Abstract Predicates and Mutable ADTs in Hoare Type Theory

Aleksandar Nanevski Amal Ahmed Greg Morrisett
Harvard University
{aleks,amal,greg}@eecs.harvard.edu

Lars Birkedal
IT University of Copenhagen
birkedal@itu.dk

September 16, 2006

Abstract

Hoare Type Theory (HTT) combines a dependently typed, higher-order language with monadically-encapsulated, stateful computations. The type system incorporates pre- and post-conditions, in a fashion similar to Hoare and Separation Logic, so that programmers can modularly specify the requirements and effects of computations within types.

This paper extends HTT with quantification over abstract predicates (i.e., higher-order logic), thus embedding into HTT the Extended Calculus of Constructions. When combined with the Hoare-like specifications, abstract predicates provide a powerful way to define and encapsulate the invariants of private state; that is, state which may be shared by several functions, but is not accessible to their clients. We demonstrate this power by sketching a number of abstract data types and functions that demand ownership of mutable memory, including an idealized custom memory manager.

1 Introduction

The combination of *dependent* and *refinement* types provides a powerful form of specification for higher-order, functional languages. For example, using dependency and refinements, we can specify the signature of an array subscript operation as:

$$\texttt{sub} : \forall\alpha.\Pi x{:}\texttt{array}\,\alpha.\Pi y{:}\{i{:}\texttt{nat} \mid i < x.\texttt{size}\}.\alpha$$

The type of the second argument, y, refines the underlying type `nat` using a predicate which, in this case, ensures that y is a valid index for the array x.

The advantages of dependent types have long been recognized, but integrating them into practical programming languages has proven challenging for two reasons: First, because types can include terms and predicates, type-comparisons require some procedure for determining equality of terms and implication of predicates, which is generally undecidable. Second, the presence of any computational effect, including non-termination, exceptions, access to a store, or I/O, can quickly render a dependent type system unsound.

Both problems can be addressed by severely restricting dependencies (as in for instance DML [44].) But the goal of our work is to try to realize the full power of dependent types for specification of effectful programming languages. To that end, we have been developing the foundations of a language that we call *Hoare Type Theory* or HTT [32, 33]. To address the problem of decidability, HTT takes the usual type-theoretic approach of providing two notions of equality: *Definitional* equality is limited but decidable and thus can be discharged automatically. In contrast, *propositional* equality allows for much coarser notions of equivalence that may be undecidable, but in general, will require a witness.

To address the problem of effects, HTT starts with a pure, dependently typed core language and augments it with an indexed monad type of the form $\{P\}x{:}A\{Q\}$. This type encapsulates and describes effectful computations which may diverge or access a mutable store. The type can be read as a Hoare-like partial correctness assertion stating that if the computation is run in a world satisfying the pre-condition P, then if it terminates, it will return a value x of type A and be in a world described by Q.

Importantly, Q can depend not only upon the return value x, but also the initial and final stores making it possible to capture *relational* properties of stateful computations in the style of *Standard Relational Semantics* well-known in the research on Hoare Logic for first-order languages [13].

In our previous work, we described a polymorphic variant of HTT where predicates were restricted to first-order logic and used the McCarthy array axioms to model memory. The combination of polymorphism and first-order logic was sufficient to encode the connectives of separation logic, making it possible to use concise, small-footprint specifications for programs that mutate state. We established the soundness of the type system through a novel combination of denotational and operational techniques.

HTT was not the first attempt to define a program logic for reasoning about higher-order, effectful programs. For instance, Berger et al. [4] and Krishnaswami [19] have each constructed program logics for higher-order, ML-like languages. However, we believe that HTT has two key advantages over these and other proposed logics: First, HTT supports *strong* (i.e., type-varying) updates of mutable locations. In contrast, all other program logics for higher-order programs (of which we are aware) require that the types of memory locations are *invariant*. This restriction makes it difficult to model stateful protocols as in the Vault programming language [9], or low-level languages such as TAL [31] and Cyclone [17] where memory management is intended to be coded within the language.

The second advantage is that HTT integrates specifications into the type system instead of having a separate program logic. We believe this integration is crucial as it allows programmers (and meta-theorists) to reason contextually, based on types, about the behavior of programs. Furthermore, the integration makes it possible to uniformly abstract over re-usable program components (e.g., libraries of ADTs and first-class objects.) Indeed, through the combination of its dependent type constructors, polymorphism, and indexed monad, HTT already provides the underlying basis of first-class, ML-style modules with specifications.

However, truly reusable components require that their *internal invariants* are appropriately abstracted. That is, the interfaces of components and objects need to include not only abstract types, but also abstract specifications. Thus it is natural to attempt to extend HTT with support for predicate abstraction (i.e., higher-order logic), which is the focus of this paper.

More specifically, we describe a variant of HTT that includes the Extended Calculus of Constructions [23] (modulo minor changes described in Section 9). This allows terms, types, and predicates to all be abstracted within terms, types, and predicates respectively.

There are several benefits of this extension. First, higher-order logic can formulate almost any predicate that may be encountered during program verification, including predicates defined by induction and coinduction. Second, we can reason *within* the system, about the equality of terms, types and predicates, *including abstract types and predicates.* In the previous version of HTT [32], we could only reason about the equality of terms, whereas equality on types and predicates was a judgment (accessible to the typechecker), but *not* a proposition (accessible to the programmer). The extension endows HTT with the basis of *first-class* ML-style modules [24, 14] that can contain types, terms, *and axioms.* Internalized reasoning on types is also important in order to fully support strong updates of mutable locations. Third, higher-order logic can *define* many constructs which in the previous version had to be primitive. For instance, the definition of heaps can now be encoded within the language, thus simplifying some aspects of the meta theory.

However, the most important benefit, which we consider the main contribution of this paper is that *abstraction over predicates suffices to represent private state within types.* Private state can be hidden from the clients of a function or a datatype, by existentially abstracting over the state invariant. Thus, libraries for mutable state can provide precise specifications, yet have sufficient abstraction mechanisms that different implementations can share a common signature.

At the same time, hiding local state by using such a standard logical construct like existential abstraction, ensures that we can scale the language to support dependent types and thus also expressive specifications. This is in contrast to most of the previous work on type systems for local state, which has been centered around the notion of *ownership types* (see for example [1] and [20] and the extensive references therein), where supporting dependency may be significantly more difficult, if not impossible.

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

2 Overview

The type system of HTT is structured so that typechecking can be split into two independent phases. In the first phase, the typechecker ignores the expressive specifications in the form of pre- and postconditions, and only checks that the program satisfies the underlying simple types. At the same time, the first phase generates the verification conditions that will imply the functional correctness of the program. In the second phase, the generated conditions are discharged.

We emphasize that the second phase may be implemented in many different ways, offering a range of correctness assurances. For example, the verification conditions may be discharged in interaction with the programmer, or checked against a supplied formal proof, or passed to a theorem prover which can automatically prove or disprove some of the conditions, thus discovering bugs. The verification conditions may even be ignored if the programmer does not care about the benefits (and the cost) of full correctness, but is satisfied with the assurances provided by the first phase of typechecking.

In order to support the split into phases, HTT supports two notions of equality. The first phase uses *definitional equality* which is weak but decidable, and the second phase uses *propositional* equality which is strong, but may undecidable.

The split into two different notions of equality leads to a split in the syntax of HTT between the fragment of pure terms, containing higher-order functions and pairs, and the fragment of impure terms, containing the effectful commands for memory lookup and strong update as well as the conditionals, and recursion (memory allocation and deallocation can be defined). The expressions from the effectful fragment can be coerced into the pure one by monadic encapsulation [29, 30, 18, 41]. The encapsulation is associated with the type of Hoare triples $\{P\}x{:}A\{Q\}$, which are monads indexed by predicates P and Q [32].

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

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

The type constructors include the primitive types of booleans and natural numbers, the standard constructors 1, Π and Σ for the unit type, and dependent products and sums, respectively, but also the Hoare triples $\{P\}x{:}A\{Q\}$, and the subset types $\{x{:}A.\, P\}$. The Hoare type $\{P\}x{:}A\{Q\}$ classifies effectful computations that may execute in any initial heap satisfying the assertion P, and either diverge, or terminate returning a value $x{:}A$ and a final heap satisfying the assertion Q. The subset type $\{x{:}A.\, P\}$ classifies all the elements of A that satisfy the predicate P. We adopt the standard convention and write $A{\to}B$ and $A{\times}B$ instead of $\Pi x{:}A.\, B$ and $\Sigma x{:}A.\, B$ when B does not depend on x.

To support abstraction over types and predicates, HTT introduces constructors mono and prop which are used to classify types and predicates respectively. These types are a standard feature in the Extended Calculus of Construction (ECC) [23] and Coq [8, 5]. In fact, HTT may be viewed as a fragment of ECC, extended primitively with the monadic type of Hoare triples.

HTT supports only predicative type polymorphism [28, 34], by differentiating *small types*, which do not admit type quantification, from *large types* (or just types for short), which can quantify over small types only. For example, the polymorphic identity function can be written as

$$\lambda\alpha.\lambda y.y : \Pi\alpha{:}\mathsf{mono}.\Pi y{:}\alpha.\alpha$$

but α ranges over only small types. The restriction to predicative polymorphism is crucial for ensuring that during type-checking, normalization of terms, types, and predicates terminates [32]. Note, however, that "small" Hoare triples $\{P\}x{:}\tau\{Q\}$ and subset types $\{x{:}\tau.\,P\}$, where P and Q (but not τ) may contain type quantification can safely be considered small types. This is because P and Q are *refinements*, i.e. they do not influence the underlying semantics and the equational reasoning about terms. A term of some Hoare or subset type has the same operational behavior no matter which refining assertion is used in its type.

Using the type mono, HTT can *compute* with small types as if they were data. This is one of the basic properties needed to encode ML-style modules [24, 14]. For example, if $x{:}\mathsf{mono}\times(\mathsf{nat}\rightarrow\mathsf{nat})$, then the variable x may be seen as a structure declaring a small type and a function on nats. The expression $\mathsf{fst}\ x$ extracts the small type.

Using the type prop, HTT can compute with and abstract over assertions as if they were data. The types mono and prop together are the main technical tool that we will use to hide the local state of computations, while revealing only the invariants of the local state.

Terms. The terms are classified as introduction or elimination terms, according to their standard logical properties. The split facilitates equational reasoning and bidirectional typechecking [38]. The terms are not annotated with types, as in most cases, the typechecker can infer them. When this is not the case, the constructor $M : A$ may supply the type explicitly. This construct also switches the direction in the bidirectional typechecking.

HTT features the usual terms for lambda abstraction and applications, pairs and the projections, as well as natural numbers, booleans and the unit element. The introduction form for the Hoare types is $\mathsf{dia}\ E$[1] which encapsulates the effectful computation E, and suspends its evaluation. The constructor in is a coercion from A into a subset type $\{x{:}A.\,P\}$, and out is the opposite coercion.

The definitional equality of HTT equates all the syntactic constructs up to alpha, beta and eta reductions on terms, but does not admit the reshuffling of the order of effectful commands in $\mathsf{dia}\ E$, or reasoning by induction (the later is allowed for the propositional equality).

Terms also include small types τ, σ and assertions P, Q, R which are the elements of mono and prop respectively. We interchangeably use the terms assertions, *propositions* or *predicates*. HTT does not currently have any constructors to inspect the structure of such elements. They are used solely during typechecking and theorem proving, and can be safely erased (in a type-directed fashion) before program execution.

We use P, Q, R to range over not only propositions, but also over lambda expressions which produce an assertion (i.e., predicates). This is apparent in the syntax for Hoare triples, where we write $\{P\}x{:}A\{Q\}$, but P and Q are actually propositional functions that abstract over the beginning and the ending heap of the computation that is classified by the Hoare triple.

Finally, the constructor $\mathsf{eta}_K\ L$ records that the term L has to be eta expanded with respect to the small type K. This construct is needed for the internal equational reasoning during type checking. It is not supposed to be used in the source programs, and we will not provide operational semantics for it. We discuss this construct further in the Section 4 on hereditary substitutions, and in the Section 5 on the type system of HTT.

Example Consider a simple SML-like program below, where we assume a free variable $x{:}\mathsf{nat\ ref}$.

$$\begin{array}{l} \mathsf{let\ val}\ f = \lambda y{:}\mathsf{unit}.\, x := !x + 1; \\ \qquad\qquad\qquad\qquad\quad \mathsf{if}\ (!x = 1)\ \mathsf{then}\ 0\ \mathsf{else}\ 1 \\ \mathsf{in}\ f\ () \end{array}$$

We translate this program into HTT as follows.

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

[1]Monads correspond to the $\diamond$ ("diamond") modality of modal logic, hence we use dia to suggest the connection.

There are several characteristic properties of the translation that we point out. First notice that all the stateful fragments of this program belong to the syntactic domain of computations. Each computation can intuitively be described as a semi-colon-separated list of imperative commands of the form $x = c$, ending with a return value. Here the variable x is immutable, as is customary in modern functional programming, and its scope extends to the right until the end of the block enclosed by the nearest dia.

Aside from the primitive commands, there are two more computation constructors. The computation M is a pure computation which immediately returns the value M. The computation $\mathsf{let\ dia}\ x = K\ \mathsf{in}\ E$ executes the encapsulated computation K, binds the obtained result to x and proceeds to execute E. These two constructs are directly related to monads [37], and correspond to the monadic *unit* and *bind*, respectively. We choose this syntax over the standard monadic syntax, because it makes eta-expansions for computations somewhat simpler [33].

The commands $!_\tau\, M$ and $M :=_\tau N$ are used to read and write memory respectively. The index τ is the type of the value being read or written. Note that unlike ML and most statically-typed languages, HTT supports *strong updates*. That is, if x is a location holding a nat, then we can update the contents of x with a value of an arbitrary (small) type, not just another nat.[2] Type-safety is ensured by the pre-condition for memory reads which captures the requirement that to read a τ value out of location M, we must be able to prove that M current holds such a value.

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

When a computation is enclosed by dia, its evaluation is suspended, and the whole enclosure is considered pure, so that it can appear in the scope of functional abstractions and quantifiers, or in type dependencies.

