\,[A']$	A is a type, and A' is its canonical form
$\Delta; \Psi \vdash H \Leftarrow \mathsf{heap}\,[H']$	H is a heap, and H' is its canonical form

In all of these judgments, the canonical forms are the outputs, and all the other involved expressions are inputs.

Variable and proposition context formation. Here we define the judgment $\vdash \Delta\ \mathsf{ctx}$ for variable context formation, and the judgment $\Delta; \Psi \vdash \Gamma\ \mathsf{pctx}$ for proposition context formation. In the second judgment, Δ is a variable context which is implicitly assumed well-formed (i.e. $\vdash \Delta\ \mathsf{ctx}$), and Ψ is a context of heap variables.

$$\frac{}{\vdash \cdot\ \mathsf{ctx}} \qquad \frac{\vdash \Delta\ \mathsf{ctx} \quad \Delta \vdash A \Leftarrow \mathsf{type}\,[A]}{\vdash (\Delta, x{:}A)\ \mathsf{ctx}} \qquad \frac{}{\Delta; \Psi \vdash \cdot\ \mathsf{pctx}} \qquad \frac{\Delta; \Psi \vdash \Gamma\ \mathsf{pctx} \quad \Delta; \Psi \vdash P \Leftarrow \mathsf{prop}\,[P]}{\Delta; \Psi \vdash (\Gamma, P)\ \mathsf{pctx}}$$

Notice that the judgment $\vdash \Delta\ \mathsf{ctx}$ admits only canonical variable contexts by insisting that the type A equals its canonical forms. A similar comment applies to the proposition P in the judgment for proposition contexts.

Type formation The judgment for type formation is $\Delta \vdash A \Leftarrow \mathsf{type}\,[A']$. It is assumed that $\vdash \Delta\ \mathsf{ctx}$. The rules are mostly self-explanatory, and the only intricacy appears in the formation of the computation type $\{P\}x{:}A\{Q\}$. As explained in Section 2, the precondition P may depend on the variable mem that denotes the heap prior to the computation. The postcondition Q depends on heap variables mem and init, where mem is the heap after the computation, and init is the heap before the computation (thus init in the postcondition and mem in the precondition denote the same heap). In addition, the postcondition Q may depend on the variable x which binds the result of the computation.

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

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

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

Proposition formation. The judgment for proposition formation is $\Delta; \Psi \vdash P \Leftarrow \mathsf{prop}\,[P']$. It is assumed that $\vdash \Delta\ \mathsf{ctx}$, and Ψ is a context of heap variables. The proposition P' is the canonical form of P, and is returned as output.

The rules of the judgment describe formation of the primitive propositions Id and Hld, the standard propositional connectives of classical logic, and the quantification over types and heaps.

$$\frac{\Delta; \Psi \vdash H_1 \Leftarrow \mathsf{heap}\,[H_1'] \quad \Delta; \Psi \vdash H_2 \Leftarrow \mathsf{heap}\,[H_2']}{\Delta; \Psi \vdash \mathsf{Hld}(H_1, H_2) \Leftarrow \mathsf{prop}\,[\mathsf{Hld}(H_1', H_2')]}$$

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

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

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

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

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

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

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

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

$$\frac{\Delta \vdash A \Leftarrow \mathsf{type}\,[A'] \quad \Delta; \Psi \vdash H \Leftarrow \mathsf{heap}\,[H'] \quad \Delta \vdash M \Leftarrow \mathsf{nat}\,[M'] \quad \Delta \vdash N \Leftarrow A'\,[N']}{\Delta; \Psi \vdash \mathsf{upd}_A(H, M, N) \Leftarrow \mathsf{heap}\,[\mathsf{upd}_{A'}(H', M', N')]}$$

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 forms 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 derived types, we need two auxiliary functions: $\mathsf{apply}_A(M, N)$ and $\mathsf{expand}_A(N)$.

In $\mathsf{apply}_A(M, N)$, A is a canonical type, 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. Other possibilities cause the function to fail, but they cannot arise if the arguments are well typed.

$\mathsf{apply}_A(K, M)$	$=$	$K\ M$	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

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

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

Now we can present the rest of the typing rules.

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

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

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

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

The elimination rule $\Pi\mathsf{E}$ first synthesizes the type $\Pi x{:}A.\ B$ and the canonical form N' of the function part of the application. Then the synthesized type is used in checking the argument part of the application.

The result type is synthesized using hereditary substitutions in order to remove the dependency of the type B on the variable x. Notice that the arguments to this hereditary substitution are canonical. Finally, we compute the canonical form of the whole application, using the auxiliary function apply.

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

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.

Computations. The judgment $\Delta; P \vdash E \Rightarrow x{:}A.\ Q\,[E']$ generates the strongest postcondition Q of the computation E with respect to the precondition P. In this sense, it is a formalization of the Verification Condition Generator. The postcondition Q may depend on the variable $x{:}A$ that binds the result returned by E. The type A, however, is not synthesized by the judgment; it has to be supplied as an input. Our formulation of the judgment essentially follows the monadic formulation of lax logic by Pfenning and Davies [38], with additional constructs that are specific to our application and that serve to formalize the various effectful operations on heaps.

The second judgment for typing computations has the form $\Delta; P \vdash E \Leftarrow x{:}A.\ Q\,[E']$, and it checks that Q is a postcondition – but not necessarily the strongest one – for E with respect to P.

In both judgments, it is assumed that Δ, A, P and Q are well-formed and canonical, or in other words $\vdash \Delta\ \mathsf{ctx}$, $\Delta \vdash A \Leftarrow \mathsf{type}\,[A]$, $\Delta; \mathsf{init}, \mathsf{mem} \vdash P \Leftarrow \mathsf{prop}\,[P]$, and $\Delta, x{:}A; \mathsf{init}, \mathsf{mem} \vdash Q \Leftarrow \mathsf{prop}\,[Q]$.

Before we can state the rules of these two judgment, 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.

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

Next we define the *strongest postconditions* for the primitive effectful commands of HTT. Given an effectful command c for allocation, lookup or mutation, its strongest postcondition $\mathsf{sp}(c)$ is a proposition that most precisely captures the relationship between the heap init prior to the execution of c and the heap mem obtained after the execution.[3] In the definition of sp we assume that all the involved expressions are canonical.

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

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

[3] Dually, c may be seen as a witness for its strongest postcondition.

Finally, we introduce another propositional connective. Given the propositions P and Q, we write $P; Q$ for the proposition

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

This connective will feature prominently in the computation judgments, because it captures how the heap evolves with the computation. Intuitively, the proposition $P; Q$ holds of the current heap mem, if Q is true of mem, and there exist a prior heap h of which P was true.

We can now present the typing rules for computations. We start with the general rules that correspond to the monadic fragment, and than proceed with the rules for the individual effectful commands.

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

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

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

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

The rule $\mathsf{consequent}$ 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. Thus, the generated postcondition equals the precondition extended with the proposition stating the equality between the canonical forms of x and M.

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

The rule $\{\ \}\mathsf{E}$ corresponds to the monadic bind, and it 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 the assertion logic can establish that the precondition R_1 for K is implied by the precondition P for the composite command, and if the computation E checks against the postcondition for K. The later is taken to be $P; R_2$, because the computation encapsulated by K is executed in the heap of which P holds. The normal form of the whole computation is obtained by invoking the auxiliary function reduce. We repeat here that the type B which is the result type of the computations E and $\mathsf{let}\ \mathsf{dia}\ x = K\ \mathsf{in}\ E$ is an *input* of the typing judgments. By the implicit convention that we outlined at the beginning of this paragraph, B is a well-formed type 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, so we need to existentially abstract x in the postcondition of the conclusions. Similar remark applies to the rules for the specific effectful constructs that we present below.

We start with the primitive commands for allocation, lookup and mutation. Each of the rules first computes the canonical forms of the involved types and terms. Then, in order to compose a command with an arbitrary computation E, this computation needs to check against a precondition obtained as a

postcondition of the command.

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

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

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

In the cases of lookup and mutation, we need to prove in the assertion logic that the location being dereferenced or updated is actually allocated in the current heap, and is initialized with a value of an appropriate type. The sequent

$$\Delta; \mathsf{init}, \mathsf{mem}; P \Longrightarrow \mathsf{seleq}_{A'}(\mathsf{mem}, M', -)$$

that appears in the last two rules serve precisely such a purpose. The sequent is invoked with parametric heap variables init and mem because these may appear in the involved propositions. The free variables of the judgments are, of course, considered universally quantified.

The typing rule for $x = \mathsf{if}_A(M, E_1, E_2)$ first checks the two brunches 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.

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

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.