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

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

Assertions. The assertions logic is classical and includes the standard propositional connectives and quantifiers over all types of HTT. Since prop is a type, we can quantify over propositions, and more generally over propositional functions, giving us the power of higher-order logic. The primitive proposition $\mathsf{xid}_{A,B}(M, N)$ implements *heterogeneous equality* (aka. *John Major equality* [27, 5]), and is true only if the types A and B, as well as the terms $M{:}A$ and $N{:}B$ are propositionally equal. We will use this proposition to express that if two heap locations x_1 (pointing to value $M_1{:}\tau_1$) and x_2 (pointing to value $M_2{:}\tau_2$) are equal, then $\tau_1 = \tau_2$ and $M_1 = M_2$. When the index types are equal in the heterogeneous equality $\mathsf{xid}_{A,A}(M, N)$, we abbreviate that as $\mathsf{id}_A(M, N)$, and often also write $M =_A N$ or just $M = N$. Dually, we also abbreviate $\neg\mathsf{id}_A(M, N)$ as $M \neq_A N$ or $M \neq N$. When the equality symbol appears in the judgments, it should be clear that we are using definitional equality (i.e. syntactic equality modulo alpha, beta and eta reductions). But when we use the equality symbol in the propositions, it is the abbreviation for id_A.

We notice here that id takes a type A as a parameter. Because A is an arbitrary type, and HTT can only quantify over small types, it means that we cannot actually define id as a function in the language and bind it to a variable. Rather, we consider id to be added as a primitive construct through a definition in a

[2]Obviously, we make the simplifying assumption that locations can hold a value of any type (e.g., values are boxed.)

kind of a “standard library”, and this definition is appropriately expanded during type checking and theorem proving. Similar comment will apply to quite a few propositions and predicates defined in this paper. We indicate such a predicate by annotating its name with a type when we define it. In the actual use, however, we will often omit this type annotation.

We next define the standard set-membership predicate, and the usual orderings on integers, for which we assume the customary infix fixity. We also introduce some standard definitions about functions.

$$\begin{array}{rcl}
\in_A & : & A \to (A \to \mathsf{prop}) \to \mathsf{prop} \\
 & = & \lambda p.\, \lambda q.\, q\ p \\
\leq & : & \mathsf{nat} \to \mathsf{nat} \to \mathsf{prop} \\
 & = & \lambda m.\, \lambda n.\, \exists k{:}\mathsf{nat}.n =_{\mathsf{nat}} m + k \\
< & : & \mathsf{nat} \to \mathsf{nat} \to \mathsf{prop} \\
 & = & \lambda m.\, \lambda n.\, (m \leq n) \wedge (m \neq_{\mathsf{nat}} n) \\
\mathsf{Injective}_{A,B} & : & (A \to B) \to \mathsf{prop} \\
 & = & \lambda f.\, \forall x{:}A.\, \forall y{:}A.\, f\ x =_B f\ y \supset x =_A y \\
\mathsf{Surjective}_{A,B} & : & (A \to B) \to \mathsf{prop} \\
 & = & \lambda f.\, \forall y{:}B.\, \exists x{:}A.\, y =_B f\ x \\
\mathsf{Infinite}_A & : & (A \to \mathsf{prop}) \to \mathsf{prop} \\
 & = & \lambda x.\, \exists f{:}\mathsf{nat} \to A.\, \mathsf{Injective}_{\mathsf{nat},A}\ f \wedge \forall n{:}\mathsf{nat}.\, f\ n \in_A x \\
\mathsf{Finite}_A & : & (A \to \mathsf{prop}) \to \mathsf{prop} \\
 & = & \lambda x.\, \exists n{:}\mathsf{nat}.\exists f{:}\{y{:}\mathsf{nat}.\, y < n\} \to x.\, \mathsf{Injective}(f) \wedge \mathsf{Surjective}(f) \\
\mathsf{Functional}_{A,B} & : & (A \times B \to \mathsf{prop}) \to \mathsf{prop} \\
 & = & \lambda R.\, \forall x{:}A.\forall y_1, y_2{:}B.\, (x, y_1) \in R \wedge (x, y_2) \in R \supset y_1 =_B y_2
\end{array}$$

With the above predicates, we can define the type of heaps as the following subset type.

$$\mathsf{heap} \quad = \quad \{h{:}(\mathsf{nat} \times \Sigma\alpha{:}\mathsf{mono}.\alpha){\to}\mathsf{prop}.\, \mathsf{Finite}(h) \wedge \mathsf{Functional}(h)\}$$

Here the type $\mathsf{nat} \times \Sigma\alpha{:}\mathsf{mono}$ describes that a heap is a ternary relation — it takes $M : \mathsf{nat}$, $\alpha : \mathsf{mono}$ and $N : \alpha$ and decides if the location M points to $N : \alpha$. Every heap assigns to at most finitely many locations, and assigns at most one value to every location.

As can be noticed from this definition of heaps, HTT treats locations as concrete natural numbers, rather than as members of an abstract type of references (as is usual in functional programming). This will simplify the semantic considerations somewhat, and will also allow us to program with and reason about pointer arithmetic. Also, heaps in HTT can store only values of small types. This is sufficient to model language with predicative polymorphism like SML, but is too weak for modeling Java, or the impredicative polymorphism of Haskell.

We next define several basic operators for working on heaps.

$$
\begin{array}{lcl}
\mathsf{empty} & : & \mathsf{heap} \\
\mathsf{empty} & = & \mathsf{in}\ (\lambda h.\, \bot) \\
\\
\mathsf{upd} & : & \Pi\alpha{:}\mathsf{mono}.\, \mathsf{heap} \to \mathsf{nat} \to \alpha \to \mathsf{heap} \\
\mathsf{upd} & = & \lambda\alpha.\, \lambda h.\, \lambda n.\, \lambda x. \\
 & & \mathsf{in}\ (\lambda t.\, n = \mathsf{fst}\ t \supset \mathsf{snd}\ t = (\alpha, x) \\
 & & \quad \wedge\ n \neq \mathsf{fst}\ t \supset (\mathsf{out}\ h)\ t) \\
\\
\mathsf{seleq} & : & \Pi\alpha{:}\mathsf{mono}.\mathsf{heap} \to \mathsf{nat} \to \alpha \to \mathsf{prop} \\
\mathsf{seleq} & = & \lambda\alpha.\, \lambda h.\, \lambda n.\, \lambda x.\, \mathsf{out}\ h\ (n, (\alpha, x)) \\
\\
\mathsf{free} & : & \mathsf{heap} \to \mathsf{nat} \to \mathsf{heap} \\
 & = & \lambda h.\, \lambda n.\, \mathsf{in}\ (\lambda t.\, (n \neq \mathsf{fst}\ t \wedge \mathsf{out}\ h\ t) \\
\\
\mathsf{dom} & : & \mathsf{heap} \to \mathsf{nat} \to \mathsf{prop} \\
 & = & \lambda h.\, \lambda n.\, \exists\alpha{:}\mathsf{mono}.\exists x{:}\alpha.\, \mathsf{out}\ h\ (n, (\alpha, x)) \\
\\
\mathsf{share} & : & \mathsf{heap} \to \mathsf{heap} \to \mathsf{nat} \to \mathsf{prop} \\
 & = & \lambda h_1.\, \lambda h_2.\, \lambda n.\, \forall\alpha{:}\mathsf{mono}.\forall x{:}\alpha.\, \mathsf{seleq}\ \alpha\ h_1\ n\ x =_{\mathsf{prop}} \mathsf{seleq}\ \alpha\ h_2\ n\ x \\
\\
\mathsf{splits} & : & \mathsf{heap} \to \mathsf{heap} \to \mathsf{heap} \to \mathsf{prop} \\
 & = & \lambda h.\, \lambda h_1.\, \lambda h_2.\, \forall n{:}\mathsf{nat}.\, (n \notin_{\mathsf{nat}} \mathsf{dom}\ h_1 \wedge \mathsf{share}\ h\ h_2\ n) \vee (n \notin_{\mathsf{nat}} \mathsf{dom}\ h_2 \wedge \mathsf{share}\ h\ h_1\ n)
\end{array}
$$

Let us now explain the meaning of the definitions more intuitively. empty is the empty heap, because $\lambda t.\, \bot$ is the characteristic function of the empty subset of $\mathsf{nat} \times \Sigma\alpha{:}\mathsf{mono}.\, \alpha$. The function $\mathsf{upd}\ \alpha\ h\ n\ x$ returns the heap h' obtained by changing h so that h now maps the location n to the value $x{:}\alpha$. The function $\mathsf{free}\ h\ n$ returns the heap h' obtained by removing the assignment (if any) to n from h. The proposition $\mathsf{seleq}\ \alpha\ h\ n\ x$ holds whenever the heap h maps n to $x{:}\alpha$. The predicate $\mathsf{dom}\ h$ defines the subset of locations to which the heap h assigns. The predicate $\mathsf{share}\ h_1\ h_2\ n$ holds if the heaps h_1 and h_2 share the same assignment to n. The predicate $\mathsf{splits}\ h\ h_1\ h_2$ holds if the heap h can be split into two disjoint subheaps h_1 and h_2.

We are now prepared to define the propositions from Separation Logic [35, 39, 36]. In HTT, all of these are attached an additional heap argument (e.g., instead of kind prop, a separating proposition will have kind $\mathsf{heap} \to \mathsf{prop}$). The additional heap argument abstracts the *current* heap, so that separating propositions and predicates are localized and only state facts only about the heap under consideration.

$$
\begin{array}{rcl}
\mathsf{emp} & : & \mathsf{heap} \to \mathsf{prop} \\
 & = & \lambda h.\, (h =_{\mathsf{heap}} \mathsf{empty}) \\
\mapsto & : & \Pi\alpha{:}\mathsf{type}.(\mathsf{nat} \to \alpha \to \mathsf{heap} \to \mathsf{prop}) \\
 & = & \lambda\alpha.\, \lambda n.\, \lambda x.\, \lambda h.\, (h =_{\mathsf{heap}} \mathsf{upd}\ \alpha\ \mathsf{empty}\ n\ x) \\
\hookrightarrow & : & \Pi\alpha{:}\mathsf{type}.(\mathsf{nat} \to \alpha \to \mathsf{heap} \to \mathsf{prop}) \\
 & = & \lambda\alpha.\, \lambda n.\, \lambda x.\, \lambda h.\, \mathsf{seleq}\ \alpha\ h\ n\ x \\
* & : & (\mathsf{heap} \to \mathsf{prop}) \to (\mathsf{heap} \to \mathsf{prop}) \to (\mathsf{heap} \to \mathsf{prop}) \\
 & = & \lambda p.\, \lambda q.\, \lambda h.\, \exists h_1, h_2{:}\mathsf{heap}.\, (\mathsf{splits}\ h\ h_1\ h_2) \wedge p\ h_1 \wedge q\ h_2 \\
-\!* & : & (\mathsf{heap} \to \mathsf{prop}) \to (\mathsf{heap} \to \mathsf{prop}) \to (\mathsf{heap} \to \mathsf{prop}) \\
 & = & \lambda p.\, \lambda q.\, \lambda h.\, \forall h_1, h_2{:}\mathsf{heap}.\, \mathsf{splits}(h_2, h_1, h) \supset p\ h_1 \supset q\ h_2 \\
\mathsf{this} & : & \mathsf{heap} \to \mathsf{heap} \to \mathsf{prop} \\
 & = & \mathsf{id}_{\mathsf{heap}}
\end{array}
$$

We adopt an infix notation and write $n \mapsto_\alpha x$, $n \hookrightarrow_\alpha x$, $p * q$ and $p \mathrel{-\!\!*} q$ instead of $\mapsto\ \alpha\ n\ x$, $\hookrightarrow\ \alpha\ n\ x$, $*\ p\ q$ and $\mathrel{-\!\!*}\ p\ q$, respectively. Furthermore, we abbreviate $n \mapsto_\alpha -$ instead of $\lambda h.\, \exists x{:}\alpha.\, (n \mapsto_\alpha x)\ h$, and $n \mapsto -$ instead of $\lambda h.\, \exists \alpha{:}\mathsf{mono}.\, (n \mapsto_\alpha -)\ h$. Similarly for $\hookrightarrow$.

The predicate emp holds of the current heap if and only if that heap is empty. $n \mapsto_\alpha x$ holds only if the current heap assigns x to n, but contains no other assignments. $n \hookrightarrow_\alpha x$ holds if the current heap assigns x to n, but may possibly contain more assignments. $p * q$ holds if the current heap can be split into two disjoint subheaps of which p and q hold, respectively. $p \mathrel{-\!\!*} q$ holds if whenever the current heap is extended with a subheap of which p holds, then q holds of the extension.

It is well-known that in Higher-Order Logic, we can define inductive (and also coinductive) predicates within the logic [15]. For example, let us suppose that $Q{:}(A{\to}\mathsf{prop}){\to}A{\to}\mathsf{prop}$ is monotone (i.e. the application $Q(f)$ contains f only in positive positions). Then the least fixed point of Q is definable as follows.

$$\mathsf{lfp}_A(Q) = \lambda x{:}A.\, \forall g{:}A{\to}\mathsf{prop}.\, (\forall y{:}A.\, (Q\ g\ y) \supset g\ y) \supset g\ x.$$

Another way to see the above equation is to understand g as a subset of elements of A, and write $g \subseteq A$ instead of $g : A \to \mathsf{prop}$, and $g \subseteq h$ instead of $\forall y{:}A.\, g(y) \supset h(y)$. These notations are obviously equivalent, as each subset can be represented by its characteristic function. Then the above equation is nothing but a definition of the characteristic function for the set $\bigcap\{g \subseteq A \mid Q(g) \subseteq g\}$. But of course, this set is precisely the least fixed point of Q, as established by the Knaster-Tarski theorem.

Similarly, the greatest fixed point of a monotone Q is defined as:

$$\mathsf{gfp}_A(Q) \quad = \quad \lambda x{:}A.\, \exists g{:}A{\to}\mathsf{prop}.\, (\forall y{:}A.\, g\ y \supset (Q\ g\ y)) \wedge g\ x$$

3 Examples

Diverging computation. In HTT, the term language is pure (and terminating). Recursion is an effect, and is delegated to the fragment of impure computations. Given the type A, we can write a diverging computation of type $\{P\}x{:}A\{Q\}$ as follows.

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

The computation $\mathsf{diverge}$ first sets up a recursive function $f(y : 1) = \mathsf{dia}\ (\mathsf{eval}\ (f\ y))$. The function is immediately applied to $(\,)$. The result – which will never be obtained – is returned as the overall output of the computation.