The looping construct of HTT requires the initial value of the loop accumulator M, the Boolean loop guard $x.\ N$ which depends on the current value of the accumulator x, and the body of the loop $x.\ F$ which is a computation that also depends on the current value of the accumulator. The loop is annotated with the type of the accumulator A, and the loop invariant I. The later is a proposition that relates the beginning heap and the accumulator (init and x) and the ending heap and the accumulator (mem, y) of the loop body. The canonical forms of these expressions are computed as M', N', F' and I'.

$$\frac{\begin{array}{cc}\Delta \vdash A \Leftarrow \mathsf{type}\,[A'] & \\ \Delta \vdash M \Leftarrow A'\,[M'] & \Delta, x{:}A', y{:}A', z{:}A'; \mathsf{init}, \mathsf{mem}; [z/y]I'; [z/x]I' \Longrightarrow I' \\ \Delta, x{:}A' \vdash N \Leftarrow \mathsf{bool}\,[N'] & \Delta, x{:}A', y{:}A'; \mathsf{init}, \mathsf{mem}; I' \Longrightarrow [\mathsf{mem}/\mathsf{init}, y/x]I' \\ \Delta, x{:}A', y{:}A'; \mathsf{init}, \mathsf{mem} \vdash I \Leftarrow \mathsf{prop}\,[I'] & \Delta, x{:}A'; \mathsf{HId}(\mathsf{init}, \mathsf{mem}) \wedge [x/y]I' \wedge \mathsf{Id}_{\mathsf{bool}}(N', \mathsf{true}) \vdash F \Leftarrow y{:}A'.\ I' \\ \Delta; \mathsf{init}, \mathsf{mem}; P \Longrightarrow [M'/x, M'/y]^p_{A'}([\mathsf{mem}/\mathsf{init}]I') & \Delta, y{:}A'; P; [M'/x]^p_{A'}(I') \wedge \mathsf{Id}_{\mathsf{bool}}([y/x]N', \mathsf{false}) \vdash E \Rightarrow z{:}C.\ Q\end{array}}{\Delta; P \vdash y = \mathsf{loop}^I_A(M, x.\ N, x.\ F); E \Rightarrow z{:}C.\ (\exists y{:}A'.\ Q)\,[y = \mathsf{loop}^{I'}_{A'}(M', x.\ N', x.\ F'); E']}$$

Before entering the loop, the rule checks whether the precondition implies the loop invariant. This check is necessary in case the loop is never entered, because the loop guard is not satisfied. Therefore, it is taken in the loop invariant I' that the beginning heap and accumulator match the ending heap and the accumulator, leading to the sequent

$$\Delta; \mathsf{init}, \mathsf{mem}; P \Longrightarrow [M'/x, M'/y]^p_{A'}([\mathsf{mem}/\mathsf{init}]I')$$

In general, when the two heaps and two accumulators are equated in I', as in the above sequent, we obtain a proposition that is a loop invariant in the standard sense, i.e., it is a proposition that holds before and after every iteration of the loop. Such a proposition relates the heap and the accumulator of any one iteration, but by its very definition, it does not relate heaps and accumulators of different iterations.

If I' is to relate different iterations, it needs to satisfy further conditions. If a certain state of heap and accumulator has been reached by the loop, that states becomes the beginning state of the next iteration. The next iteration may or may not be taken, depending on the loop guard, so we must verify that I' holds if such a beginning state is taken as the ending state as well. In other words, we must prove the sequent

$$\Delta, x{:}A', y{:}A'; \mathsf{init}, \mathsf{mem}; I' \Longrightarrow [\mathsf{mem}/\mathsf{init}, y/x]I'$$

We refer to this property as *reflexivity* of I' on the heap and accumulator states reachable by the loop.

We further need *transitivity* of I' so that we can relate the beginning and the ending state of a sequence of iterations. It suffices to consider two iterations. Assume that init and x are the heap and accumulator at the beginning of the first iteration, and that mem and y are the heap and accumulator at the end of the second iteration. Assume further that z denotes the value of the accumulator between the two iterations. Then we need to prove the sequent

$$\Delta, x{:}A', y{:}A', z{:}A'; \mathsf{init}, \mathsf{mem}; [z/y]I'; [z/x]I' \Longrightarrow I'$$

(where the intermediate heap between the two iterations is abstracted by the propositional operator ;).

We further note that the body of the loop is checked with a precondition expressing that the invariant holds and that the loop guard is satisfied. The postcondition requires that the invariant holds between the starting and ending heap and accumulator of the iteration.

The computation following the loop is checked with a precondition expressing that the loop guard is not satisfied, and that loop invariant holds between the current heap and accumulator and the heap and accumulator that existed before entering the loop.

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 $\Pi x{:}A.\ \{R_1\}y{:}B\{R_2\}$, expressing that f is a function whose range is a computation with precondition R_1 and postcondition R_2.

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

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

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

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

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

And weakening and contraction.

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

The propositional connectives do not require much comment either.

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

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

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

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

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

We allow quantification over types and heaps.

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

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

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

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

Of course, we assume the usual proviso that the variables x 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. The reflexivity is the equality-right rule, and the two substitutability rules are equality-left rules. The rules are restricted to primitive propositions q.

$$\frac{}{\Delta; \Psi; \Gamma_1 \Longrightarrow \mathsf{Id}_A(M, M), \Gamma_2}$$

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

Notice that the two substitutability rules above assume that the free variables of M and N are all declared in Δ. This is somewhat restrictive, because we often need to reason about equality of terms with additional free variables. The case in point is extensional equality of functions, where two functions are considered propositionally equal iff their bodies are propositionally equal. For example, the function $\lambda x.\ \lambda y.\ x + y$ should be propositionally equal to $\lambda x.\ \lambda y.\ y + x$, because of commutativity of addition. But function bodies depend on abstracted variables, which are not declared in Δ, and so this equality cannot be derived by reflexivity and substitutability alone. One way around this problem is to introduce an additional axiom for extensionality of functions.

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

where it is assumed that the bound variable x does not appear free in the involved contexts.

Equality over heaps also requires its own rules for reflexivity and substitutability.

$$\frac{}{\Delta; \Psi; \Gamma_1 \Longrightarrow \mathsf{HId}(H, H), \Gamma_2}$$

$$\frac{\Delta; \Psi; \Gamma_1, \mathsf{HId}(H_1, H_2) \Longrightarrow [H_1/h]q, \Gamma_2}{\Delta; \Psi; \Gamma_1, \mathsf{HId}(H_1, H_2) \Longrightarrow [H_2/h]q, \Gamma_2} \qquad \frac{\Delta; \Psi; \Gamma_1, \mathsf{HId}(H_1, H_2) \Longrightarrow [H_2/h]q, \Gamma_2}{\Delta; \Psi; \Gamma_1, \mathsf{HId}(H_1, H_2) \Longrightarrow [H_1/h]q, \Gamma_2}$$

Of course, we require additional rules that axiomatize the structure of heaps. First, heaps that differ up to the permutation of assignments are considered equal.

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

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

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

Heaps are functional – each location in the heap can point to only one term.

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

Empty heaps do not contain any assignments.

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

And if a location appears in a heap, then it is in the heap's domain.

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

Before proceeding further, let us notice that the above axioms do not completely axiomatize the notion of heaps that we have informally outlined in Section 2. What is missing is a way to conclude that $\mathsf{seleq}_A(H, M, N_1) \wedge \mathsf{seleq}_B(H, M, N_2)$ implies $A = B$ and $\mathsf{Id}_A(N_1, N_2)$. The special case when $A = B$ is derivable from the above axiom for heap functionality, but we cannot conclude the general case as we do not have a proposition for type equality (we have a judgment for type equality, which is different). Furthermore, we cannot even rule out the above situation in which the order of updates is unclear, by adopting a rule like the following

$$\frac{A \neq B}{\Delta; \Psi; \Gamma, \mathsf{seleq}_A(H, M, N_1), \mathsf{seleq}_B(H, M, N_2) \Longrightarrow \Gamma_1}\ \mathsf{wrong}$$

For one, in order to have a rule like this, we must define the judgment $A \neq B$, and take care that it is preserved under substitution. A naive definition which simply checks that the decidable judgment $A = B$ does not hold is unsatisfactory in this respect. Indeed, A and B are dependent types, so even if they are syntactically different, some substitution of their free variables may potentially make them equal, and invalidate a whole derivation in which this rule appears.

Fortunately, our monadic judgments generate only postconditions from which it is always clear in which order the updates of the heaps occur. The only way that ambiguities of the above sort may appear in the generated sequents is if they are deliberately supplied by the programmer via type annotations or loop invariants. Of course, even in those situations, such propositions do not introduce unsoundness; they simply render the code in which they appear inapplicable in some situations, and thus less useful than it would have otherwise been. It remains future work to derive an assertion logic which would be more abstract in the above sense. We conjecture that such logic would have to make essential use of type polymorphism.

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.

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

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

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

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.

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

Examples. As a first example, we notice that the following propositional schemes are provable in the sequent calculus, when each free variable H, M, N, M_i, N_i, A, B is substituted with a concrete expression from the appropriate syntactic category.

1. $\Longrightarrow \mathsf{seleq}_A(\mathsf{upd}_A(H, M, N), M, N)$.
2. $\Longrightarrow \mathsf{seleq}_A(H, M_1, N_1) \wedge \neg\mathsf{Id}(M_1, M_2) \supset \mathsf{seleq}_A(\mathsf{upd}_A(H, M_2, N_2), M_1, N_1)$.
3. $\Longrightarrow \mathsf{seleq}_A(H, M, N_1) \wedge \mathsf{seleq}_A(H, M, N_2) \supset \mathsf{Id}_A(N_1, N_2)$
4. $\Longrightarrow \mathsf{seleq}_A(H, M, N) \supset M \in \mathsf{dom}(H)$
5. $\Longrightarrow M \in \mathsf{dom}(H) \wedge N \notin \mathsf{dom}(H) \supset \neg\mathsf{Id}_{\mathsf{nat}}(M, N)$

The first two sequents are variants of the familiar McCarthy axioms for seleq and upd [28]. The third sequent states the functionality of heaps. The fourth relates the propositions seleq and indom. The fifth sequent concludes inequality between locations M and N if a heap H can be found which differentiates between the locations. Proving the above sequents is rather simple, and mostly requires a simple expansion of the definition of seleq and indom in the sequent calculus, followed by a straightforward application of rules for propositional equality and heap equality.