Small footprints. HTT supports small-footprint specifications, as in Separation Logic [32]. With this approach, if $\mathsf{dia}\ E$ has type $\{P\}x{:}A\{Q\}$, then P and Q need only describe the properties of the heap fragment that E actually requires in order to run. The actual heap in which E will run may be much larger, but the unspecified portion will automatically be assumed invariant. To illustrate this idea, let us consider a simple program that reads from the location x and increases its contents.

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

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

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

In order to avoid clutter in specifications, we introduce a convention: if $P, Q{:}\mathsf{heap}{\rightarrow}\mathsf{prop}$ are predicates that may depend on the free variable $x{:}A$, we write

$$x{:}A.\, \{P\}y{:}B\{Q\}$$

instead of

$$\{\lambda i.\, \exists x{:}A.\, P(i)\}y{:}B\{\lambda i.\, \lambda m.\, \forall x{:}A.\, P(i) \supset Q(m)\}.$$

This notation lets x seem to scope over both the pre- and post-condition. For example the type of incx can now be written

$$n{:}\mathsf{nat}.\, \{x \mapsto_{\mathsf{nat}} n\}r{:}1\{x \mapsto_{\mathsf{nat}} n{+}1\}.$$

The convention is easily generalized to a finite context of variables, so that we can also abbreviate the type of incxy as

$$n{:}\mathsf{nat}, k{:}\mathsf{nat}.\, \{x \mapsto_{\mathsf{nat}} n * y \mapsto_{\mathsf{nat}} k\}r{:}1\{x \mapsto_{\mathsf{nat}} n{+}1 * y \mapsto_{\mathsf{nat}} k\}.$$

Following the terminology of Hoare Logic, we call the variables abstracted outside of the Hoare triple, like n and k above, *logic variables.*

Inductive types. In higher-order logic, we can define inductive types by combining inductively defined predicates with subset types. Here we consider the example of lists (with element type A). We emphasize that this definition of lists will only be accessible in the assertions, but will not be accessible to the computational language. For example, we will have a function for testing lists for equality $\mathsf{id}_{\mathsf{list}_A} : \mathsf{list}_A{\rightarrow}\mathsf{llist}_A{\rightarrow}\mathsf{prop}$. But, as we mentioned before, elements of type prop cannot be tested for equality during run-time. If we want an equality test that is usable at run time, we need a function of type $\mathsf{llist}_A{\rightarrow}\mathsf{llist}_A{\rightarrow}\mathsf{bool}$. Currently, HTT does not have any special features for defining inductive or recursive datatypes whose elements are accessible at run time, as described above. However, it should be clear that any specific example can easily be added.

We now proceed to describe how lists can be defined in the assertion logic. We can intuitively view a list of size n as a finite set of the form $\{(0, a_0), \ldots, (n, a_n)\}$, where $a_i : A$. Hence, a list can be described as a predicate of type $(\mathsf{nat} \times A){\rightarrow}\mathsf{prop}$ which takes as input a pair (n, a) and returns $\top$ if a is the n-the element of the list, and $\bot$ otherwise. With this intuition, we can define the basic list constructors as follows. In this example, and in the future, we write $\lambda(p, q).\, (M\ p\ q)$ instead of $\lambda x.\, (M\ \mathsf{fst}\ x\ \mathsf{snd}\ x)$, and similarly for n-tuples.

$$\begin{array}{rcl} \mathsf{nil}'_A & : & \mathsf{nat} \times A \rightarrow \mathsf{prop} \\ & = & \lambda x.\, \bot \\ \mathsf{cons}'_A & : & (\mathsf{nat} \times A \rightarrow \mathsf{prop}) \rightarrow A \rightarrow (\mathsf{nat} \times A \rightarrow \mathsf{prop}) \\ & = & \lambda f.\, \lambda a.\, \lambda(i, b). \\ & & \qquad (i = 0 \supset a = b) \wedge \\ & & \qquad (\forall j{:}\mathsf{nat}.\, i = \mathsf{s}\ j \supset f\ (j, b)) \end{array}$$

Obviously, not all elements of the type $(\mathsf{nat} \times A){\rightarrow}\mathsf{prop}$ are valid lists. We want to admit as well-formed lists only those elements that can be constructed using nil'_A and cons'_A. We next define the inductive predicate islist_A to test for this property.

$$\begin{array}{rcl} \mathsf{islist}_A & : & (\mathsf{nat} \times A \rightarrow \mathsf{prop}) \rightarrow \mathsf{prop} \\ & = & \mathsf{lfp}\ (\lambda F.\, \lambda f.\, f = \mathsf{nil}'_A \vee \\ & & \qquad \exists f'{:}\mathsf{nat}{\times}A{\rightarrow}\mathsf{prop}.\, \exists a{:}A.\, f = \mathsf{cons}'_A\ f'\ a \wedge F(f')) \end{array}$$

Using islist, we can define the type of lists as a subset type of $\mathsf{nat} \times A \to \mathsf{prop}$.

$$\mathsf{list}_A \quad = \quad \{f{:}\mathsf{nat}{\times}A{\to}\mathsf{prop}.\, \mathsf{islist}_A(f)\}$$

Finally, the characteristic constructors for the type list_A are defined by subset coercions from the corresponding elements of $(\mathsf{nat} \times A){\to}\mathsf{prop}$:

$$\begin{array}{rcl} \mathsf{nil}_A & : & \mathsf{list}_A \\ & = & \mathsf{in}\ \mathsf{nil}'_A \\ \mathsf{cons}_A & : & \mathsf{list}_A \to A \to \mathsf{list}_A \\ & = & \lambda l.\, \lambda a.\, \mathsf{in}\ (\mathsf{cons}'_A\ (\mathsf{out}\ l)\ a) \end{array}$$

Inductively defined separation predicates. Because the main constructs of Separation Logic can be defined in the assertion logic of HTT, it is not surprising that we can also define the inductive predicates that are customarily used in Separation Logic. In this example we define the predicate lseg, so that $(\mathsf{lseg}\ \tau\ l\ p\ q)$ holds iff the current heap contains a non-aliased linked list between the locations p and q. Here, the linked list stores elements of a given small type τ, and the elements can be collected into $l : \mathsf{list}_\tau$. We first define the helper predicate lseg' which is uncurried in order to match the type of the lfp functional, and then curry this predicate into lseg.

$$\begin{array}{rcl} \mathsf{lseg}' & : & \Pi\alpha{:}\mathsf{mono}.\, (\mathsf{list}_\alpha \times \mathsf{nat} \times \mathsf{nat} \times \mathsf{heap}) \to \mathsf{prop} \\ & = & \lambda\alpha.\, \mathsf{lfp}\ (\lambda F.\, \lambda(l, p, q, h). \\ & & \qquad (l = \mathsf{nil}_\alpha \supset p = q \wedge \mathsf{emp}\ h) \wedge \\ & & \qquad \forall l'{:}\mathsf{list}_\alpha.\, \forall x{:}\alpha.\, (l = \mathsf{cons}_\alpha\ l' x) \supset \exists r{:}\mathsf{nat}.\, ((p \mapsto_{\alpha\times\mathsf{nat}} (x, r)) * (\lambda h'.\, F\ (l', r, q, h')))(h) \\ \mathsf{lseg} & : & \Pi\alpha{:}\mathsf{mono}.\, \mathsf{list}_\alpha \to \mathsf{nat} \to \mathsf{nat} \to \mathsf{heap} \to \mathsf{prop} \\ & = & \lambda\alpha.\, \lambda l.\, \lambda p.\, \lambda q.\, \lambda h.\, \mathsf{lseg}'_A\ \alpha\ (l, p, q, h) \end{array}$$

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

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

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

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

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

where the record notation $[x_1{:}A_1, x_2, {:}A_2, \ldots, x_n{:}A_n]$ abbreviates a sum $\Sigma x_1{:}A_1.\Sigma x_2{:}A_2.\cdots\Sigma x_n{:}A_n.1$. In English, the interface says that there is some abstract invariant I, reflecting the internal invariant of the module, paired with two functions. Both functions require that the invariant I holds before and after calls to the functions. In addition, a call $\mathsf{alloc}\,\tau\,x$ will yield a location r and a guarantee that r points to x. Furthermore, we know from the use of the spatial conjunction that r is disjoint from the internal invariant I. Thus, updates by the client to r will not break the invariant I. Dually, $(\mathsf{dealloc})$ requires that we are given a location n, pointing to some value and disjoint from the memory covered by the invariant I. Upon return, the invariant is restored and the location consumed.

If M is a module with this signature, then a program fragment that wishes to use this module will have to start with a pre-condition $\mathsf{fst\ M}$. That is, clients will generally have the type

$$\Pi\,\mathsf{M}{:}\mathsf{Alloc}.\{(\mathsf{fst\ \ M}) * P\}r{:}A\{\lambda i.\,(\mathsf{fst\ \ M}) * Q(i)\}$$

where Alloc is the signature above.

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

First, we define a predicate that describes the free memory as a list of consecutive locations initialized with $(\,) : 1$.

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

Now we can implement the allocator module:

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

Allocator Module 2. In this example we present a (slightly) more sophisticated allocator module. The module will have the same Alloc signature as in the previous example, but the implementation does not leak memory upon deallocation.

We take some liberties and assume a standard set of definitions and operations for the inductive type of lists which can be encoded using inductive predicates and memory, but to simplify the presentation, we will assume these are primitive.

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

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

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

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

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

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

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

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

We start by translating a simple SML-like program.

$$\mathsf{let\ val}\ x = \mathsf{ref}\ 0\ \mathsf{in}\ \lambda y{:}\mathsf{unit}.\, x{:=}!x + 1; !x$$

The naive (and incorrect) translation into HTT may be as follows. To reduce clutter, we remove the inessential variable y : unit and represent the return function as a dia-encapsulated computation instead.

$$\begin{array}{l}
E = \mathsf{dia}\ (\mathsf{let\ dia}\ x = \mathsf{alloc\ nat}\ 0 \\
\qquad\qquad \mathsf{in\ dia}\ (z = !_{\mathsf{nat}}\, x; x :=_{\mathsf{nat}} z{+}1; z{+}1))
\end{array}$$

The problem with E is that it cannot be given as strong a type as one would want. E's return value is itself a computation with a type $v{:}\mathsf{nat}.\, \{x \mapsto_{\mathsf{nat}} v\} r{:}\mathsf{nat}\{\lambda m.\, (x \mapsto_{\mathsf{nat}} v{+}1)(m) \wedge r = v{+}1\}$. But because this type depends on x, it is not well-formed outside of x's scope, and hence cannot be used in the type of E.

An obvious way out of this problem is to make x global, by making it part of E's return return result. However, in HTT we can use the ability to combine terms, propositions and Hoare triples, and abstract x away, while exposing only the invariant that the computation increases the content of x.

$$\begin{array}{rcl}
E' & = & \mathsf{dia}\ (\mathsf{let\ dia}\ x = \mathsf{alloc\ nat}\ 0 \\
 & & \qquad \mathsf{in}\ (\lambda v.\, x \mapsto_{\mathsf{nat}} v, \mathsf{dia}\ (z = !_{\mathsf{nat}}\, x; x :=_{\mathsf{nat}} z{+}1; z{+}1))) \\
 & : & \{I\} \\
 & & \quad t{:}\Sigma \mathit{inv}{:}\mathsf{nat}{\to}\mathsf{heap}{\to}\mathsf{prop}. \\
 & & \qquad\qquad v{:}\mathsf{nat}.\, \{\mathit{inv}\ v\} r{:}\mathsf{nat}\{\lambda h.\, (\mathit{inv}\ (v{+}1)\ h) \wedge r = v{+}1\} \\
 & & \{\lambda i.\, I * (\mathsf{fst}\ t\ 0)\}
\end{array}$$

In addition to the original value, E' now returns the invariant on its local state $\lambda v.\, x \mapsto_{\mathsf{nat}} v$. However, because HTT does not have any computational constructs for inspecting the structure of assertions, no client of E' will be able to use this invariant and learn, at run time, about the structure of the local state. The role of the invariant is only at compile time, to facilitate typechecking.

The type of E' makes it clear that the only important aspect of the local state of the return function is the natural number v which gets increased every time the function is called. Moreover, the execution of the whole program returns a local state for which $v = 0$ as the separating conjunct (fst t 0) in the postcondition formally states (because fst $t = \mathit{inv}$).

The important point is that the way in which v is obtained from the local state is completely hidden. In fact, from the outside, there is no reason to believe that the local state consists of only one location. For example, the same type could be ascribed to a similar program which maintains two locations.

$$
\begin{array}{rcl}
E'' & = & \mathsf{dia}\ (\mathsf{let\ dia}\ x = \mathsf{alloc\ nat}\ 0 \\
& & \qquad\quad \mathsf{dia}\ y = \mathsf{alloc\ nat}\ 0 \\
& & \qquad \mathsf{in}\ (\lambda v.\, (x \mapsto_{\mathsf{nat}} v) * (y \mapsto_{\mathsf{nat}} v), \\
& & \qquad\quad \mathsf{dia}\ (z = !_{\mathsf{nat}}\, x; w = !_{\mathsf{nat}}\, y; \\
& & \qquad\qquad x :=_{\mathsf{nat}} z{+}1; y :=_{\mathsf{nat}} w{+}1; (z{+}w)/2 + 1)))
\end{array}
$$

E'' has a different local invariant from E', but because the type abstracts over the invariants, the two programs have the same type. The equal types hint that the E' and E'' are observationally equivalent, i.e. they can freely be interchanged in any context. We do not prove this property here, but it is an intriguing future work, related to the recent result of Honda et al. [16, 4] on observational completeness of Hoare Logic.