As a second example, let us try to typecheck, step-by-step the example from Section 2. We annotate the various control points in the computation with propositions (enclosed in braces) that hold at that point. We denote by P and Q respectively, the precondition and the postcondition listed below.

```
double' : [x:nat] [y:nat]
           {seleq(mem, x, -) && seleq(mem, y, -) && ~~Id(x, y)} r : nat
           {!!v1:nat. !!v2:nat.
               (seleq(init, x, v1) && seleq(init, y, v2)) ==>
                 Id(r, v1+v2) && seleq(mem, x, 2*v1) && seleq(mem, y, 2*v2)} =
      fn x. fn y. dia ({P}
                       w1 = [x];
                       {P1}
                       [x] = w1 + w1;
                       {P2}
                       w2 = [y];
                       {P3}
                       [y] = w2 + w2;
                       {P4}
                       w1 + w2)
```

We assume that the seleq's and upd's in the annotations above are all indexed by the type nat, as the variables x and y stand for pointers to integer values. The propositions P_i are generated by the monadic judgment as follows.

1. $P_1 = P; \mathsf{HId}(\mathsf{mem}, \mathsf{init}) \wedge \mathsf{seleq}(\mathsf{mem}, x, w_1)$
2. $P_2 = P_1; \mathsf{HId}(\mathsf{mem}, \mathsf{upd}(\mathsf{init}, x, w_1 + w_1)$
3. $P_3 = P_2; \mathsf{HId}(\mathsf{mem}, \mathsf{init}) \wedge \mathsf{seleq}(\mathsf{mem}, y, w_2)$
4. $P_4 = P_3; \mathsf{HId}(\mathsf{mem}, \mathsf{upd}(\mathsf{init}, y, w_2 + w_2)$

Typechecking requires that the following sequents be shown. These sequents essentially correspond to the verification condition of the function double' (for the sake of brevity, we omit the type annotations, and the contexts of variables and heaps from these sequents).

1. $P \Longrightarrow \mathsf{seleq}(\mathsf{mem}, x, -)$ so that x can be dereferenced in the first command.

2. $P_1 \Longrightarrow \mathsf{seleq}(\mathsf{mem}, x, -)$ so that x can be updated in the second command.

3. $P_2 \Longrightarrow \mathsf{seleq}(\mathsf{mem}, y, -)$ so that y can be dereferenced in the third command.

4. $P_3 \Longrightarrow \mathsf{seleq}(\mathsf{mem}, y, -)$ so that y can be updated in the fourth command

5. $\exists w_1, w_2{:}\mathsf{nat}.\ P_4 \wedge \mathsf{Id}(r, w_1 + w_2) \Longrightarrow Q$ so that Q is a valid postcondition.

In order to see that the above sequents from are satisfied, we need to expand the definition of the propositional operator ";" in each of P_i. After some simplification we obtain the following logical equivalents.

1. $P_1 = \mathsf{seleq}(\mathsf{mem}, x, w_1) \wedge \mathsf{seleq}(\mathsf{mem}, y, -) \wedge \neg\mathsf{Id}(x, y)$

2. $P_2 = \exists h{:}\mathsf{heap}.\ \mathsf{seleq}(h, x, w_1) \wedge \mathsf{seleq}(h, y, -) \wedge \neg\mathsf{Id}(x, y) \wedge \mathsf{HId}(\mathsf{mem}, \mathsf{upd}(h, x, w_1 + w_1)$

3. $P_3 = \exists h{:}\mathsf{heap}.\ \mathsf{seleq}(h, x, w_1) \wedge \mathsf{seleq}(h, y, w_2) \wedge \neg\mathsf{Id}(x, y) \wedge \mathsf{HId}(\mathsf{mem}, \mathsf{upd}(h, x, w_1 + w_1)$

4. $P_4 = \exists h{:}\mathsf{heap}.\ \mathsf{seleq}(h, x, w_1) \wedge \mathsf{seleq}(h, y, w_2) \wedge \neg\mathsf{Id}(x, y) \wedge \mathsf{HId}(\mathsf{mem}, \mathsf{upd}(\mathsf{upd}(h, x, w_1 + w_1), y, w_2 + w_2)$

Now the validity of the sequents from the verification condition is obvious. The most interesting is the case of $\exists w_1, w_2{:}\mathsf{nat}.\ P_4 \wedge \mathsf{Id}(r, w_1 + w_2) \Longrightarrow Q$. To see that this is valid, we remove the existential and universal quantifiers, and then apply functionality of heaps to establish that the universally quantified variables v_1 and v_2 in Q must be equal to w_1 and w_2, respectively. Then the result follows using the above simplification of P_4 and the validity of McCarthy axioms about seleq and upd established previously.

As a third example, let us consider a simple function that loops through the integers 1 thorough n, computes their sum in a heap allocated memory, and eventually returns the computed sum. We assume given the primitive ordering relations on integers ($\leq$, $<$, $>$, $\geq$), and their implementations in the language of terms as boolean functions (`<=`, `<`, `>`,`>=` respectively). The relations and the boolean functions are tied through the assertion logic. For example, we assume that `x <= y` equals `true` iff $x \leq y$ is provable as a proposition of the assertion logic. We explicitly provide in the form of brace-enclosed comments, the propositions P_i that the typing judgments generate at each location of the program. We will also denote by Q the proposition that is given as the postcondition in the type of the function below.

```
sumfunc : [n : nat] {true} r : nat {??sum:nat. Hid(mem, upd(init, sum, r)) &&
                                                ~~indom(init, sum) &&
                                                Id(2 * r, n * (n + 1))} =
    fn n. dia(sum = alloc(0);
              {P1}
              y = loop (1, x. x <= n,
                        x. {P2}
                           s = [sum];
                           {P3}
                           [sum] = s + x;
                           {P4}
                           x + 1);
              {P5}
              t = [sum];
              {P6}
              t)
```

The variable `sum` above is a pointer to an integer, and thus seleq's and upd's for this variable are all indexed by nat. The function sumfunc first allocates a memory cell, initialized with 0 and binds the address to the variable `sum`. The loop starts with the accumulator set to 1, and exits when the accumulator reaches $n + 1$. Thus, the accumulator enumerates the numbers to be summed up. We use the local variable x to refer to the accumulator in the boolean guard of the loop and in the loop body. The loop body adds the next number

in the enumeration (here x) to `sum`, and returns $x+1$ as the value to be bound to the accumulator variable x in the next iteration. The return result of the whole program is the content of the location `sum`.

The loop invariant is given by the proposition I bellow. The invariant relates the accumulator x, and the heap init at the beginning of the iteration, with the new accumulator y and the new heap mem, at the end of the iteration.

$$I = \exists v_1, v_2{:}\mathsf{nat}.\ \mathsf{Id}(2{*}v_1{+}x, x{*}x){\wedge}\mathsf{seleq}(\mathsf{init},\mathsf{sum},v_1){\wedge}\mathsf{Id}(2{*}v_2{+}y, y{*}y){\wedge}\mathsf{mem} = \mathsf{upd}(\mathsf{init},\mathsf{sum},v_2){\wedge}y \leq n{+}1$$

It is easy to establish that I satisfies the reflexivity and transitivity properties discussed in Section 4, i.e. that the sequents $I \Longrightarrow [\mathsf{mem}/\mathsf{init}, y/x]I$ and $[z/y]I; [z/x]I \Longrightarrow I$ are provable.

The propositions generated at each program point during typechecking are as follows.

1. $P_1 = \mathsf{HId}(\mathsf{mem}, \mathsf{upd}(\mathsf{init}, \mathsf{sum}, 0) \wedge \mathsf{sum} \notin \mathsf{dom}(\mathsf{init})$
2. $P_2 = \mathsf{HId}(\mathsf{init}, \mathsf{mem}) \wedge [x/y]I \wedge x \leq n$
3. $P_3 = P_2; \mathsf{HId}(\mathsf{mem}, \mathsf{init}) \wedge \mathsf{seleq}(\mathsf{mem}, \mathsf{sum}, s)$
4. $P_4 = P_3; \mathsf{HId}(\mathsf{mem}, \mathsf{upd}(\mathsf{init}, \mathsf{sum}, s + x))$
5. $P_5 = P_1; [1/x]I \wedge y > n$
6. $P_6 = P_5; \mathsf{HId}(\mathsf{mem}, \mathsf{init}) \wedge \mathsf{seleq}(\mathsf{mem}, \mathsf{sum}, t)$

And the sequents to be proved as part of typechecking (and which hence constitute the verification condition) are:

1. $P_1 \Longrightarrow [1/x, 1/y][\mathsf{mem}/\mathsf{init}]I$, so that the loop invariant holds before entering the loop.
2. $P_2 \Longrightarrow \mathsf{seleq}(\mathsf{mem}, \mathsf{sum}, -)$, so that sum can be dereferenced.
3. $P_3 \Longrightarrow \mathsf{seleq}(\mathsf{mem}, \mathsf{sum}, -)$, so that sum can be updated
4. $\exists s{:}\mathsf{nat}.\ P_4 \wedge \mathsf{Id}(y, x+1) \Longrightarrow I$, so that the loop invariant holds at the end of the loop.
5. $P_5 \Longrightarrow \mathsf{seleq}(\mathsf{mem}, \mathsf{sum}, -)$, so that sum can be dereferenced.
6. $\exists y, t{:}\mathsf{nat}.P_6 \wedge \mathsf{Id}(r, t) \Longrightarrow Q$, so that the postcondition given in the type is valid.

The sequents are not hard to prove, after expanding the definitions of the connective ";", and establishing some relatively simple arithmetic facts, such as commutativity, associativity of $+$ and $\times$, distributivity of $\times$ over $+$, and implications such as $y \leq n+1 \wedge y > n \supset \mathsf{Id}(y, n)$ and $\mathsf{Id}(2*v+x, x*x) \supset \mathsf{Id}(2*(v+x)+(x+1), (x+1)*(x+1))$.

For the last example, let us rewrite the above function using recursion, rather than loops. We use the more customary notation `if M then E else F` instead of $\mathsf{if}(M, E, F)$.

```
sumfunc' : [n : nat] {true} r : nat {HId(mem, init) && Id(2 * r, n * (n + 1))} =
    fn n. dia(y = fix(f. x.
                        {P0}
                        t = if x > n then {P1} 0
                            else
                                 {P2}
                                 let dia s = f (x + 1)
                                 in
                                     {P3}
                                     s + x
                                 end;
                        {P4}
                        t,
                    1);
              {P5}
              y)
```

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

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

essentially expressing that the recursive function computes the sum $y = x + (x+1) + \cdots + n$. Let us denote the postcondition from this type as Q, i.e.

$$Q = \mathsf{HId}(\mathsf{mem},\mathsf{init}) \wedge (x \leq n \supset \mathsf{Id}(2*y+x*x, n*(n+1)+x)) \wedge (x > n \supset \mathsf{Id}(y,0))$$

The propositions generated by the typing at the various points in the computation are:

1. $P_0 = \mathsf{HId}(\mathsf{init},\mathsf{mem}) \wedge \mathsf{true}$
2. $P_1 = P_0; \mathsf{HId}(\mathsf{init},\mathsf{mem}) \wedge x > n$
3. $P_2 = P_0; \mathsf{HId}(\mathsf{init},\mathsf{mem}) \wedge x \leq n$
4. $P_3 = P_2; [x+1/x, s/y]^p_{\mathsf{nat}}(Q)$
5. $P_4 = (P_1 \wedge \mathsf{Id}(t,0)) \vee (\exists s{:}\mathsf{nat}.\ P_3 \wedge \mathsf{Id}(t, s+x))$
6. $P_5 = P_0; [1/x]^p_{\mathsf{nat}}(Q)$

The sequents to be proved are:

1. $P_2 \Longrightarrow \mathsf{true}$, so that the computation obtained from $f\ (x+1)$ can be activated.
2. $P_4 \wedge \mathsf{Id}(y,t) \Longrightarrow Q$, so that the body of the recursive function satisfies the specified postcondition.
3. $P_5 \wedge \mathsf{Id}(r,y) \Longrightarrow \mathsf{HId}(\mathsf{mem},\mathsf{init}) \wedge \mathsf{Id}(2*r, n*(n+1))$, so that the function sumfunc' satisfies the specified postcondition.

The most interesting is the second sequent. To see that this sequent is valid, we simplify the antecedent $P_4 \wedge \mathsf{Id}(y,t)$ using some simple arithmetic properties, and the reflexivity, symmetry and transitivity of equality on terms and heaps, to obtain the equivalent

$$\begin{aligned} P_4 \wedge \mathsf{Id}(y,t) \quad = \quad & \mathsf{HId}(\mathsf{init},\mathsf{mem}) \wedge \\ & (x > n \supset y = 0) \wedge \\ & (x < n \supset \mathsf{Id}(2*y+x*(x+1), n*(n+1)+2*x)) \wedge \\ & (\mathsf{Id}(x,n) \supset \mathsf{Id}(y,n)) \end{aligned}$$

This is then easily seen to entail Q.

To establish the third sequent, we simplify the proposition $P_5 \wedge \mathsf{Id}(r, y)$ into an equivalent

$$P_5 \wedge \mathsf{Id}(r, y) = \mathsf{HId}(\mathsf{mem}, \mathsf{init}) \wedge (1 \leq n \supset \mathsf{Id}(2 * r, n * (n + 1))) \wedge (1 > n \supset \mathsf{Id}(r, 0))$$

which is easily seen to entail the required $\mathsf{HId}(\mathsf{mem}, \mathsf{init}) \wedge (2 * r, n * (n + 1))$.

5 Properties

In this section we present the basic properties of HTT, leading up to the substitution principles. The development is organized as follows. 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. This canonical fragment of HTT is crucial because it is foundation of our semantics. In a sense, canonical forms carry the real meaning of HTT expressions, while the general (i.e., non-canonical forms) may be viewed as simply a syntactic convenience for writing programs. Finally, we lift the substitution principles to full HTT, thus essentially establishing that substitution commutes with normalization – substituting and then normalizing produces the same result as normalizing first, and then subsisting hereditarily the obtained canonical forms.

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

Theorem 6 (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. ■

Clearly, in the above theorem, an oracle deciding the sequents from the assertion logic may potentially be substituted with a proof that serves as a *checkable* witness of the sequent's validity. In the spirit of Proof-carrying code [31], it is possible to embed such proofs into HTT terms and computations, which we plan to do in the future work. Then the above theorem will loose the attribute "relative"; typechecking will include proof checking for sequents, and thus become decidable.

There is another way to interpret Theorem 6. 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 7 (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 $\Delta, x{:} \vdash J$*
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 8 (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 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 9 (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. As explained in Section 4, canonical forms are beta reduced, and eta expanded expressions. They are also outputs of our typing judgments. For example, in $\Delta \vdash N \Leftarrow A\,[N']$, N' is a canonical form of N. In the following lemmas, we characterize canonical forms via the typing judgments, as those expressions that equal the form returned as the output of their own typechecking. For example, K is a canonical elimination form if $\Delta \vdash K \Rightarrow A\,[K]$, and N is a canonical introduction form if $\Delta \vdash N \Leftarrow A\,[N]$.

In the next lemma, we explore the properties of hereditary substitution of variable expansions, as a first step towards proving a more general 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 10 (Properties of variable expansion)

1. *If* $\Delta, x{:}A, \Delta_1 \vdash K \Rightarrow B\,[K]$, *then* $[\mathsf{expand}_A(x)/x]^k_A(K)$ *exists, and*
   (a) *if* $[\mathsf{expand}_A(x)/x]^k_A(K) = K'$ *is an elim term, then* $K' = K$
   (b) *if* $[\mathsf{expand}_A(x)/x]^k_A(K) = N' :: S$ *is an intro term, then* $N' = \mathsf{expand}_B(K)$, *and* $S = B^-$.
2. *If* $\Delta, x{:}A, \Delta_1 \vdash N \Leftarrow B\,[N]$, *then* $[\mathsf{expand}_A(x)/x]^m_A(N) = N$.
3. *If* $\Delta, x{:}A, \Delta_1; P \vdash E \Leftarrow y{:}B.\ Q\,[E]$, *then* $[\mathsf{expand}_A(x)/x]^e_A(E) = E$.

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

Proof: First of all, we notice that the function $\mathsf{expand}_A(-)$ only really depends on the shape A^-, rather than on the whole type A. Now, we can proceed to prove the above proposition, by mutual nested induction, first on the structure of A and then on the structure of the expression involved. In other words, we will invoke the induction hypotheses either with a strictly smaller shape, but possibly larger expression, or with the same shape, but strictly smaller expression. The characteristic case of the lemma is 1(b), and we present its proof in more detail.

In this case, by Lemma 2 it must be $\mathsf{head}(K) = x$. When $K = x$ the statement is obvious, so we consider the case when $K = L\ N$ for some elim term $L \Rightarrow \Pi y{:}B_1.\ B_2$ and intro term $N \Leftarrow B_1$ where $B = [N/y]^a_{B_1}(B_2)$. We know that $\mathsf{head}(L) = x$ as well, so by induction hypothesis on L, $[\mathsf{expand}_A(x)/x]^k_A(L) = \mathsf{expand}_{B_1^- \to B_2^-}(L) = \lambda y.\ \mathsf{expand}_{B_2^-}(L\ \mathsf{expand}_{B_1^-}(y))$ for some fresh variable y. By Termination of hereditary substitutions (Theorem 1), we also know that $B_1^- \to B_2^- \leq A^-$. Applying the induction hypothesis again, but this time on N, obtains $[\mathsf{expand}_A(x)/x]^k_A(N) = N$.

Now, $[\mathsf{expand}_A(x)/x]^k_A(K) = [\mathsf{expand}_A(x)/x]^k_A(L\ N) = [N/y]^m_{B_1^-}(\mathsf{expand}_{B_2^-}(L\ \mathsf{expand}_{B_1^-}(y))) :: B_2^-$, if this hereditary substitution does not fail. The substitution indeed does not fail, because the last expression is defined simultaneously with, and is equal to $\mathsf{expand}_{B_2^-}(L\ [N/y]^m_{B_1^-}(\mathsf{expand}_{B_1^-}(y)))$, since the variable y does not appear in L. But, by induction hypothesis 7 on $B_1^- < A^-$, this is defined and equals $\mathsf{expand}_{B_2^-}(L\ N)$. As $B_2^- = B^-$, and $K = L\ N$, we obtain the required $[\mathsf{expand}_A(x)/x]^k_A(K) = \mathsf{expand}_B(K) :: B^-$. ■

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 11 (Identity principles)

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

2. *If* $\Delta; \Psi; \Gamma_1, \mathit{Id}_B(M, N) \Longrightarrow [M/x]^p_B(P), \Gamma_2$ *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* $\Delta; \Psi; \Gamma_1, \mathit{Id}_B(M, N) \Longrightarrow [N/x]^p_B(P), \Gamma_2$.