We now turn to another SML-like program.

$$
\begin{array}{l}
\lambda f{:}(\mathsf{unit}{\to}\mathsf{unit}){\to}\mathsf{unit}. \\
\qquad \mathsf{let\ val}\ x = \mathsf{ref}\ 0 \\
\qquad\quad \mathsf{val}\ g = \lambda y{:}\mathsf{unit}.\, x{:=}!x + 1; y \\
\qquad \mathsf{in} \\
\qquad\quad f\ g
\end{array}
$$

The main property of this program is that the function f can access the local variable x only through a call to the function g. The naive translation into HTT is presented below. We again remove some inessential bound variables, and represent g as a computation instead of a function. Again, the first attempt at the translation cannot be typed.

$$
\begin{array}{rcl}
F & = & \lambda f.\, \mathsf{dia}\ (\mathsf{let\ dia}\ x = \mathsf{alloc\ nat}\ 0 \\
& & \qquad\qquad \mathsf{val}\ g = \mathsf{dia}\ (z = !_{\mathsf{nat}}\, x; x :=_{\mathsf{nat}} z{+}1; (\,)) \\
& & \qquad\qquad \mathsf{eval}\,(f\ g))
\end{array}
$$

Part of the problem of F is similar as before; the local state of g must be abstracted in order to hide the dependence on x from f. However, this is not sufficient. Because we evaluate $f\ g$ at the end, we also need to know the invariant for f in order to state the postcondition for F. But f is a function of g, so the invariant of f may depend on the invariant of g. In other words, the invariant of f must be a *higher-order* predicate.

$$
\begin{array}{rcl}
F' & = & \lambda f.\, \mathsf{dia}\ (\mathsf{let\ dia}\ x = \mathsf{alloc\ nat}\ 0 \\
& & \qquad\qquad \mathsf{val}\ g = (\lambda v.\, x \mapsto_{\mathsf{nat}} v, \mathsf{dia}\ (z = !_{\mathsf{nat}}\, x; x :=_{\mathsf{nat}} z{+}1; (\,))) \\
& & \qquad\qquad \mathsf{eval}\,((\mathsf{snd}\ f)\ g)) \\
& : & \Pi f{:}\Sigma p{:}\mathsf{nat}{\to}(\mathsf{nat}{\to}\mathsf{heap}{\to}\mathsf{prop}){\to}\mathsf{heap}{\to}\mathsf{prop}. \\
& & \qquad \Pi g{:}\Sigma inv{:}\mathsf{nat}{\to}\mathsf{heap}{\to}\mathsf{prop}. \\
& & \qquad\quad v{:}\mathsf{nat}.\, \{inv\ v\}r{:}1\{inv\ (v+1)\}. \\
& & \qquad w{:}\mathsf{nat}.\, \{\mathsf{fst}\ g\ w\}s{:}1\{p\ w\ (\mathsf{fst}\ g)\}. \\
& & \{I\} \\
& & \quad t{:}1 \\
& & \{\lambda i.\, I * \lambda h.\, \exists x{:}\mathsf{nat}.\, (\mathsf{fst}\ f')\ 0\ (\lambda v.\, x \mapsto_{\mathsf{nat}} v)\ h\}
\end{array}
$$

In this program, f and g carry the invariants of their local states (e.g., $p = \mathsf{fst}\ f$ is the invariant of $\mathsf{snd}\ f$ and $inv = \mathsf{fst}\ g = \lambda v.\, x \mapsto_{\mathsf{nat}} v$ is the invariant of $\mathsf{snd}\ g$). The predicate p takes an argument inv and a natural number n, and returns a description of the state obtained after applying f to g in a state where $inv(n)$ holds. The postcondition for F_1 describes the ending heap as $p\ 0\ inv$ thus showing that HTT can also reveal the information about local state when needed. For example, it reveals that $x \mapsto 0$ before the last application of f, and that f was applied to a function with invariant $\lambda v.\, x \mapsto_{\mathsf{nat}} v$.

4 Hereditary substitutions

The equational reasoning in HTT is centered around the concept of *canonical forms*. A canonical form is an expression which is beta normal and eta long. We compare two expressions for equality modulo beta and eta equations by reducing the terms to their canonical forms and then comparing for syntactic equality.

In HTT there is a simple syntactic way to test if an expression is beta normal. We simply need to check that the expression does not contain any occurrences of the constructor $M : A$. Indeed, without this constructor, it is not possible to write a beta redex in HTT.

In order to deal with arithmetic we add several further equations to the definitions of canonical forms. In particular, we will consider the expressions $\mathsf{z} + M$ and $M + \mathsf{z}$ to be redexes that reduce to M. Similarly, $(\mathsf{s}\ M) + N$ and $M + \mathsf{s}\ N$ reduce to $\mathsf{s}\ (M + N)$. In the case of multiplication, $\mathsf{z} * M$ and $M * \mathsf{z}$ reduce to z, $(\mathsf{s}\ M) * N$ reduces to $M * N + N$, and symmetrically for $M * (\mathsf{s}\ N)$. Finally, $\mathsf{eq}_{\mathsf{nat}}(\mathsf{z}, \mathsf{z})$ reduces to true, $\mathsf{eq}_{\mathsf{nat}}(\mathsf{z}, \mathsf{s}\ M)$ and $\mathsf{eq}_{\mathsf{nat}}(\mathsf{s}\ M, \mathsf{z})$ reduce to false, and $\mathsf{eq}_{\mathsf{nat}}(\mathsf{s}\ M, \mathsf{s}\ N)$ reduces to $\mathsf{eq}_{\mathsf{nat}}(M, N)$.

A related concept is that of hereditary substitutions, which combines ordinary capture-avoiding substitutions with beta and eta normalization. For example, where an ordinary substitution creates a redex $(\lambda x. M)\ N$, a hereditary substitution proceeds to on-the-fly substitute N for x in M. This may produce another redex, which is immediately reduced, producing another redex, and so on. If the terms M and N were already canonical (i.e., beta normal and eta long), then the result of hereditary substitution is also canonical.

Hereditary substitutions are only defined on expressions which do not contain the constructor $M : A$, and also produce expressions which are free of this constructor. In Section 5 we will present a type system of HTT which computes canonical forms in parallel with type checking, by systematically removing the occurrences of $M : A$ from the expression being typechecked.

The rest of the present section defines the notion of hereditary substitutions. Thus, we here consider that all the expressions are free of the constructor $M : A$, as hereditary substitutions will never be applied over a term with this constructor.

Another important property is that hereditary substitutions are defined on possibly ill-typed expressions. Thus, we can formulate the equational theory on HTT while avoiding the mutual dependence with the typing judgments. This significantly simplifies the development and the meta theory of HTT.

When the input terms of the hereditary substitutions are not well-typed, the hereditary substitution does not need to produce a result. However, whether the hereditary substitution will have a result can be determined in a *finite number* of steps. Hereditary substitutions are always *terminating*.

Our development of hereditary substitutions is based on the work of Watkins et al. [42], and also our previous work on HTT [33], but is somewhat more complicated now. The bulk of the complication comes from the fact that we can now compute with small types, and we can create types that are not only variable or constant but can contain elimination forms. An example is a type $A = \Pi x{:}\mathsf{mono}{\times}\mathsf{mono}.\ \mathsf{nat}{\times}(\mathsf{fst}\ x)$, where the subexpression $\mathsf{fst}\ x$ denotes a small type that is, obviously, computed out of x.

A first consequence of this new expressiveness is that type dependencies do not only arise from the refinements, as it used to be the case in [33]. The example type A above is essentially dependent and yet, contains no refinements.

This property makes it impossible to use simple types, as we did in [33], as indexes to the hereditary substitutions, and as a termination metric, since now erasing the refinements need not result in a simple type. In particular, the termination metric needs to differentiate between small and large types, and order the small types before the large ones. Moreover, as we will soon see, the Π and Σ types must be larger than all their substitution instances.

The termination metric $m(A)$ of the type A is the pair (l, s) where l is the number of large type constructors appearing in A, and s is the number of small constructors. Here, the primitive constructors like $\mathsf{nat}, \mathsf{bool}, \mathsf{prop}, 1$, etc do not matter, as they are both large and small so they do not contribute to differentiating between types (i.e. we do not count them in the metric). The pairs (l, s) are ordered lexicographically, i.e. $(l_1, s_1) < (l_2, s_2)$ iff $l_1 < l_2$ or $l_1 = l_2$ and $s_1 < s_2$ (i.e. one large constructor matters more than an arbitrary number of small constructors).

The definition of the metric follows.

$$\begin{array}{rcl}
m(K) & = & (0,0) \\
m(\mathsf{nat}) & = & (0,0) \\
m(\mathsf{bool}) & = & (0,0) \\
m(\mathsf{prop}) & = & (0,0) \\
m(1) & = & (0,0) \\
m(\Pi x{:}\tau.\, B) & = & (0,1) + m(\tau) + m(B) \\
m(\Sigma x{:}\tau.\, B) & = & (0,1) + m(\tau) + m(B) \\
m(\{P\}x{:}\tau\{Q\}) & = & (0,1) + m(\tau) \\
m(\{x{:}\tau.\, P\}) & = & (0,1) + m(\tau) \\
\\
m(\mathsf{mono}) & = & (1,0) \\
m(\Pi x{:}A.\, B) & = & (1,0) + m(A) + m(B) \\
m(\Sigma x{:}A.\, B) & = & (1,0) + m(A) + m(B) \\
m(\{P\}x{:}A\{Q\}) & = & (1,0) + m(A) \\
m(\{x{:}A.\, P\}) & = & (1,0) + m(A)
\end{array}$$

In the last four cases we assume that A is not small, as otherwise one of the previous cases of the definition would apply. We will often write $A \leq B$ and $A < B$ instead of $m(A) \leq m(B)$ and $m(A) < m(B)$, respectively.

With this metric, it is should be intuitively clear that any substitution into a well-formed Π or Σ type reduces the metric (and we will prove that this property holds even for ill-formed types). Given a type $\Pi x{:}\tau.\, B$, then x can appear only in the refinements of B. Since the refinements do not contribute to the metric, neither x nor any substitute for it will be counted.

If, on the other hand, the type of x is large, i.e. we have $\Pi x{:}A.\, B$, then substituting N for x into B can increase the metric, but it can only increase the count of small constructors. Because N is a term, it can only contain small type constructors, as predicativity does not allow large types to be considered as terms. Now, even if we increase the count of small constructors, the substitution will remove Π and thus decrease the count of large constructors by one, resulting in an overall decrease of the metric.

Now we define the following set of mutually recursive functions:

$$\begin{array}{lcl}
\mathsf{expand}_A(K) & = & N' \\
[M/x]^k_A(K) & = & K' \text{ or } N' :: A' \\
[M/x]^m_A(N) & = & N' \\
[M/x]^p_A(P) & = & P' \\
[M/x]^a_A(A) & = & A' \\
[M/x]^e_A(E) & = & E' \\
\langle E/x\rangle_A(F) & = & F'
\end{array}$$

The function $\mathsf{expand}_A(K)$ eta expands the argument elim term K, according to the index type A. The functions $[M/x]^*_A(-)$ for $* \in \{k, m, p, a, e\}$ are hereditary substitutions of M into elim terms, intro terms, assertions, types and computations, respectively. The function $\langle M/x\rangle_A(-)$ is a hereditary version of monadic substitution which encodes beta reduction for the type of Hoare triples [33].

The functions are all partial, in the sense that on some inputs, the output may be undefined. We will show, however, that if the output is undefined, the functions fail in finitely many steps (there is no divergence). In the next section we will prove that the functions are always defined on well-typed inputs.

$$
\begin{array}{llll}
\mathsf{expand}_L(K) & = & \mathsf{eta}_L\ K & \\
\mathsf{expand}_{\mathsf{bool}}(K) & = & K & \\
\mathsf{expand}_{\mathsf{nat}}(K) & = & K & \\
\mathsf{expand}_{\mathsf{prop}}(K) & = & K & \\
\mathsf{expand}_{\mathsf{mono}}(K) & = & K & \\
\mathsf{expand}_1(K) & = & (\,) & \\
\mathsf{expand}_{\Pi x{:}A_1.\,A_2}(K) & = & \lambda x.\,N & \text{where } M = \mathsf{expand}_{A_1}(x) \\
 & & & \text{and } N = \mathsf{expand}_{A_2}(K\ M) \\
 & & & \text{choosing } x \notin \mathsf{FV}(A_1, K) \\
\mathsf{expand}_{\Sigma x{:}A_1.\,A_2}(K) & = & (N_1, N_2) & \text{where } N_1 = \mathsf{expand}_{A_1}(\mathsf{fst}\ K) \\
 & & & \text{and } A_2' = [N_1/x]^a_{A_1}(A_2) \\
 & & & \text{and } N_2 = \mathsf{expand}_{A_2'}(\mathsf{snd}\ K) \\
\mathsf{expand}_{\{P\}x{:}A\{Q\}}(K) & = & \mathsf{dia}\ E & \text{where } M = \mathsf{expand}_A(x) \\
 & & & \text{and } E = (\mathsf{let\ dia}\ x = K\ \mathsf{in}\ M) \\
 & & & \text{choosing } x \notin \mathsf{FV}(A) \\
\mathsf{expand}_{\{x{:}A.\,P\}}(K) & = & \mathsf{in}\ N & \text{where } N = \mathsf{expand}_A(\mathsf{out}\ K) \\
\mathsf{expand}_A(K) & & \text{fails} & \text{otherwise}
\end{array}
$$

The only characteristic case of $\mathsf{expand}_\tau(K)$ is when $\tau = L$ is an elim form. Substituting free variables in L may turn L into small types with different top-level constructors. This makes it impossible to predict the eventual shape of L and eagerly perform even the first step of eta expansion of K. With this in mind, expand simply returns a suspension $\mathsf{eta}_L\ K$ which postpones the expansion of K until some substitution into L results with a concrete type like nat or bool or $\Pi x{:}\tau_1.\,\tau_2$.