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

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

Proof: By simultaneous induction on the structures of P and A. The interesting cases of the first statement are when $P = \forall x{:}A.\ Q$ or $P = \exists x{:}A.\ Q$. In these cases, we first eliminate the quantifier on one side by introducing a fresh variable $x{:}A$, and then instantiate the quantifier on the other side with $\mathsf{expand}_A(x)$. By Lemma 10, $[\mathsf{expand}_A(x)/x]^p_A(Q) = Q$, so the result follows by the induction hypothesis on Q. Of course, before we can conclude the derivation, we must show that $\mathsf{expand}_A(x) \Leftarrow A$. This follows by inductively appealing to the fourth statement of the lemma, which is justified because A is a strict subexpressions of P.

Statements 2 and 3 employ statement 1 and the rule of cut in order to break down the propositions $[M/x]^p_B(P)$ and $[H_1/h]P$ into their primitive parts, before the parts can be reconstituted into $[N/x]^p_B(P)$ and $[H_2/h]P$, respectively. In the case when P is a quantified proposition, we also inductively appeal to statement 4. The condition on well-formedness of $[N/x]^p_B(P)$ in statement 2 will be shown as redundant,

after we establish the substitution principles for canonical forms in one of the following theorems. However, we will use the identity principles in the proof of the substitution principles, so this condition must be made explicit. No such check is needed in the statement 3, because we have already established the heap substitution principles in Lemma 9.

In statement 4, the most interesting case is when $A = \{P\}x{:}B\{Q\}$. In this case, $\mathsf{expand}_A(K) = \mathsf{dia}\ (\mathsf{let\ dia}\ y = K\ \mathsf{in}\ \mathsf{expand}_B(y))$. In order for this term to check against A, the typing rules require that the following sequents be proved:

1. $\Delta; \mathsf{init}, \mathsf{mem}; \mathsf{Hld}(\mathsf{init}, \mathsf{mem}) \wedge P \Longrightarrow P$
2. $\Delta; \mathsf{init}, \mathsf{mem}; \exists y{:}B.\ \exists h{:}\mathsf{heap}.\ \mathsf{Hld}(\mathsf{init}, h) \wedge [h/\mathsf{mem}]P \wedge [h/\mathsf{init}][y/x]Q \wedge \mathsf{Id}_B(\mathsf{expand}_B(x), \mathsf{expand}_B(y)) \Longrightarrow Q$

The first sequent shows that the precondition for K is satisfied at the point in the computation where K is executed. The sequent is easy to derive, by applying the left rule for conjunction and then inductively appealing to the first statement, which is justified because P is a strict subexpression of A.

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 $\mathsf{Hld}(\mathsf{init}, \mathsf{mem}) \wedge P$ 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 well-formed and canonical. To derive this sequent, we first apply the left rules for existential quantification and conjunction. After that, the result follows by properties of variable expansion (Lemma 10) for the proposition Q, and then by inductively appealing to the statements 3 and then 2 in order to deal with the substitutions into Q. The two inductive steps are justified, because Q is a strict subexpression 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. We also include here the property on the preservation of history, which essentially states that a computation does not depend on how the heap in which it executes may have been obtained. Thus, if the computation has a precondition P and a postcondition Q, these can be composed with an arbitrary proposition R into a new precondition $R; P$ and a new postcondition $R; Q$.

Lemma 12 (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* $\Delta; (R; P) \vdash E \Leftarrow x{:}A.\ (R; Q)\,[E']$.

Proof: To prove weakening of consequent, 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 $E = \mathsf{let\ dia}\ y = K\ \mathsf{in}\ F$, and we present its proof in more detail. In this case, from the typing of E we obtain: $\Delta \vdash K \Rightarrow \{R_1\}y{:}B\{R_2\}\,[N']$ where $\Delta; \mathsf{init}, \mathsf{mem}; P \Longrightarrow R_1$, and $\Delta, y{:}B; P; R_2 \vdash F \Rightarrow x{:}A.\ S\,[F']$ 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. $\Delta; \mathsf{init}, \mathsf{mem}; R \Longrightarrow R_1$, and
2. $\Delta, y{:}B; (R; R_2) \vdash F \Rightarrow x{:}A.\ S'\,[F']$ 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 $P \Longrightarrow R_1$ obtained from the typing of E. To derive (2), we first observe that $\Delta, y{:}B; P; R_2 \vdash F \Rightarrow x{:}A.\ S\,[F']$ implies $\Delta, y{:}B; P; R_2 \vdash F \Leftarrow x{:}A.\ S\,[F']$, by the inference rule $\mathsf{consequent}$, and using the identity principle (Lemma 11) to establish $S \Longrightarrow S$. It is also easy to show that the sequent $\Delta; \mathsf{init}, \mathsf{mem}; R; R_2 \Longrightarrow P; R_2$

is derivable, after first expanding the definition of the propositional connective ";". Now, by induction hypothesis on F, we have $\Delta, y{:}B; (R; R_2) \vdash F \Leftarrow x{:}A.\ S\,[F']$.

The later means that there exists a proposition S' such that $\Delta, y{:}B; R; R_2 \vdash F \Rightarrow x{:}A.\ S'\,[F']$ where $\Delta, y{:}B, x{:}A; \mathsf{init}, \mathsf{mem}; S' \Longrightarrow S$. But then we can also have $\Delta, x{:}A; \mathsf{init}, \mathsf{mem}; \exists y{:}B.\ S' \Longrightarrow \exists y{:}B.\ S$. 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 $(\Delta; (R; P) \vdash E \Leftarrow x{:}A.\ (R; Q)\,[E'])$, we need to establish that:

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

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

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

At this point, we would like to emphasize the relationship between our preservation of history and the frame rule of Separation logic [33, 41, 34]. In Separation logic, if E is a computation satisfying the Hoare triple $\{P\}\ E\ \{Q\}$, we can use the frame rule to derive $\{P * C\}\ E\ \{Q * C\}$. Here C is an arbitrary proposition and $*$ is a propositional connective defined so that $P * Q$ holds of a heap if the heap can be split into two *disjoint* parts so that P holds of the first and Q holds of the second part. Thus, in essence, the frame rule states that the parts of the heap that are not touched by E actually remain invariant.

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

We are now prepared to state the substitution principles for the canonical fragment of HTT.

Lemma 13 (Canonical 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., $\Delta, \Delta_1'; \mathsf{init}, \mathsf{mem} \vdash$*

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

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

7. *If* $\Delta, x{:}A, \Delta_1; \Psi; \Gamma_1 \Longrightarrow \Gamma_2$*, 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* $x \notin \mathit{FV}(B, Q)$*, 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. To abbreviate the notation, we write $(-)'$ instead of $[M/x]^*_A(-)$.

In this case, by the typing derivation of E, we know that $\Delta, x{:}A, \Delta_1 \vdash K \Rightarrow \{R_1\}z{:}C\{R_2\}\,[K]$, and $\Delta, x{:}A, \Delta_1; \mathsf{init}, \mathsf{mem}; P \Longrightarrow R_1$, and $\Delta, x{:}A, \Delta_1, z{:}C; (P; R_2) \vdash F \Leftarrow y{:}B.\ Q\,[F]$ and $\Delta, \Delta_1'; \mathsf{Hld}(\mathsf{init}, \mathsf{mem}) \wedge R_1' \vdash E_1 \Leftarrow z{:}C'.\ R_2'\,[E_1]$. We also know by Theorem 1 that $(\{R_1\}z{:}C\{R_2\})^- \leq A^-$, 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 12), $\Delta, \Delta_1'; (P'; \mathsf{Hld}(\mathsf{init}, \mathsf{mem}) \wedge R_1') \vdash E_1 \Leftarrow z{:}C'.\ (P'; R_2')\,[E_1]$. From the typing of F, by induction hypothesis, $\Delta, \Delta_1', z{:}C'; (P'; R_2') \vdash F' \Leftarrow y{:}B'.\ Q'\,[F']$. By induction hypothesis on $C'^- = C^- < A^-$, using the above two judgments, $\Delta, \Delta_1'; (P'; \mathsf{Hld}(\mathsf{init}, \mathsf{mem}) \wedge R_1') \vdash E' \Leftarrow y{:}B'.\ Q'\,[E']$.

Finally, by induction hypothesis on the derivation of the sequent $P \Longrightarrow R_1$ we obtain $P' \Longrightarrow R_1'$, and therefore also $P' \Longrightarrow P'; \mathsf{Hld}(\mathsf{init}, \mathsf{mem}) \wedge R_1'$. Now we can apply strengthening of the precedent (Lemma 12) 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 13.

Lemma 14 (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 just present the proof of statement 3 in case $N = K$ is an elimination term, and hence the last inference rule in the typing derivation is the rule $\Rightarrow\Leftarrow$.

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 11), $\Delta \vdash \mathsf{expand}_A(M') \Leftarrow A\,[\mathsf{expand}_A(M')]$. ■

Finally, we can state and prove the substitution principles on general, rather than only on canonical forms. 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 substitution principles.

Lemma 15 (General 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: By simultaneous induction on the structure of the principal derivations. We present only the proof of statement 1, in the case when $K = L\ N$.

In this case, from the typing derivation of K, we have $\Delta, x{:}A', \Delta_1 \vdash L \Rightarrow \Pi y{:}B_1.B_2\,[M'_1]$, and $\Delta, x{:}A', \Delta_1 \vdash N \Leftarrow B_1\,[M'_2]$, and $B = [M'_2/y]^a_{B_1}(B_2)$, and $N' = \mathsf{apply}_{B_1}(M'_1, M'_2)$. By induction hypotheses on L and N, we have $\Delta, [M'/x]_A(\Delta_1) \vdash [M : A/x]L \Rightarrow \Pi y{:}[M'/x]^a_A(B_1).[M'/x]^a_A(B_2)\,[[M'/x]^m_A(M'_1)]$, and $\Delta, [M'/x]_A(\Delta_1) \vdash [M : A/x]N \Leftarrow [M'/x]^a_A(B_1)\,[[M'/x]^m_A(M'_2)]$. We can now use the typing rule $\Pi\mathsf{E}$ on the term $[M : A/x]K = [M : A/x](L\ N)$ to synthesizes the type $B' = [[M'/x]^m_A(M'_2)/y]^a_{B_1}([M'/x]^a_A(B_2))$, and the canonical form $O' = \mathsf{apply}_{[M'/x]^a_A(B_1)}([M'/x]^m_A(M'_1), [M'/x]^m_A(M'_2))$.