Next we define the hereditary substitution into elim forms.

$$
\begin{array}{llll}
[M/x]^k_A(x) & = & M :: A & \\
[M/x]^k_A(y) & = & y & \text{if } y \neq x \\
[M/x]^k_A(K\ N) & = & K'\ N' & \text{if } [M/x]^k_A(K) = K' \text{ and } [M/x]^m_A(N) = N' \\
[M/x]^k_A(K\ N) & = & O' :: A_2' & \text{if } [M/x]^k_A(K) = \lambda y.\,M' :: \Pi y{:}A_1.\,A_2 \\
 & & & \text{where } N' = [M/x]^m_A(N) \\
 & & & \text{and } O' = [N'/y]^m_{A_1}(M') \text{ and } A_2' = [N'/y]^a_{A_1}(A_2) \\
[M/x]^k_A(\mathsf{fst}\ K) & = & \mathsf{fst}\ K' & \text{if } [M/x]^k_A(K) = K' \\
[M/x]^k_A(\mathsf{fst}\ K) & = & N_1' :: A_1 & \text{if } [M/x]^k_A(K) = (N_1', N_2') :: \Sigma y{:}A_1.\,A_2 \\
[M/x]^k_A(\mathsf{snd}\ K) & = & \mathsf{snd}\ K' & \text{if } [M/x]^k_A(K) = K' \\
[M/x]^k_A(\mathsf{snd}\ K) & = & N_2' :: A_2' & \text{if } [M/x]^k_A(K) = (N_1', N_2') :: \Sigma y{:}A_1.\,A_2 \\
 & & & \text{where } A_2' = [N_1'/y]^a_{A_1}(A_2) \\
[M/x]^k_A(\mathsf{out}\ K) & = & \mathsf{out}\ K' & \text{if } [M/x]^k_A(K) = K' \\
[M/x]^k_A(\mathsf{out}\ K) & = & N' :: A' & \text{if } [M/x]^k_A(K) = \mathsf{in}\ N' :: \{y{:}A'.\,P\} \\
[M/x]^k_A(K') & & \text{fails} & \text{otherwise}
\end{array}
$$

The characteristic case above is the substitution into $K\ N$ when $[M/x]^k_A(K)$ results with a function $\lambda y.\,M' :: \Pi y{:}A_1.\,A_2$. The hereditary substitution proceeds to substitute $N' = [M/x]^m_A(N)$ for y in M', possibly triggering new hereditary substitutions, and so on. Because hereditary substitutions are applied over not necessarily well-typed terms, it is quite possible that $[M/x]^k_A(K)$ may produce an intro form that is not a lambda abstraction. In that case, we cannot proceed with the substitution of N' as there is no abstraction to substitute into – the result fails to be defined. On well-typed inputs, however, the results are always defined and well-typed, as we show in the next section.

Before we can define substitution into intro forms, we need three (total) helper functions that define the semantics of operations on natural numbers. These functions essentially perform the reductions on arithmetic expressions that we described at the beginning of the section.

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

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

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

Now the substitution into intro forms.

$$\begin{array}{lcll}
[M/x]^m_A(K) & = & K' & \text{if } [M/x]^k_A(K) = K' \\
[M/x]^m_A(K) & = & N' & \text{if } [M/x]^k_A(K) = N' :: A' \\
[M/x]^m_A(\mathsf{eta}_K\ L) & = & \mathsf{eta}_{K'}\ L' & \text{if } [M/x]^k_A(K) = K' \text{ and } [M/x]^k_A(L) = L' \\
[M/x]^m_A(\mathsf{eta}_K\ L) & = & N' & \text{if } [M/x]^k_A(K) = \tau' :: \mathsf{mono} \text{ and } [M/x]^k_A(L) = L' \\
 & & & \text{and } \mathsf{expand}_{\tau'}(L') = N' \\
[M/x]^m_A(\mathsf{eta}_K\ L) & = & \mathsf{eta}_{\tau'}\ L' & \text{if } [M/x]^k_A(L) = \mathsf{eta}_{\tau'}\ L' :: \tau' \text{ and } \tau' = [M/x]^k_A(K) \\
[M/x]^m_A(\mathsf{eta}_K\ L) & = & N' & \text{if } [M/x]^k_A(L) = N' :: \tau' \text{ and } [M/x]^k_A(K) = \tau' :: \mathsf{mono} \\
 & & & \text{and } [N'/z]_{\tau'}(\mathsf{expand}_{\tau'}(z)) = N' \\
 & & & \text{where } z \notin \mathsf{FV}(N', \tau') \\
[M/x]^m_A((\ )) & = & (\ ) & \\
[M/x]^m_A(\lambda y.\, N) & = & \lambda y.\, N' & \text{where } [M/x]^m_A(N) = N' \\
 & & & \text{choosing } y \notin \mathsf{FV}(M) \text{ and } y \neq x \\
[M/x]^m_A((N_1, N_2)) & = & (N_1', N_2') & \text{where } [M/x]^m_A(N_i) = N_i' \\
[M/x]^m_A(\mathsf{dia}\ E) & = & \mathsf{dia}\ E' & \text{if } [M/x]^e_A(E) = E' \\
[M/x]^m_A(\mathsf{in}\ N) & = & \mathsf{in}\ N' & \text{if } [M/x]^m_A(N) = N' \\
[M/x]^m_A(\mathsf{true}) & = & \mathsf{true} & \\
[M/x]^m_A(\mathsf{false}) & = & \mathsf{false} & \\
[M/x]^m_A(\mathsf{z}) & = & \mathsf{z} & \\
[M/x]^m_A(\mathsf{s}\ N) & = & \mathsf{s}\ N' & \text{where } [M/x]^m_A(N) = N' \\
[M/x]^m_A(N_1 + N_2) & = & \mathsf{plus}(N_1', N_2') & \text{where } [M/x]^m_A(N_1) = N_1' \text{ and } [M/x]^m_A(N_2) = N_2' \\
[M/x]^m_A(N_1 \times N_2) & = & \mathsf{times}(N_1', N_2') & \text{where } [M/x]^m_A(N_1) = N_1' \text{ and } [M/x]^m_A(N_2) = N_2' \\
[M/x]^m_A(\mathsf{eq}_{\mathsf{nat}}(N_1, N_2)) & = & \mathsf{equals}_{\mathsf{nat}}(N_1', N_2') & \text{where } [M/x]^m_A(N_1) = N_1' \text{ and } [M/x]^m_A(N_2) = N_2' \\
[M/x]^m_A(\tau) & = & \tau' & \text{if } [M/x]^a_A(\tau) = \tau', \text{ assuming } \tau \neq K \\
[M/x]^m_A(P) & = & P' & \text{if } [M/x]^p_A(P) = P', \text{ assuming } P \neq K \\
[M/x]^m_A(N) & & \text{fails} & \text{otherwise}
\end{array}$$

The characteristic cases in this definitions concern substitution into $\mathsf{eta}_K\ L$. There are two important observations to be made about these cases. First, the invariant associated with these cases is that the result of the substitutions must always be eta expanded with respect to $\tau' = [M/x]^a_A(K)$, i.e. it must look like an output of $\mathsf{expand}_{\tau'}(-)$. This property will be essential later in Lemma 8 where we give an inductive proof that hereditary substitutions compose. But if $[M/x]^k_A(L) = N' :: \tau'$, then we do not have any guarantees that N' is actually expanded with respect to τ'. That is why we strengthen the induction hypothesis by extending the definition with a check that N' is expanded, i.e. $N' = [N'/z]_{\tau'}(\mathsf{expand}_{\tau'} z)$.

If the inputs to the hereditary substitutions are well-typed, then this check is superfluous, because HTT typing rules will ensure that canonical forms are eta expanded. The check is required only for the meta-level arguments about compositionality of substitutions 8, but need not be present in the actual implementation where substitutions are always invoked with well-typed inputs.

The second observation concerns how the termination metric decreases in the case when $[M/x]^k_A(L) = N' :: \tau'$. The check for expansion of N' that we just described, uses a hereditary substitution with an index τ', so we must make sure that $\tau' < A$, or else we cannot prove that hereditary substitutions terminate.

This property will be ensured by the second equation in the definition of the case, which requires that $[M/x]^k_A(K) = \tau' :: \mathsf{mono}$. As we show in Theorem 3, the equation ensures that $\mathsf{mono} \leq A$, and because τ' is small, we have $\tau' < \mathsf{mono}$, and thus $\tau' < A$. If we merely required that $[M/x]^k_A(K) = \tau' :: A'$ for some unspecified A', we would not be able to carry out the inductive argument.

Substitution into propositions is straightforward, and does not require much comment.

$$\begin{array}{lcll}
[M/x]^p_A(K) & = & K' & \text{if } [M/x]^k_A(K) = K' \\
[M/x]^p_A(K) & = & P' & \text{if } [M/x]^k_A(K) = P' :: \mathsf{prop} \\
[M/x]^p_A(\mathsf{xid}_{A_1,A_2}(N_1, N_2)) & = & \mathsf{xid}_{A'_1,A'_2}(N'_1, N'_2) & \text{where } [M/x]^a_A(A_i) = A'_i \text{ and } [M/x]^m_A(N_i) = N'_i \\
[M/x]^p_A(\top) & = & \top & \\
[M/x]^p_A(\bot) & = & \bot & \\
[M/x]^p_A(P_1 \wedge P_2) & = & P'_1 \wedge P'_2 & \text{where } [M/x]^p_A(P_i) = P'_i \\
[M/x]^p_A(P_1 \vee P_2) & = & P'_1 \vee P'_2 & \text{where } [M/x]^p_A(P_i) = P'_i \\
[M/x]^p_A(P_1 \supset P_2) & = & P'_1 \supset P'_2 & \text{where } [M/x]^p_A(P_i) = P'_i \\
[M/x]^p_A(\neg P) & = & \neg P' & \text{where } [M/x]^p_A(P) = P' \\
[M/x]^p_A(\forall y{:}B.\, P) & = & \forall y{:}B'.\, P' & \text{where } [M/x]^a_A(B) = B' \text{ and } [M/x]^p_A(P) = P' \\
 & & & \text{choosing } y \notin \mathsf{FV}(M) \text{ and } y \neq x \\
[M/x]^p_A(\exists y{:}B.\, P) & = & \exists y{:}B'.\, P' & \text{where } [M/x]^a_A(B) = B' \text{ and } [M/x]^p_A(P) = P' \\
 & & & \text{choosing } y \notin \mathsf{FV}(M) \text{ and } y \neq x \\
[M/x]^p_A(P) & & \text{fails} & \text{otherwise}
\end{array}$$

Substitution into types follows.

$$\begin{array}{lcll}
[M/x]^a_A(K) & = & K' & \text{if } [M/x]^k_A(K) = K' \\
[M/x]^a_A(K) & = & \tau' & \text{if } [M/x]^k_A(K) = \tau' :: \mathsf{mono} \\
[M/x]^a_A(\mathsf{bool}) & = & \mathsf{bool} & \\
[M/x]^a_A(\mathsf{nat}) & = & \mathsf{nat} & \\
[M/x]^a_A(\mathsf{prop}) & = & \mathsf{prop} & \\
[M/x]^a_A(1) & = & 1 & \\
[M/x]^a_A(\mathsf{mono}) & = & \mathsf{mono} & \\
[M/x]^a_A(\Pi y{:}A_1.\, A_2) & = & \Pi y{:}A'_1.\, A'_2 & \text{if } [M/x]^a_A(A_1) = A'_1 \text{ and } [M/x]^a_A(A_2) = A'_2 \\
 & & & \text{choosing } y \notin \mathsf{FV}(M) \text{ and } x \neq y \\
[M/x]^a_A(\Sigma y{:}A_1.\, A_2) & = & \Sigma y{:}A'_1.\, A'_2 & \text{if } [M/x]^a_A(A_1) = A'_1 \text{ and } [M/x]^a_A(A_2) = A'_2 \\
 & & & \text{choosing } y \notin \mathsf{FV}(M) \text{ and } x \neq y \\
[M/x]^a_A(\{P\}y{:}B\{Q\}) & = & \{P'\}y{:}B'\{Q'\} & \text{if } [M/x]^m_A(P) = P' \text{ and } [M/x]^a_A(B) = B' \\
 & & & \text{and } ([M/x]^m_A(Q) = Q' \\
 & & & \text{choosing } y \notin \mathsf{FV}(M) \text{ and } x \neq y \\
[M/x]^a_A(\{y{:}B.\, P\}) & = & \{y{:}B'.\, P'\} & \text{if } [M/x]^m_A(P) = P' \text{ and } [M/x]^a_A(B) = B' \\
 & & & \text{choosing } y \notin \mathsf{FV}(M) \text{ and } x \neq y \\
[M/x]^a_A(B) & & \text{fails} & \text{otherwise}
\end{array}$$

The most characteristic case here is substitution into elim forms K. This is yet another instance of the same problem described previously about substitution into intro form $\mathsf{eta}_K\ L$. Types that are elim forms have metric zero. But when substituting into an elim form, it is possible that the result becomes a type that is an intro form, and such types may have positive metric. Thus, substituting into a type may increase the termination metric.

When the index type A of the substitution is large, this is nothing to worry about. As argued previously, the substitution may only increase the count of small type constructors, but the substitution itself is usually a consequence of instantiating a large Π or Σ constructor thus, overall, decreasing the termination metric.

But if the index type A of the substitution is small, then the increase in the count of small constructors should be prevented, by preventing that the elim form K is turned into an intro form. We prevent that by insisting that $[M/x]^k_A(K) = \tau' :: A'$, where $A' = \mathsf{mono}$. This forces the result to be a small type, and moreover, forces A to be large, because $A' = \mathsf{mono}$ can only hold if A is large, i.e. $\mathsf{mono} \leq A$. (as we show in the termination theorem (Theorem 3).