In order to prove the case, we need to show that $B' = [M'/x]^a_A(B)$, and $O' = [M'/x]^m_A(N')$. The first equality follows by composition of hereditary substitutions (Lemma 5), because $B' = [M'/x]^a_A([M'_2/y]^a_{B_1}(B_2)) = [M'/x]^a_A(B)$. The second equality follows relatively easily, by the canonical substitution principles (Lemma 13) once we establish that M', M'_1 and M'_2 are canonical and well-typed. Of course, the last fact is a direct consequence of the involution of canonical forms (Lemma 14), thus concluding the proof of the case. ■

6 Operational semantics

In the previous sections, we have defined the type system of HTT with the associated notions of proof equality, normalization and canonical forms. We now proceed to develop the view of HTT proofs as programs that can be executed. For that purpose, in this section we define the call-by-value left-to-right structured operational semantics for HTT. The operational semantics is defined on typed expressions. The outcome of the evaluation does not depend on types, but defining the semantics on typed expressions simplifies the statements and the proofs of the appropriate preservation and progress theorems, which we also carry out. Preservation and progress theorems together establish that HTT is sound with respect to evaluation. Our proofs are relative to the assumed soundness of the HTT assertion logic. The soundness of the assertion logic is not established here and is left for future work.

We start the definition of the operational semantics, by presenting the syntactic categories related to evaluation.

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

Values. The definition of values is rather standard. It includes the boolean and integer constants, unit constant ( ), function values and suspended computations $\mathsf{dia}\ E$. We adopt the definition where the body of a function value is itself unevaluated, which is the standard approach in functional programming. But, we note that this is not strictly necessary in HTT, as HTT functions are pure and can thus be normalized without the change in their semantics.

We use v to range over values, and l to range over integers when they are used as pointers into the heap.

Heap values. Heap values are assignments from integers to values, where each assignment is indexed by a type. Heap values are a special subset of heaps defined in Section 2 in the same way in which values are a subset of terms. However, we here make the distinction between heap values and heaps more apparent, by adopting a somewhat different syntax for heap values, which will result in a more parsimonious definition of the operational semantics. So, we write $\cdot$ instead of emp, and $\chi, l \mapsto_A v$ instead of $\mathsf{upd}_A(H, l, v)$. The essential difference between the two notations is that in heap values each location can be assigned at most once. These properties are made explicit in the judgment $\vdash \chi :\ \mathsf{heapval}$, which checks if χ is a well-formed heap value.

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

Notice how the judgment requires that the index type A 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 8 – so that l can be checked for membership in $\mathsf{dom}(\chi)$. Heap values are considered equal up to the reordering of their assignments.

We will frequently need to convert value heaps into heap canonical forms, so we introduce the following conversion function.

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

We will also write $\Delta; \chi \vdash P$ as short for $\Delta; \mathsf{mem}; \mathsf{Hld}(\mathsf{mem}, [\![\chi]\!]) \Longrightarrow P$. This judgment will essentially establish that the proposition P holds of the heap value 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.

To reduce clutter, we will usually omit the terminating $\cdot$ from non-empty continuations. Given two continuations $\kappa_1 = x_1{:}A_1.E_1; \cdots; x_n{:}A_n.E_n$ and $\kappa_2 = y_1{:}B_1.F_1; \cdots; y_m{:}B_m.F_m$, we write $\kappa_1; \kappa_2$ for the concatenation

$$\kappa_1; \kappa_2 = x_1{:}A_1.E_1; \cdots; x_n{:}A_n.E_n; y_1{:}B_1.F_1; \cdots; y_m{:}B_m.F_m$$

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. The correspondence with computations can actually be made more precise, as 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

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

We express the nested computation above as a control expression because the control expressions make it easier to formulate the call-by-value operational semantics. 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, to composition must evaluate E (which is not required in the equational theory) before binding the result to x to be used in F. This is formally described by creating the control expression $x.\ F; \cdot \triangleright E$; or in other words, we first push $x.\ F$ onto the continuation, and proceed to evaluate E.

The typing judgment for control expressions has the form $\Delta; P \vdash \kappa \triangleright E \Leftarrow x{:}A.\ Q$, and its meaning is similar to the one for computations: if executed in a heap of which the proposition P holds, the control expression $\rho = \kappa \triangleright E$ results with a value $x{:}A$ and a heap of which the proposition Q holds. 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 usual syntactic conventions: $\vdash \Delta\ \mathsf{ctx}$, $\Delta; \mathsf{init}, \mathsf{mem} \vdash P \Leftarrow \mathsf{prop}\,[P]$, $\Delta \vdash A \Leftarrow \mathsf{type}\,[A]$, and $\Delta, x{:}A; \mathsf{init}, \mathsf{mem} \vdash Q \Leftarrow \mathsf{prop}\,[Q]$.

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

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

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 16 (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. We only present the proof of the first statement in case $\kappa = \kappa'; z{:}C.F$, for some type $\Delta \vdash C \Leftarrow \mathsf{type}\,[C']$ and $z \notin \mathsf{FV}(A, Q)$. In this case, by the typing of $\kappa \triangleright E$, we have that $\Delta; P \vdash \kappa' \triangleright E \Leftarrow z{:}C'.\ S$ for some proposition S, and $\Delta, z{:}C'; S \vdash F \Leftarrow x{:}A.\ Q\,[F']$. By induction hypothesis on κ', we obtain $\Delta; P \vdash E \Leftarrow y{:}B.\ R\,[E']$, and if $\Delta_1; P_1 \vdash \kappa_1 \triangleright E_1 \Leftarrow y{:}B.\ R$ for some Δ_1 extending Δ, then $\Delta_1; P_1 \vdash \kappa_1; \kappa' \triangleright E_1 \Leftarrow z{:}C'.\ S$. Because Δ_1 extends Δ, we also have $\Delta_1, z{:}C'; S \vdash F \Leftarrow x{:}A.\ Q\,[F']$ by weakening the typing derivation for F. Using the last two derivation as premises in the typing rule for composite control expressions, we derive $\Delta_1; P_1 \vdash \kappa_1; \kappa'; z{:}C.F \triangleright E_1 \Leftarrow x{:}A.\ Q$. As $\kappa = \kappa'; z{:}C.F$, this is precisely what we set out to prove. ■

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

The type information for an abstract machine should specify the type of the result that the evaluation of the abstract machine returns, as well as a description of the heap in which the evaluation terminates (if it does terminate). Our typing judgment for abstract machines captures exactly this information. It has the form $\chi, \kappa \triangleright E \Leftarrow x{:}A.\ Q$, where A is the result type and Q is a proposition describing the resulting heap. Of course, we assume the usual conventions of well-formedness of all the involved elements. In particular, $\vdash \chi : \ \mathsf{heapval}$, $\cdot \vdash A \Leftarrow \mathsf{type}\,[A]$ and $x{:}A; \mathsf{init}, \mathsf{mem} \vdash Q \Leftarrow \mathsf{prop}\,[Q]$.

The judgment is defined as follows. We say that $\vdash \chi, \kappa \triangleright E \Leftarrow x{:}A.\ Q$ iff $\cdot; P \vdash \kappa \triangleright E \Leftarrow x{:}A.\ Q$, where $P = \mathsf{Hld}(\mathsf{init}, \mathsf{mem}) \wedge \mathsf{Hld}(\mathsf{mem}, [\![\chi]\!])$. In other words, we first convert the heap χ into a canonical proposition P which uniquely defines χ, and then check that the control expression $\kappa \triangleright E$ is well-typed with respect to P, A and Q.

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

The inference rules of the evaluation judgments are mostly straightforward, with several peculiarities arising because of the syntactic structure of HTT. For example, HTT is dependently typed, so we must substitute into types and not only into terms. More importantly, we must make special provisions for evaluating the constructs for type ascription $M : A$, which may be iterated as in $M : A_1 : \cdots : A_n$. We will adopt the following conventions regarding type ascription. When evaluating introduction terms, the ascription $v : A$ where v is a value is considered a redex, and is reduced by simply returning v. When evaluating elimination terms, we want the evaluation to terminate with an annotated value $v : A$ because the type A is needed for evaluating beta redexes. However, we make sure that only one type is returned, rather than a whole iterated sequence of ascriptions $v : A_1 : \cdots : A_n$.

We start by presenting the rules for evaluating elimination terms.

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

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

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

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 a redex of the form $v : A$, we further evaluate by removing the type annotation.

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

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.

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

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

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

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

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

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

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

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

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

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

$$\frac{E' = \langle [v : A/x]F/z : A\rangle(y = \mathsf{loop}^I_A(z, x.\ N, x.\ F); y)}{\chi, \kappa \triangleright x = \mathsf{loop}^I_A(v, x.\ N, x.\ F); E \hookrightarrow_e \chi, \kappa \triangleright x = \mathsf{if}_A([v : A/x]N, E', v); E}$$

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

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

The preservation theorem, as usual, states that the evaluation step on a well-typed expression results with well-typed result. In the pure fragment of HTT (i.e., in the case of elim and intro terms), there is an additional claim that evaluation preserves the canonical form of the evaluated term. This should not be surprising. Evaluation of functions is compatible with normalization because it is based on beta reduction. Evaluation on primitive types is compatible with normalization, because values of primitive types exactly correspond to closed canonical forms (Lemma 8).