Substitution into commands is compositional, so we omit it. The substitution into computations is standard and does not require much comment.

$$
\begin{array}{lcll}
[M/x]^e_A(N) & = & N' & \text{if } [M/x]^m_A(N) = N' \\
[M/x]^e_A(\mathsf{let\ dia}\ y = K\ \mathsf{in}\ E) & = & \mathsf{let\ dia}\ y = K'\ \mathsf{in}\ E' & \text{if } [M/x]^k_A(K) = K' \text{ and } [M/x]^e_A(E) = E' \\
 & & & \text{choosing } y \notin \mathsf{FV}(M) \text{ and } y \neq x \\
[M/x]^e_A(\mathsf{let\ dia}\ y = K\ \mathsf{in}\ E) & = & F' & \text{if } [M/x]^k_A(K) = \mathsf{dia}\ F :: \{P\}y{:}A'\{Q\} \\
 & & & \text{and } [M/x]^e_A(E) = E' \\
 & & & \text{and } F' = \langle F/y\rangle_{A'}(E') \\
 & & & \text{choosing } y \notin \mathsf{FV}(M) \text{ and } y \neq x \\
[M/x]^e_A(y = c; E) & = & y = c'; E' & \text{where } c' = [M/x]^c_A(c) \text{ and } E' = [M/x]^e_A(E) \\
 & & & \text{choosing } y \notin \mathsf{FV}(M) \text{ and } y \neq x \\
[M/x]^e_A(E) & & \text{fails} & \text{otherwise}
\end{array}
$$

Finally, the monadic hereditary substitution.

$$
\begin{array}{lcll}
\langle M/x\rangle_A(F) & = & F' & \text{if } F' = [M/x]^e_A(F) \\
\langle \mathsf{let\ dia}\ y = K\ \mathsf{in}\ E/x\rangle_A(F) & = & \mathsf{let\ dia}\ y = K\ \mathsf{in}\ F' & \text{if } F' = \langle E/x\rangle_A(F) \\
 & & & \text{choosing } y \notin \mathsf{FV}(F) \\
\langle y = c; E/x\rangle_A(F) & = & y = c; F' & \text{if } F' = \langle E/x\rangle_A(F) \\
 & & & \text{choosing } y \notin \mathsf{FV}(F) \\
\langle E/x\rangle_A(F) & & \text{fails} & \text{otherwise}
\end{array}
$$

4.1 Properties of hereditary substitutions

Definition 1 (Head of an elim term)
Given an elim term K, we define $\mathsf{head}(K)$ to be the variable at the beginning of K. More formally:

$$
\begin{array}{rcl}
\mathit{head}(x) & = & x \\
\mathit{head}(K\ M) & = & \mathit{head}(K) \\
\mathit{head}(\mathit{fst}\ K) & = & \mathit{head}(K) \\
\mathit{head}(\mathit{snd}\ K) & = & \mathit{head}(K) \\
\mathit{head}(\mathit{out}\ K) & = & \mathit{head}(K)
\end{array}
$$

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

1. *$[M/x]^k_A(K) = K'$ is elim iff $\mathit{head}(K) \neq x$*
2. *$[M/x]^k_A(K) = N' :: A'$ is intro iff $\mathit{head}(K) = x$*

Proof: Straightforward. ■

Theorem 3 (Termination of hereditary substitutions)

1. *If $[M/x]^k_A(K) = N' :: A'$, then $A' \leq A$.*

2. *If $[M/x]^a_\tau(B)$ exists, then $m([M/x]^a_\tau(B)) = m(B)$.*

3. *If $[M/x]^a_A(B)$ exists, then $m([M/x]^a_A(B) \leq (0, n) + m(B)$ for some natural number n.*

4. *if $[M/x]^a_A(B)$ exists, then $m([M/x]^a_A(B)) < m(\Pi x{:}A.\,B), m(\Sigma x{:}A.\,B)$*

5. *$\mathsf{expand}_\tau(K)$, $[M/x]^*_A(-)$ and $\langle E/x\rangle_A(-)$ terminate, either by returning a result, or failing in finitely many steps.*

Proof: By mutual nested induction, first on $m(A)$ and then on the structure of the argument into which we substitute or which we expand (in the last statement). For the case of monadic substitutions, we use induction on the structure of E.

For the first statement, we simply go through all the cases, and notice that we either return the exact index type, or apply a hereditary substitution into a strictly smaller type, obtained as a body of Π or Σ type. In the later case, we can apply the induction hypothesis and statement 4 conclude that the metric decreased.

The second and third statements are easy, and the only interesting case is when $B = K$, and $\mathsf{head}(K) = x$. In the second statement that case cannot arise, because that would lead us to conclude by the first statement that $\mathsf{mono} \leq \tau$, which is not possible. In the third statement we know that $[M/x](K) = N' :: \mathsf{mono}$, so we just take $n = m(N')$.

For the fourth statement, we consider several cases:

- case $B = K$. If $[M/x]^a_A(B) = K'$, then the statement is trivial, as the measure of K' is 0. If $[M/x]^a_A(B) = N' :: \mathsf{mono}$, then by the first statement, we know $A > \mathsf{mono}$. Thus, A is not small, and hence $m(\Pi x{:}A.B), m(\Sigma_{x:A}.B) \geq (1, 0)$, whereas $m(N) < (1, 0)$, simply because N is a term. Other outcomes cannot appear if $B = K$.

- case $B = \mathsf{nat}, \mathsf{bool}, \mathsf{prop}, \mathsf{mono}, 1$ are simple, as all the measures are 0.

- case $B = \Pi x{:}B_1.\,B_2$. If A is simple, we know that substituting into B retains the same measure, but abstraction increases the measure by 1, so that the quantified proposition must be larger. If A is not simple, we know that substitution increases the measure by $(0, n)$, for some n. But quantification increases the measure by $(1, 0)$, so the quantified proposition must again be larger. ∎

Lemma 4 (Trivial hereditary substitutions)
If $x \notin \mathit{FV}(T)$, then $[M/x]^*_A(T) = T$.

Proof: Straightforward induction on T. ∎

Lemma 5 (Hereditary substitutions and primitive operations)
Suppose that $[M/x]^m_A(N_1)$ and $[M/x]^m_A(N_2)$ exist. Then the following holds.

1. $[M/x]^m_A(\mathit{plus}(N_1, N_2)) = \mathit{plus}([M/x]^m_A(N_1), [M/x]^m_A(N_2))$.

2. $[M/x]^m_A(\mathit{times}(N_1, N_2)) = \mathit{times}([M/x]^m_A(N_1), [M/x]^m_A(N_2))$.

3. $[M/x]^m_A(\mathit{equals}(N_1, N_2)) = \mathit{equals}([M/x]^m_A(N_1), [M/x]^m_A(N_2))$.

4. $[M/x]^m_A(\mathit{equals}_{\mathit{nat}}(N_1, N_2)) = \mathit{equals}_{\mathit{nat}}([M/x]^m_A(N_1), [M/x]^m_A(N_2))$.

5. $[M/x]^m_A(\mathit{equals}_{\mathit{ref}}(N_1, N_2)) = \mathit{equals}_{\mathit{ref}}([M/x]^m_A(N_1), [M/x]^m_A(N_2))$.

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

Definition 6 (Expandedness)
The intro term N is expanded with respect to a type A, if $[N/z]^m_A(\mathsf{expand}_A(z)) = N$, for $z \notin \mathsf{FV}(N, A)$.

The definition of expandedness will be used in this section only when $A = \tau$ is a small type.

Lemma 7 (Hereditary substitutions and expansions)
If $\mathsf{expand}_A(K)$ exists, then it is expanded with respect to A, i.e. $[\mathsf{expand}_A(K)/z]_A(\mathsf{expand}_A(z)) = \mathsf{expand}_A(K)$, for $z \notin \mathsf{FV}(K, A)$.

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

Lemma 8 (Composition of hereditary substitutions)
Suppose that T ranges over expressions of any syntactic category (i.e., elim terms, intro terms, assertions, types and computations), and let $* \in \{k, m, p, a, e\}$ 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 $[M_0/x]^m_A(M_1)$ and $[M_0/x]^a_A(B)$ exist, then $[M_0/x]^*_A(T_1) = [[M_0/x]^m_A(M_1)/y]^*_{[M_0/x]^a_A(B)}(T_0)$.*
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 $[M_0/x]^e_A(E_1)$ and $[M_0/x]^a_A(B)$ exist, then $[M_0/x]^e_A(F_1) = \langle [M_0/x]^e_A(E_1)/y\rangle_{[M_0/x]^a_A(B)}(F_0)$.*
3. *If $x \notin \mathsf{FV}(F, B)$ and $\langle E_1/y\rangle_B(F) = F_1$ and $\langle E_0/x\rangle_A(E_1)$ exist, then $\langle E_0/x\rangle_A(F_1) = \langle\langle E_0/x\rangle_A(E_1)/y\rangle_B(F)$.*
4. *If $[M/x]^k_A(K) = K'$ exists and is an elim form and $[M/x]^a_A(B)$ and $\mathsf{expand}_B(K)$ exists, then $[M/x]^m_A(\mathsf{expand}_B(K)) = \mathsf{expand}_{[M/x]^a_A(B)}K'$.*

Proof: By nested induction, first on $m(A) + m([M_0/x]_A(B))$, and then on the structure of the involved expressions (T, F and E_0, respectively). There are many cases, but they are all straightforward. ■

5 Type system

We describe the syntax of the 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

Next the assertion logic judgment.

$$\Delta \Longrightarrow P \quad \text{assuming all propositions in } \Delta \text{ are true, then } P \text{ is true}$$

And the formation judgments.

$\vdash \Delta\ \mathsf{ctx}\,[\Delta']$	Δ is a variable context, and Δ' is its canonical form
$\Delta \vdash A \Leftarrow \mathsf{type}\,[A']$	A is a type, and A' is its canonical form

Context formation. Here we define the judgment $\vdash \Delta\ \mathsf{ctx}\,[\Delta']$ for context formation.

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

We write $\Delta \vdash \Delta_1 \Leftarrow \mathsf{ctx}\,[\Delta_1']$ as a shorthand for $\vdash \Delta, \Delta_1\ \mathsf{ctx}\,[\Delta, \Delta_1']$.

Type formation. The judgment for type formation is $\Delta \vdash A \Leftarrow \mathsf{type}\,[A']$. It is assumed that $\vdash \Delta\ \mathsf{ctx}\,[\Delta]$. The rules are self-explanatory. We emphasize only that the formation for Hoare type $\{P\}x{:}A\{Q\}$ requires that the precondition P and the postcondition Q are not just assertions, but are predicates. For example, $P{:}\mathsf{heap}{\rightarrow}\mathsf{prop}$ so that P can express the properties of the heap that exists before the computation starts. On the other hand, the postcondition Q depends on two heaps (i.e., $Q{:}\mathsf{heap}{\rightarrow}\mathsf{heap}{\rightarrow}\mathsf{prop}$) so that it can relate the initial heap with the ending heap.

$$\frac{\Delta \vdash K \Rightarrow \mathsf{mono}\,[N']}{\Delta \vdash K \Leftarrow \mathsf{type}\,[N']} \qquad \frac{}{\Delta \vdash \mathsf{nat} \Leftarrow \mathsf{type}\,[\mathsf{nat}]} \qquad \frac{}{\Delta \vdash \mathsf{bool} \Leftarrow \mathsf{type}\,[\mathsf{bool}]} \qquad \frac{}{\Delta \vdash \mathsf{prop} \Leftarrow \mathsf{type}\,[\mathsf{prop}]}$$

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

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

$$\frac{\Delta \vdash P \Leftarrow \mathsf{heap} \rightarrow \mathsf{prop}\,[P'] \qquad \Delta \vdash A \Leftarrow \mathsf{type}\,[A'] \qquad \Delta, x{:}A' \vdash Q \Leftarrow \mathsf{heap} \rightarrow \mathsf{heap} \rightarrow \mathsf{prop}\,[Q']}{\Delta \vdash \{P\}x{:}A\{Q\} \Leftarrow \mathsf{type}\,[\{P'\}x{:}A'\{Q'\}]}$$

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

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

The rules for the primitive operations are self-explanatory, and we present them first. We use the auxiliary functions plus, times and equals defined in Section 4 in order to compute canonical forms of expressions involving primitive operations.

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

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

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

Checking assertion well-formedness.

$$\frac{\Delta \vdash A \Leftarrow \mathsf{type}\,[A'] \qquad \Delta \vdash B \Leftarrow \mathsf{type}\,[B'] \qquad \Delta \vdash M \Leftarrow A'\,[M'] \qquad \Delta \vdash N \Leftarrow B'\,[N']}{\Delta \vdash \mathsf{xid}_{A,B}(M,N) \Leftarrow \mathsf{prop}\,[\mathsf{xid}_{A',B'}(M',N')]}$$

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

$$\frac{\Delta \vdash M \Leftarrow \mathsf{prop}\,[M'] \qquad \Delta \vdash N \Leftarrow \mathsf{prop}\,[N']}{\Delta \vdash M \wedge N \Leftarrow \mathsf{prop}\,[M' \wedge N']} \qquad \frac{\Delta \vdash M \Leftarrow \mathsf{prop}\,[M'] \qquad \Delta \vdash N \Leftarrow \mathsf{prop}\,[N']}{\Delta \vdash M \vee N \Leftarrow \mathsf{prop}\,[M' \vee N']}$$

$$\frac{\Delta \vdash M \Leftarrow \mathsf{prop}\,[M'] \qquad \Delta \vdash N \Leftarrow \mathsf{prop}\,[N']}{\Delta \vdash M \supset N \Leftarrow \mathsf{prop}\,[M' \supset N']} \qquad \frac{\Delta \vdash M \Leftarrow \mathsf{prop}\,[M']}{\Delta \vdash \neg M \Leftarrow \mathsf{prop}\,[\neg M']}$$

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

Checking monotypes.

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

$$\frac{\Delta \vdash \tau \Leftarrow \mathsf{mono}\,[\tau'] \qquad \Delta, x{:}\tau' \vdash \sigma \Leftarrow \mathsf{mono}\,[\sigma']}{\Delta \vdash \Pi x{:}\tau.\,\sigma \Leftarrow \mathsf{mono}\,[\Pi x{:}\tau'.\,\sigma']} \qquad \frac{\Delta \vdash \tau \Leftarrow \mathsf{mono}\,[\tau'] \qquad \Delta, x{:}\tau' \vdash \sigma \Leftarrow \mathsf{mono}\,[\sigma']}{\Delta \vdash \Sigma x{:}\tau.\,\sigma \Leftarrow \mathsf{mono}\,[\Sigma x{:}\tau'.\,\sigma']}$$

$$\frac{\Delta \vdash P \Leftarrow \mathsf{heap} \to \mathsf{prop}\,[P'] \qquad \Delta \vdash \tau \Leftarrow \mathsf{mono}\,[\tau'] \qquad \Delta, x{:}\tau' \vdash Q \Leftarrow \mathsf{heap} \to \mathsf{heap} \to \mathsf{prop}\,[Q']}{\Delta \vdash \{P\}x{:}\tau\{Q\} \Leftarrow \mathsf{mono}\,[\{P'\}x{:}\tau'\{Q'\}]}$$

$$\frac{\Delta \vdash \tau \Leftarrow \mathsf{mono}\,[\tau'] \qquad \Delta, x{:}\tau' \vdash P \Leftarrow \mathsf{prop}\,[P']}{\Delta \vdash \{x{:}\tau.\,P\} \Leftarrow \mathsf{mono}\,[\{x{:}\tau'.\,P'\}]}$$

Before we can state the rules for the composite types, we need several auxiliary functions. For example, the function $\mathsf{apply}_A(M,N)$ normalizes the application $M\ N$. Here, A is a canonical type, and the arguments M and N are canonical intro terms. If M is a lambda abstraction, the redex $M\ N$ is immediately normalized by substituting N hereditarily in the body of the lambda expression. If M is an elim term, there is no redex, and the application is returned unchanged. In other cases, apply is not defined, but such cases cannot arise during typechecking, where these functions are only applied to well-typed arguments. Similarly, we need functions for reducing first and second projections and coercions out of subset types. We also define a function exp which eta expands its argument, if the argument is an elim form (and does nothing if the argument is intro).