Theorem 17 (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* $\alpha_0 \hookrightarrow_e \alpha_1$ *and* $\vdash \alpha_0 \Leftarrow x{:}A.\ Q$, *then* $\vdash \alpha_1 \Leftarrow x{:}A.\ Q$.

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 16). The only involved cases are the evaluation of $\mathsf{let\ dia}$, loop and fixpoint constructs. We present here the case when $\alpha_0 = \chi_0, \kappa_0 \triangleright \mathsf{let\ dia}\ y = \mathsf{dia}\ F : \{R_1\}y{:}B\{R_2\}\ \mathsf{in}\ E$.

In this case, $\alpha_1 = \chi_0, (y{:}B.E; \kappa_0) \triangleright F$. Let $P = \mathsf{Hld}(\mathsf{init}, \mathsf{mem}) \wedge \mathsf{Hld}(\mathsf{mem}, [\![\chi_0]\!])$. From the typing of α_0, we know $P \vdash \kappa_0 \triangleright \mathsf{let\ dia}\ y = \mathsf{dia}\ F : \{R_1\}y{:}B\{R_2\}\ \mathsf{in}\ F \Leftarrow x{:}A.\ Q$. By the replacement lemma, $P \vdash \mathsf{let\ dia}\ y = \mathsf{dia}\ F : \{R_1\}y{:}B\{R_2\}\ \mathsf{in}\ F \Leftarrow z{:}C.\ S$, 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 $\mathsf{mem} \vdash R_1 \Leftarrow \mathsf{prop}\,[R_1']$ and $\mathsf{init}, \mathsf{mem}; R_2 \Leftarrow \mathsf{prop}\,[R_2']$ and

1. $\mathsf{init}, \mathsf{mem}; P_1 \Longrightarrow R_1'$
2. $\mathsf{Hld}(\mathsf{init}, \mathsf{mem}) \wedge R_1' \vdash F \Leftarrow y{:}B'.\ R_2'$
3. $y{:}B'; P; R_2' \vdash E \Leftarrow z{:}C.\ S.$

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 12), $P; \mathsf{Hld}(\mathsf{init}, \mathsf{mem}) \wedge R_1' \vdash F \Leftarrow y{:}B'.\ (P; R_2')$. Then, because (1) implies $P \Longrightarrow P; \mathsf{Hld}(\mathsf{init}, \mathsf{mem}) \wedge R_1'$, we can strengthen the precedent and obtain $P \vdash F \Leftarrow y{:}B'.\ (P; R_2')$. 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. ■

The last theorem in this section is the progress theorem. Progress theorem 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 18 (Heap soundness)
The assertion logic of HTT is heap sound iff for every heap value χ, the existence of a derivation for the sequent $\cdot; \mathsf{mem}; \mathsf{Hld}(\mathsf{mem}, [\![\chi]\!]) \Longrightarrow \mathsf{seleq}_A(\mathsf{mem}, l, -)$ implies that $l \mapsto_A v \in \chi$, for some value v.

Heap soundness is important, because it relates the assertion logic to the actual heaps encountered during evaluation. For example, if the assertion logic can prove that a location l is allocated, then the evaluation rules should be able to look up or update the value that l points to. This is a crucial property, because the typing rules for computation from Section 4 rely on the assertion to logic to establish the preconditions for various effectful commands. If the assertion logic does not correspond to evaluation, then the type system cannot be sound.

We do not prove in this paper that our assertion logic is heap sound, and we leave that proof as an important future work. The standard way to establish soundness of logics defined by sequent calculi is to prove the appropriate notion of *cut elimination*. In our assertion logic, that is not quite possible, for several reasons. First, we have extra-logical axioms that describe the properties of heaps and propositional equality, and the statement of cut elimination must usually be weaken in the presence of extra-logical axioms [15]. Second, we have the primitive type of integers with induction principles, so that our logic includes the whole of Peano arithmetic. Thus, the proof of heap soundness will necessarily have to go beyond cut elimination, and will probably have to apply the techniques of ordinal arithmetic and transfinite induction as developed in Gentzen's classical proof of the soundness of Peano arithmetic [14, 43, 40]. This is why we felt that the proof of heap soundness is beyond the scope of the current paper, especially since in the future we plan to significantly extend the assertion logic with second-order features and with inductive definitions, as we discuss in Section 7.

We now state the Progress theorem, and present some cases of its proof. In the light of the above comments, the Progress theorem should be consider as a *relative soundness*, because it relies on the unproved heap soundness of the assertion logic.

Theorem 19 (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: By straightforward case analysis on the involved expressions, employing inversion on the typing derivations. The interesting cases arise in the proof of the third statement, when E_0 is one of the effectful commands for allocation, lookup or update, as these are the only evaluation rules for abstract machines which depend on premises about typing or about the shape of the heap values. We present the case when $E_0 = (y = [l]_B; E)$, which is representative.

In this case, from the typing of the abstract machine $\chi_0, \kappa_0 \triangleright E_0 \Leftarrow x{:}A.\ Q$, we know that $P \vdash \kappa_0 \triangleright E_0 \Leftarrow x{:}A.\ Q$, where $P = \mathsf{Hld}(\mathsf{init}, \mathsf{mem}) \wedge \mathsf{Hld}(\mathsf{mem}, [\![\chi_0]\!])$. By replacement (Lemma 16), $P \vdash y = [l]_B; E \Leftarrow z{:}C.\ S$ for some type C and proposition S. But then, by the typing rules for lookup, $\cdot \vdash B \Leftarrow \mathsf{type}\,[B']$ and $P \Longrightarrow \mathsf{seleq}_{B'}(\mathsf{mem}, l, -)$. Now, by the assumed heap soundness, we know that there exists a value v of type B' such that $l \mapsto_{B'} v \in \chi_0$ (we can ignore the heap init in the proposition P, as it is stated as equal to mem). Thus, the premises of the evaluation rule for lookup are met, and we can make the transition $\chi_0, \kappa_0 \triangleright y = [l]_B; E \hookrightarrow_e \chi_0, \kappa_0 \triangleright [v : B/x]E$. ■

7 Future and related work

In this section we discuss several issues that we plan to address in future work, and how they relate to the previous work on Hoare logic and type theory.

Completeness. The standard notion of completeness in Hoare logic is Cook completeness [10], which essentially requires that the assertion logic can express the strongest postconditions for any computation. It should be clear that the present version of HTT is *not* Cook complete. For example, say that we want to write a stateful function for computing factorials. In order to do this, we first need an inductive predicate in the assertion logic which defines the relation between an integer and its factorial. Without this predicate, we cannot write an appropriate specification, and thus the appropriate type for the factorial function.

Cook completeness is essentially a property of programs which are not annotated with specifications, unlike in HTT, where each loop is associated with an invariant, and each computation is associated with explicit pre- and postconditions. Thus, the study of Cook completeness for HTT will require defining the operation of type erasure, and possibly considering type and invariant inference for unannotated terms.

On a related note, it is well known that a Hoare-like logic for a language with higher-order and local functions, recursion, static scope and global variables, and with first-order assertion logic cannot be Cook complete [9]. Cook completeness in these cases requires stronger assertion logics [16].

Recently, a notion of observational completeness for a Hoare-like logic of higher-order functions have been proposed by Honda et al. [21, 5]. Their notion is based on observational equivalence of programs: a logic is observationally complete if two programs are observationally equivalent whenever they satisfy the same set of pre- and postconditions. While we do not know whether a formal account on the relationship between Cook completeness and observational completeness exists, it seems that observational completeness is strictly weaker. A logic may be observationally complete simply because the set of contexts in which observations are made is restricted in some way, perhaps by typing rules. In Cook completeness, such restrictions cannot be imposed.

Extensions and soundness of the assertion logic. As we illustrated in the above discussion of completeness and the factorial function, it seems that adding inductive predicates to HTT is our most pressing future work. This is also the first step before we can consider heap-allocated data structures like linked lists or trees. Specifications of this kind of structures require inductive predicates to describe the shape of the heap, and they also require inductive data structures like lists and trees [33, 41]. There is an interesting theoretical question on whether the addition of inductive predicates to a first-order assertion logic can make HTT relatively complete.

Once data structures are added, it becomes important to consider polymorphic quantification over types and propositions. We plan to consider predicative higher-rank type polymorphism [36]. Impredicative polymorphism may be hard to reconcile with dependent types of HTT, because hereditary substitutions

cannot be defined prior to typechecking. Thus, typechecking and equational reasoning in the presence of impredicative polymorphism will necessarily have to be intertwined.

Quantification over propositions, especially existential quantification, may make it possible to specify that several functions – which possibly share a part of the heap – preserve a common invariant, without leaking that invariant to the outside world. Thus, from the outside, all these functions may be seen as pure.

Of course, once all these additions have been made, an important question to answer concerns the soundness and heap soundness of the assertion logic, as discussed in Section 6.

Encapsulation of effects and equational reasoning. Monadic languages are well-known for enforcing a programming style in which the sequential ordering of program's effects is apparent from the source code. But sometimes this explicit sequentialization may be unnecessary. For example, when it is clear that two effectful computations operate on disjoint pieces of the heap, they can be executed out of order or in parallel. In such cases, it is desirable that the source code does not explicitly sequentialize these two computations.

The currently standard approach to this kind of functionality in the programming language Haskell is to add a primitive function runST [24]. This function takes a computation encapsulated in the Haskell's state monad and executes it in a freshly allocated temporary memory. The result of the execution is returned, and the temporary memory is deallocated.