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

$$\begin{array}{lcll}
\mathsf{first}(K) & = & \mathsf{fst}\ K & \text{if } K \text{ is an elim term} \\
\mathsf{first}((M, N)) & = & M & \\
\mathsf{first}(M) & & \text{fails} & \text{otherwise}
\end{array}$$

$$\begin{array}{lcll}
\mathsf{second}(K) & = & \mathsf{snd}\ K & \text{if } K \text{ is an elim term} \\
\mathsf{second}((M, N)) & = & N & \\
\mathsf{second}(M) & & \text{fails} & \text{otherwise}
\end{array}$$

$$\begin{array}{lcll}
\mathsf{extract}(K) & = & \mathsf{out}\ K & \text{if } K \text{ is an elim term} \\
\mathsf{extract}(\mathsf{in}\ M) & = & M & \\
\mathsf{extract}(M) & & \text{fails} & \text{otherwise}
\end{array}$$

$$\begin{array}{lcll}
\mathsf{exp}_A(K) & = & N' & \text{if } K \text{ is an elim term} \\
 & & & \text{and } N' = \mathsf{expand}_A(K) \text{ exists} \\
\mathsf{exp}_A(M) & = & M & \text{if } M \text{ is an intro term} \\
\mathsf{exp}_A(M) & & \text{fails} & \text{otherwise}
\end{array}$$

Now we can present the rest of the typing rules for terms. In general, the introduction inference rules break down a type when read from the conclusion to the premise. If the conclusion type is given, we can determine the types for the premises, and proceed to check them. Thus, intro terms should be checked. The elimination rules break down the type when read from premise to the conclusion. In the base case, the type of a variable can be read off from the context, and therefore, elim terms can always synthesize their types.

Moreover, if the types in contexts are canonical (as they are in HTT), then so are the synthesized types.

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

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

$$\frac{\Delta \vdash M \Leftarrow A\,[M'] \qquad \Delta \vdash N \Leftarrow [M'/x]^a_A(B)\,[N']}{\Delta \vdash (M, N) \Leftarrow \Sigma x{:}A.\, B\,[(M', N')]}\ \Sigma\mathsf{I}$$

$$\frac{\Delta \vdash K \Rightarrow \Sigma x{:}A.\, B\,[N']}{\Delta \vdash \mathsf{fst}\ K \Rightarrow A\,[\mathsf{first}(N')]}\ \Sigma\mathsf{E1} \qquad \frac{\Delta \vdash K \Rightarrow \Sigma x{:}A.\, B\,[N']}{\Delta \vdash \mathsf{snd}\ K \Rightarrow [\mathsf{exp}_A(\mathsf{first}(N'))/x]^a_A(B)\,[\mathsf{second}(N')]}\ \Sigma\mathsf{E2}$$

$$\frac{\Delta \vdash M \Leftarrow A\,[M'] \qquad \Delta \Longrightarrow [M'/x]^p_A(P)}{\Delta \vdash \mathsf{in}\ M \Leftarrow \{x{:}A.\, P\}\,[\mathsf{in}\ M']}\ \{\}\mathsf{I}$$

$$\frac{\Delta \vdash K \Rightarrow \{x{:}A.\, P\}\,[N']}{\Delta \vdash \mathsf{out}\ K \Rightarrow A\,[\mathsf{extract}(N')]}\ \{\}\mathsf{E1} \qquad \frac{\Delta \vdash K \Rightarrow \{x{:}A.\, P\}\,[N']}{\Delta \Longrightarrow [\mathsf{exp}_A(\mathsf{extract}(N'))/x]^p_A(P)}\ \{\}\mathsf{E2}$$

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

$$\frac{\Delta \vdash L \Rightarrow K'\,[L] \qquad K = K'}{\Delta \vdash \mathsf{eta}_K\ L \Leftarrow K\,[\mathsf{eta}_K\ L]}\ \mathsf{eta}$$

In the rule ΠI, the type $\Pi x{:}A.\, B$ is assumed to be canonical, so that in the premise, A can be placed into the context without any additional checks. On the other hand, the rule ΠE hereditarily substitutes the canonical M' for x into B, so that the synthesized type remains canonical.

Similar pattern applies to other rules as well. For example, the introduction rules for subset types $\{\,\}$I must make sure that the term satisfies the subset predicate. The first elimination rule for subset types $\{\,\}$E1, gives us the information about the base type of K. The second elimination rule $\{\,\}$E2 gives us a derivation in the assertion logic that $\Delta \Longrightarrow [\mathsf{exp}_A(\mathsf{extract}(N'))/x]^p_A(P)$, which essentially states that the underlying term of K satisfies the predicate P. The occurrences of exp and extract serve to ensure that all the involved terms are canonical, thus baking the beta and eta equations into the assertion logic.

When checking an intro term that happens to be elim (i.e. has form K) against a type B, we synthesize the type for K and explicitly compare with B, as formalized in the rule $\Rightarrow\Leftarrow$. This comparison is a simple alpha-conversion, but since the types are canonical, it accounts for beta and eta equivalence.

Conversely, given an elim term that happens to be intro (i.e. has the form $M{:}A$), its synthesized type is the canonical version of A, assuming that M checks against it, as formalized in the rule $\Leftarrow\Rightarrow$. Notice that the canonical form of $M{:}A$ equals the canonical version of M itself, but does not include the typing annotation. We have previously intuitively described the process of computing the canonical forms as a systematic removal of the constructor $M{:}A$ from the involved term. Now we can point the rule $\Leftarrow\Rightarrow$ as precisely the place where this removal occurs.

We mention an additional complication in the rule $\Rightarrow\Leftarrow$. When the types A and B are actually elimination forms (e.g., they may be variables), we cannot compute the canonical form of the checked term by eta expanding it. Indeed, the exact form of A and B may change as we substitute into these types, and thus the form of the expansion may change with substitutions. This is why we introduce the constructor $\mathsf{eta}_K\ L$ which essentially records that L should be eta expanded with respect to the type K once and if the type K is turned – by means of some substitution – into an ordinary type like nat, bool, $\Pi x{:}\tau_1.\, \tau_2$, etc. We note here that K must be a type (i.e. $K \Leftarrow \mathsf{type}$) by assumption on well-formedness of the types supplied to the checking judgment. But, furthermore, from here we infer that $K \Leftarrow \mathsf{mono}$, because the only way an elim form may be a type is by coercion from monotypes, as apparent from the formation rules for types given a couple of paragraphs above.

As we mentioned in Section 2, the constructor $\mathsf{eta}_K\ L$ is not supposed to be used in the source language, but we need it when we work with canonical forms, as illustrated above. Thus, since $\mathsf{eta}_K\ L$ is only interesting in the canonical setting, we restrict K and L to only be canonical. This is achieved by requiring in the premises of the eta rule that L equals its canonical form, and that K is its synthesized type (which is canonical by design).

Rules and axioms of the assertion logic. The judgment which establishes the provability of propositions in the assertion logic is $\Delta \Longrightarrow P$, where we assume that both Δ and P are canonical. The assumed canonicity of these inputs will avoid the need to introduce explicit rules for dealing with beta and eta equalities.

The judgment is formulated in a natural deduction style. Although the notation using $\Longrightarrow$ strongly suggests a sequent calculus formulation, this is not the case. We only use this notation to retain and emphasize the connection with the formulation of the assertion logic from our previous work [33].

Begin a formulation of higher-order logic, this judgment need only a handful of inference rules that deal with the basic constructors for implication and universal quantification. The rules for the rest of the constructs may simply be encoded as axioms.

The inference rules are fairly standard, and we list them below.

$$\frac{}{\Delta, P, \Delta_1 \Longrightarrow P}$$

$$\frac{\Delta, P \Longrightarrow Q}{\Delta \Longrightarrow P \supset Q} \qquad \frac{\Delta \Longrightarrow P \supset Q \qquad \Delta \Longrightarrow P}{\Delta \Longrightarrow Q}$$

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

The rules should be read in a bottom-up manner, defining what it means to be a proof of the assertion given in the conclusion. The concluding proposition is always assumed well-formed, so we do not need to check for it. However, at the elimination rules, we need to come up with a proposition P that is not mentioned in the conclusion. We explicitly mention here that we only consider for P propositions that are well-typed, so that assumptions about the typing of P need not be included explicitly in the typing rules.

For the rest of the propositional constructors, we provide the schemas for the introduction and elimination. (We note here that we can go one step further and define most of the constructs using Church encoding, but that's largely irrelevant for the development here.)

$$\begin{array}{rcl}
\mathsf{topi} & : & \top \\
\mathsf{bote} & : & \forall p{:}\mathsf{prop}.\, \bot \supset p \\
\mathsf{andi} & : & \forall p, q{:}\mathsf{prop}.\, p \supset q \supset p \wedge q \\
\mathsf{ande1} & : & \forall p, q{:}\mathsf{prop}.\, p \wedge q \supset p \\
\mathsf{ande2} & : & \forall p, q{:}\mathsf{prop}.\, p \wedge q \supset q \\
\mathsf{ori1} & : & \forall p, q{:}\mathsf{prop}.\, p \supset p \vee q \\
\mathsf{ori2} & : & \forall p, q{:}\mathsf{prop}.\, q \supset p \vee q \\
\mathsf{ore} & : & \forall p, q, r{:}\mathsf{prop}.\, p \vee q \supset (p \supset r) \supset (q \supset r) \supset r \\
\mathsf{noti} & : & \forall p{:}\mathsf{prop}.\, (p \supset \bot) \supset \neg p \\
\mathsf{note} & : & \forall p, q{:}\mathsf{prop}.\, \neg p \supset p \supset q \\
\mathsf{exi}_A & : & \forall x{:}A.\, \forall p{:}A {\to} \mathsf{prop}.\, p\ x \supset \exists x{:}A.\, p\ x \\
\mathsf{exe}_A & : & \forall p{:}A {\to} \mathsf{prop}.\, \forall q{:}\mathsf{prop}.\, (\exists x{:}A.\, p\ x) \supset (\forall x{:}A.\, p\ x \supset q) \supset q
\end{array}$$

Heterogeneous equality:

$$\begin{array}{rcl}
\mathsf{xidi}_A & : & \forall x{:}A.\, \mathsf{xid}_{A,A}(x, x) \\
\mathsf{xide}_A & : & \forall p{:}A {\to} \mathsf{prop}.\, \forall x, y{:}A.\, \mathsf{xid}_{A,A}(x, y) \supset p\ x \supset p\ y
\end{array}$$

Function and pair extensionality:

$$\begin{array}{rcl}
\mathsf{extfunc}_{\Pi x:A.\, B} & : & \forall f, g{:}\Pi x{:}A.\, B.\, (\forall x{:}A.\, \mathsf{id}_B(f\ x, g\ x)) \supset \mathsf{id}_{\Pi x:A.\, B}(f, g) \\
\mathsf{extpair}_{\Sigma x:A.\, B} & : & \forall s, t{:}\Sigma x{:}A.\, B.\, \mathsf{id}_A(\mathsf{fst}\ s, \mathsf{fst}\ t) \supset \mathsf{xid}_{[\mathsf{fst}\ s/x]^a_A(B), [\mathsf{fst}\ t/x]^a_A(B)}(\mathsf{snd}\ s, \mathsf{snd}\ t) \supset \mathsf{id}_{\Sigma x:A.\, B}(s, t)
\end{array}$$

It should be clear that the schemas which depend on the type A are not well-formed canonical expressions in HTT. In fact, for each given type A, the schema takes a different canonical form which depends on eta expansion with respect to A. For example, xide should really look like the term below, where the occurrences of expand_A may not be substituted by eta_A, because A need not be a small type.

$$\forall p{:}A {\to} \mathsf{prop}.\, \forall x, y{:}A.\, \mathsf{xid}_{A,A}(\mathsf{expand}_A\ x, \mathsf{expand}_A\ y) \supset p\ (\mathsf{expand}_A\ x) \supset p\ (\mathsf{expand}_A\ y).$$

We thus adopt an abuse of notation to abbreviate $\mathsf{expand}_A(x)$ into x in all well-typed contexts. We will eventually prove in Section 6, Lemma 9 that this does not lead to any difficulties. We will use this convention extensively in the axioms and typing rules that follow.

The rest of the rules is a fairly standard formulation of properties for the base types of natural numbers and booleans.

Peano arithmetic:

$$\begin{array}{rcl}
\mathsf{zneqs} & : & \forall x{:}\mathsf{nat}.\, \neg \mathsf{id}_{\mathsf{nat}}(z, \mathsf{s}\ x) \\
\mathsf{sinject} & : & \forall x, y{:}\mathsf{nat}.\, \mathsf{id}_{\mathsf{nat}}(\mathsf{s}\ x, \mathsf{s}\ y) \supset \mathsf{id}_{\mathsf{nat}}(x, y) \\
\mathsf{induct} & : & \forall p{:}\mathsf{nat} {\to} \mathsf{prop}.\, p\ \mathsf{z} \supset (\forall x{:}\mathsf{nat}.\, p\ x \supset p\ (\mathsf{s}\ x)) \supset \forall x{:}\mathsf{nat}.\, p\ x
\end{array}$$

Booleans:

$$\begin{array}{rcl}
\mathsf{tneqf} & : & \neg \mathsf{id}_{\mathsf{nat}}(\mathsf{true}, \mathsf{false}) \\
\mathsf{extbool} & : & \forall p{:}\mathsf{bool} {\to} \mathsf{prop}.\, p\ \mathsf{true} \supset p\ \mathsf{false} \supset \forall x{:}\mathsf{bool}.\, p\ x
\end{array}$$

The primitive type of propositions is axiomatized as follows. We first need an axiom which defines the propositional equality of propositions as a bi-implication. This essentially states that $\top$ and $\bot$ are different as propositions. Then we need a kind of an extensionality principle, which states that no other proposition can possibly exist, i.e. each proposition is only equal to $\top$ or $\bot$. This property precisely defines the classical nature of our assertion logic, and we formalize it with an axiom of excluded middle.

$$\begin{array}{rcl}\mathsf{propeq} & : & \forall p, q{:}\mathsf{prop}.\, (p \subset\supset q) \supset \mathsf{id}_{\mathsf{prop}}(p, q)\\ \mathsf{exmid} & : & \forall p{:}\mathsf{prop}.\, p \vee \neg p\end{array}$$

Finally, we axiomatize the properties of small types, by postulating that each type constructor is injective, and different from any other. For this purpose, we essentially treat mono as an inductive datatype. This gives us the following requirement for axiomatizing its properties:

1. We must make clear that each constructor of this type is injective
2. We must make clear that each constructor is different from any other
3. We must formulate the associated induction principle, which essentially describes that there is no other way to create a small type, except by using the provided constructors.