Unfortunately, runST is incompatible with normalization and canonical forms, and thus cannot be reconciled with dependent types of HTT. Another way to provide the functionality of runST may be based on assertion logics for reasoning about disjointness of heaps, much like in Separation logic. For example, suppose that we have a propositional connective $P * Q$ which holds of a heap if the heap can be split into two *disjoint* parts, so that P holds of the first part and Q holds of the second part. Then, it may be possible to consider a construct $E_1 \parallel E_2$ for *parallel* composition of computations E_1 and E_2, with a typing rule which ensures that E_1 and E_2 require disjoint heaps, and can thus be executed in parallel. The typing rule for parallel composition may be as below.

$$\frac{\Delta; P \Longrightarrow R_1 * R_2 \quad \Delta; R_1 \vdash E_1 \Rightarrow x_1{:}A_1.\ S_1 \quad \Delta; R_2 \vdash E_2 \Rightarrow x_2{:}A_2.\ S_2 \quad \Delta, x_1{:}A_1, x_2{:}A_2; P; (S_1 * S_2) \vdash E \Rightarrow y{:}B.\ Q}{\Delta; P \vdash x_1, x_2 = E_1 \parallel E_2; E \Rightarrow y{:}B.\ (\exists x_1{:}A_1, x_2{:}A_2.\ Q)}$$

In the above rule, it may not be possible to automatically infer the propositions R_1 and R_2, in which case manual annotations may be required. In addition, even if the heap described by P can be split into two disjoint parts, it is not necessarily true that propositions like R_1 and R_2 exist, perhaps because P abstracts over some information which is needed in both R_1 and R_2.

In any case, a construct like this brings up an important question: when should two computations be considered definitionally equal. Currently, our definitional equality on computations is very weak as it allows only for the monadic equations. But this need not necessarily be so. As shown by Watkins et al. [45], it is quite possible to add new equations which would, for example, equate effectful computation that differ only in the permutation of non-interfering commands. Of course, as discussed earlier, contextual equivalence of the HTT effectful computations is not axiomatizable, so the set of allowed equations will never be complete.

Deallocation. We have not considered deallocation in this paper, mainly in order to simplify the development. Adding deallocation requires a new heap construct for freeing a location from the heap, and axiomatizing how this construct interacts with seleq and upd. We do not expect this addition to be too difficult, because our assertion logic can already reason about dangling pointers. Indeed, locations in HTT are simply integers, and the assertions can therefore depend on locations that do not point to a currently allocated heap fragment.

Generation of space-optimal postconditions and temporal assertion logics. As observed in Section 4, the current formulation of judgment for computations may synthesize postconditions that are needlessly large. The culprit is the typing rule for the conditional $\mathsf{if}(M, E_1, E_2)$, where the postconditions generated for E_1 and E_2 may share a significant amount of information because they are computed starting from the same precondition. A possible space-optimal reformulation should reflect more of the propositional

structure into the typing judgments. For example, instead of only one precondition P, the typing judgment for computations may consider a whole stack $P_1; \cdots ; P_n$. This approach may require an assertion logic in which the connective ";" is primitive, rather than defined. Such an assertion logic may also make it possible to formalize the temporal aspects of computations, so that we can say that the computation never goes through a heap with certain properties, or always reaches a heap with certain properties, etc. Currently, our quantification over heaps does not require that the abstracted heaps are in any way related with the past or the future of the computation.

Explicit proofs. Finally, we would like to certify the sequents used in HTT by providing explicit proofs, which would become parts of terms and computations. This will make HTT suitable as a foundational framework for Proof-carrying code [31]. The main challenge here is to ensure that proofs are represented in as compact form as possible, with as much reuse of lemmas and common structure. At the same time, it is important that proofs do not change the notion of definitional equality of types. They should be *equationally irrelevant* [20, 37].

8 Conclusions

In this paper we presented a Hoare Type Theory (HTT), which is a dependent type theory that reconciles higher-order functions with effectful computations. In HTT, the effects involve non-termination and state with first-class locations and aliasing. To the best of our knowledge, this is the first type theory that allows effectful computations to appear in types.

The main idea of HTT is familiar from functional programming: we use monads to separate pure from effectful computations. The pure fragment consists of higher-order functions and admits equational reasoning by beta reduction and eta expansion. The monadic fragment consists of the commands for allocation, lookup and mutation of references, as well as conditionals, loops, and recursion. It admits equational reasoning by monadic laws, as well as reasoning by pre- and postconditions in the style of Hoare logic. Effectful computations can be coerced into the pure fragment by monadic encapsulation. The type of such an encapsulated computation is the Hoare triple $\{P\}x{:}A\{Q\}$. This type constructor can be viewed as a monad indexed by the propositions P and Q. Because monadically encapsulated effectful computations are pure terms, they can appear in types and assertions, and as a consequence, HTT can potentially specify many interesting relations over effectful computations.

There are several characteristic aspects of the HTT formulation of Hoare triples, which differentiate it from the usual Hoare logics. For example, Hoare logic start with the definition of a programming language, and then provides the specification and reasoning methods on top of the language. In HTT, we develop the programming language, the specifications and the reasoning simultaneously. By the Curry-Howard isomorphism, the programming language corresponds to the proof terms of HTT, the specifications are types, and the reasoning methods involve the definition of type equality.

Another difference between Hoare logic and HTT involves the use of logic variables. In Hoare logic, logic variables stand for some value shared by the pre- and postcondition of a program. From the point of view of type theory, logic variables are somewhat problematic. They have global scope, and a distinctly second-class status, as they can appear in assertions but not in the programs. In HTT we abandon logic variables, and specify the relations between the beginning and the ending heap of a computation in a different way. In HTT a postcondition in the type of Hoare triples explicitly depends on two heap variables: init which denotes the heap at the beginning, and mem which is the heap at the end of the computation. By using these two variables and the primitive propositions that describe heap properties, the postcondition can specify arbitrary relations and sharing between the two heaps.

With the important task of relating the beginning and ending heaps being ascribed to postconditions, one may wonder whether preconditions in HTT are obsolete. This is not the case. The precondition describes what the starting heap needs to satisfy so that the evaluation of the computation makes progress and never gets stuck. Thus, the precondition is relevant for both terminating and non-terminating computations, while the postcondition is only important for terminating computations.

In this paper, we developed the basic metatheoretic results about HTT, which include the substitution principles, as well as some basic properties of computations and Hoare triples, like weakening of the consequent, strengthening of the precedent, and the preservation of history (please see Lemma 12 in Section 5). The last property is important as it concisely captures the modular semantics of HTT. Preservation of history states that a computation only depends on the heap in which its execution starts, rather than on how that heap has been reached. Moreover, if the computation does not modify a part of the heap, the history of that part remains unchanged. As a consequence, specifications in HTT need describe only the relative changes between the beginning and the ending heap of a computation, as the rest is assumed invariant. This aspect of HTT is similar to the Frame rule of Separation logic [33, 41, 34].

In order to establish that the proof terms of HTT can be viewed as programs, we formulated a call-by-value operational semantics, and proved the Preservation and Progress theorems, thus establishing the soundness of HTT with respect to evaluation. The soundness result is relative to the soundness of the assertion logic, which we assumed, but did not prove here.

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] N. Alechina, M. Mendler, V. de Paiva, and E. Ritter. Categorical and Kripke semantics for Constructive S4 modal logic. In L. Fribourg, editor, *International Workshop on Computer Science Logic, CSL'01*, volume 2142 of *Lecture Notes in Computer Science*, pages 292–307, Paris, 2001. Springer.

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

[4] P. N. Benton, G. M. Bierman, and V. de Paiva. Computational types from a logical perspective. *Journal of Functional Programming*, 8(2):177–193, March 1998.

[5] 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.

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

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

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

[11] T. Coquand and G. Huet. The calculus of constructions. *Information and Computation*, 76(2–3):95–120, February/March 19988.

[12] 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.

[13] M. Fairtlough and M. Mendler. Propositional lax logic. *Information and Computation*, 137(1):1–33, 1997.

[14] G. Gentzen. New version of the consistency proof for elementary number theory. In M. E. Szabo, editor, *The Collected Papers of Gerhard Gentzen*, pages 252–286. North-Holland, 1969.

[15] J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*. Cambridge University Press, 1989.

[16] J. Y. Halpern. A good Hoare axiom system for an Algol-like language. In *Symposium on Principles of Programming Languages, POPL'84*, pages 262–271, 1984.

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

[18] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, Jan. 1993.

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

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] J. Launchbury and S. L. Peyton Jones. State in Haskell. *Lisp and Symbolic Computation*, 8(4):293–341, 1995.

[25] 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.

[26] 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.

[27] 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.

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

[29] J. C. Mitchell. *Foundations for Programming Languages*. MIT Press, 1996.

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

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

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

[33] 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.

[34] 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.

[35] S. Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, April 2003.

[36] S. L. Peyton Jones, D. Vytiniotis, S. Weirich, and M. Shields. Practical type inference for arbitrary-rank types. Under consideration for publication in Journal of Functional Programming, 2005.

[37] F. Pfenning. Intensionality, extensionality, and proof irrelevance in modal type theory. In *Symposium on Logic in Computer Science, LICS'01*, pages 221–230, Boston, Massachusetts, 2001.

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

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

[40] W. Pohlers. *Proof Theory: An Introduction*. Number 1407 in Lecture Notes in Mathematics. Springer-Verlag, 1989.

[41] 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.

[42] 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.

[43] G. Takeuti. *Proof Theory*. North-Holland, 1975.

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

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

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

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