This properties are realized in the following set of axioms for small types.

$$\begin{array}{rcl}\mathsf{mononeq}_{\tau,\sigma} & : & \neg\mathsf{id}_{\mathsf{mono}}(\tau, \sigma) \quad \text{where } \tau, \sigma \in \{\mathsf{nat}, \mathsf{bool}, \mathsf{prop}, 1, \Pi x{:}\tau'.\, \sigma',\\ & & \qquad\qquad \Sigma x{:}\tau'.\, \sigma', \{P\}x{:}\tau'\{Q\}, \{x{:}\tau'.\, P\}\}\\ & & \qquad \text{and } \tau, \sigma \text{ have different top-level constructors}\\ \mathsf{injectprod} & : & \forall \tau_1, \tau_2{:}\mathsf{mono}.\, \forall \sigma_1{:}\tau_1{\to}\mathsf{mono}.\, \forall \sigma_2{:}\tau_2{\to}\mathsf{mono}.\\ & & \mathsf{id}_{\mathsf{mono}}(\Pi x{:}\tau_1.\, \sigma_1(x), \Pi x{:}\tau_2.\, \sigma_2(x)) \supset \mathsf{id}_{\mathsf{mono}}(\tau_1, \tau_2) \wedge \mathsf{xid}_{\tau_1\to\mathsf{mono}, \tau_2\to\mathsf{mono}}(\sigma_1, \sigma_2)\\ \mathsf{injectsum} & : & \forall \tau_1, \tau_2{:}\mathsf{mono}.\, \forall \sigma_1{:}\tau_1{\to}\mathsf{mono}.\, \forall \sigma_2{:}\tau_2{\to}\mathsf{mono}.\\ & & \mathsf{id}_{\mathsf{mono}}(\Sigma x{:}\tau_1.\, \sigma_1(x), \Sigma x{:}\tau_2.\, \sigma_2(x)) \supset \mathsf{id}_{\mathsf{mono}}(\tau_1, \tau_2) \wedge \mathsf{xid}_{\tau_1\to\mathsf{mono}, \tau_2\to\mathsf{mono}}(\sigma_1, \sigma_2)\\ \mathsf{injecthoare} & : & \forall \tau_1, \tau_2{:}\mathsf{mono}.\, \forall p_1, p_2{:}\mathsf{heap}{\to}\mathsf{prop}.\\ & & \forall q_1{:}\tau_1{\to}\mathsf{heap}{\to}\mathsf{heap}{\to}\mathsf{prop}.\\ & & \forall q_2{:}\tau_2{\to}\mathsf{heap}{\to}\mathsf{heap}{\to}\mathsf{prop}.\\ & & \mathsf{id}_{\mathsf{mono}}(\{p_1\}x{:}\tau_1\{q_1\ x\}, \{p_2\}x{:}\tau_2\{q_2\ x\}) \supset\\ & & \mathsf{id}_{\mathsf{heap}\to\mathsf{prop}}(p_1, p_2) \wedge \mathsf{id}_{\mathsf{mono}}(\tau_1, \tau_2) \wedge \mathsf{xid}_{\tau_1\to\mathsf{heap}\to\mathsf{heap}\to\mathsf{mono}, \tau_2\to\mathsf{heap}\to\mathsf{heap}\to\mathsf{mono}}(q_1, q_2)\\ \mathsf{injectsubset} & : & \forall \tau_1, \tau{:}\mathsf{mono}.\, \forall p_1{:}\tau_1{\to}\mathsf{prop}.\, \forall p_2{:}\tau_2{\to}\mathsf{prop}.\\ & & \mathsf{id}_{\mathsf{mono}}(\{x{:}\tau_1.\, p_1\}, \{x{:}\tau_2.\, p_2\}) \supset \mathsf{id}_{\mathsf{mono}}(\tau_1, \tau_2) \wedge \mathsf{xid}_{\tau_1\to\mathsf{prop}, \tau_2\to\mathsf{prop}}(p_1, p_2)\\ \mathsf{monoinduct} & : & \forall p{:}\mathsf{mono}{\to}\mathsf{prop}.\\ & & p\ \mathsf{nat} \supset p\ \mathsf{bool} \supset p\ \mathsf{prop} \supset p\ 1 \supset\\ & & (\forall \tau{:}\mathsf{mono}.\, \forall \sigma{:}\tau{\to}\mathsf{mono}.\, p\ \tau \supset (\forall x{:}\tau.\, p\ (\sigma\ x)) \supset p\ (\Pi x{:}\tau.\, \sigma\ x)) \supset\\ & & (\forall \tau{:}\mathsf{mono}.\, \forall \sigma{:}\tau{\to}\mathsf{mono}.\, p\ \tau \supset (\forall x{:}\tau.\, p\ (\sigma\ x)) \supset p\ (\Sigma x{:}\tau.\, \sigma\ x)) \supset\\ & & (\forall q_1{:}\mathsf{heap}{\to}\mathsf{prop}.\, \forall \tau{:}\mathsf{mono}.\, \forall q_2{:}\tau{\to}\mathsf{heap}{\to}\mathsf{prop}.\, p\ \tau \supset p\ (\{q_1\}x{:}\tau\{q_2\})) \supset\\ & & (\forall \tau{:}\mathsf{mono}.\, \forall q{:}\tau{\to}\mathsf{prop}.\, p\ \tau \supset p\ (\{x{:}\tau.\, q\}) \supset\\ & & \forall \tau{:}\mathsf{mono}.\, p\ \tau\end{array}$$

Computations. The judgments for typechecking computations are $\Delta; P \vdash E \Rightarrow x{:}A.\, Q\, [E']$ and $\Delta; P \vdash E \Leftarrow x{:}A.\, Q\, [E']$, where we assume that Δ, P, A and Q are canonical. Moreover, $P, Q : \mathsf{heap}{\to}\mathsf{heap}{\to}\mathsf{prop}$.

A computation E may be seen as a heap transformer, turning the input heap into the output heap if it terminates. The judgment $\Delta; P \vdash E \Rightarrow x{:}A.\, Q\, [E']$ essentially converts E into a binary relation on heaps, so that the assertion logic can reason about E using standard mathematical machinery.

The predicates $P, Q{:}\mathsf{heap}{\to}\mathsf{heap}{\to}\mathsf{prop}$ represent binary heap relations. P is the starting relation onto which the typing rules build as they convert E one command at a time. The generated strongest postcondition Q will be the relation that, intuitively, most precisely captures the semantics of E.

The judgment $\Delta; P \vdash E \Leftarrow x{:}A.\, Q\, [E']$ checks if Q is a postcondition for E, by generating the strongest postcondition S and then trying to prove the implication $S \Longrightarrow Q$ in the assertion logic.

We note at this point that, although we refer to Q above as "the strongest postcondition", we do not formally prove that Q is indeed the strongest predicate describing the ending state of the computation. As a matter of fact, it need not be! The derivation of Q will at certain places take into account assertions that are explicitly given in E (e.g. as invariants of other computations or recursive functions), but which themselves are not the strongest possible. The situation is similar to the one encountered during verification of first-order code, where computing a postcondition of a loop needs to take the supplied loop invariant into account. If the loop invariant is not the strongest possible, then the computed postcondition will not, semantically, be the strongest one either, although it will be the strongest one with respect to the invariant. We believe that the problem of computing semantically strongest postconditions has to be attempted in a setting where the computations are not annotated with assertions, and thus naturally falls into the domain of type and annotation inference.

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

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

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

We further require an auxiliary function to beta reduce eventual redexes created by hereditary substitutions.

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

Now we can present the typing rules for computations. We start first with the generic monadic fragment.

$$\frac{\Delta; P \vdash E \Rightarrow x{:}A.\, S\, [E'] \qquad \Delta, x{:}A, i{:}\mathsf{heap}, m{:}\mathsf{heap}, (S\ i\ m) \Longrightarrow (Q\ i\ m)}{\Delta; P \vdash E \Leftarrow x{:}A.\, Q\, [E']}\ \mathsf{consequent}$$

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

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

$$\frac{\begin{array}{c}\Delta \vdash K \Rightarrow \{R\}x{:}A\{S\}\, [N'] \\ \Delta, i{:}\mathsf{heap}, m{:}\mathsf{heap}, (P\ i\ m) \Longrightarrow (R * \lambda m'.\, \top)(m) \qquad \Delta, x{:}A; P \circ (R \gg S) \vdash E \Rightarrow y{:}B.\, Q\, [E']\end{array}}{\Delta; P \vdash \mathsf{let\ dia}\ x = K\ \mathsf{in}\ E \Rightarrow y{:}B.\, (\lambda i.\, \lambda m.\, \exists x{:}A.\, (Q\ i\ m))\, [\mathsf{reduce}_A(N', x.\, E')]}\ \{\,\}\mathsf{E}$$

The rule $\mathsf{consequent}$ defines the judgment $\Delta; P \vdash E \Leftarrow x{:}A.\,Q$ in terms of $\Delta; P \vdash E \Rightarrow x{:}A.\,S$. As discussed before, in order to check that Q is a postcondition for E, we generate the strongest postcondition S and simply check that S implies Q.

The rule comp defines the strongest postcondition for the trivial, pure, computation M. This postcondition must include the precondition P, (as executing M does not change the heap). But also, the postcondition must state that M is the return value of the overall computation, which it does by equating (the canonical form of) M with (the canonical form of) x.

We reiterate at this occasion that all the variables appearing in the assertions in these typing rules should be eta expanded, but according to our assumed convention described previously in description of the assertion logic, we omit the explicit expansions. This abuse of notation is justified by the Lemma 9 in Section 6.

The rule for introducing dia, checks if $\mathsf{dia}\ E$ has type $\{R\}x{:}A\{Q\}$. This check essentially verifies that E has a postcondition $R \gg Q$, i.e. that the behavior of E can be described via the relation Q, if it is restricted to a subfragment of the input heap that satisfies R. Because R is a relation over one heap, and the typing judgments require a relation on two heaps, the checking is initialized with a diagonal relation on heaps (the condition $i = m$ in the premise). Furthermore, we require that the global heap can be proved to contain a sub-fragment satisfying R (the condition $(R * \lambda h'.\,\top)(m)$ in the premise). These requirements specified in the precondition and the postcondition capture precisely the small footprint nature of the specification $\{R\}x{:}A\{Q\}$.

To check $\mathsf{let\ dia}\ x = K\ \mathsf{in}\ E$, where K has type $\{R\}x{:}A\{S\}$, we must first prove that the beginning heap contains a sub-fragment satisfying R (the condition $(P\ i\ m) \Longrightarrow (R * \lambda h'.\,\top)(m)$), so that the precondition for K is satisfied, and K can actually be executed. The execution of K changes the heap so that it can be described by the composition $P \circ (R \gg S)$, which is then used in the checking of E.

Next we present the rules for the primitive commands. The rules for lookup and (strong) update are easily described.

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

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

Before the lookup $x = !_\tau\, M$, we must prove that M points to a value of type τ at the beginning (the condition $(P\ i\ m) \Longrightarrow M' \hookrightarrow_{\tau'} -)$. After the lookup, the heap looks exactly as before (the condition $P\ i\ m$) but we also know that x equals the content of M (the condition $M' \hookrightarrow_{\tau'} \mathsf{exp}_{\tau'}(x)$).

Before the update $M :=_\tau N$, we must prove that M is allocated and initialized with some value of *with an arbitrary type* (the condition $M' \hookrightarrow -$). After the lookup, the old value is removed from the heap, and replaced with N (the condition $P \circ ((M' \mapsto -) \multimap (M' \mapsto_{\tau'} N'))$).

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

$$\frac{\begin{array}{c}\Delta \vdash M \Leftarrow \mathsf{bool}\,[M'] \qquad \Delta \vdash A \Leftarrow \mathsf{type}\,[A'] \qquad \Delta; \lambda i.\,\lambda m.\,(P\ i\ m) \wedge \mathsf{id}_{\mathsf{bool}}(M', \mathsf{true}) \vdash E_1 \Rightarrow x{:}A'.\,P_1\,[E_1'] \\ \Delta; \lambda i.\,\lambda m.\,(P\ i\ m) \wedge \mathsf{id}_{\mathsf{bool}}(M', \mathsf{false}) \vdash E_2 \Rightarrow x{:}A'.\,P_2\,[E_2'] \\ \Delta, x{:}A'; \lambda i.\,\lambda m.\,(P_1\ i\ m) \vee (P_2\ i\ m) \vdash E \Rightarrow y{:}B.\,Q\,[E']\end{array}}{\Delta; P \vdash x = \mathsf{if}_A\ M\ \mathsf{then}\ E_1\ \mathsf{else}\ E_2; E \Rightarrow y{:}B.\,(\lambda i.\,\lambda m.\,\exists x{:}A'.\,(Q\ i\ m))\,[x = \mathsf{if}_{A'}\ M'\ \mathsf{then}\ E_1'\ \mathsf{else}\ E_2'; E']}$$

Similar explanation can be given for the typing rule for case.

$$\frac{\begin{array}{c}\Delta \vdash M \Leftarrow \mathsf{nat}\,[M'] \qquad \Delta \vdash A \Leftarrow \mathsf{type}\,[A'] \qquad \Delta; \lambda i.\, \lambda m.\, (P\ i\ m) \wedge \mathsf{id}_{\mathsf{nat}}(M', \mathsf{z}) \vdash E_1 \Rightarrow x{:}A'.\, P_1\,[E_1'] \\ \Delta, y{:}\mathsf{nat}; \lambda i.\, \lambda m.\, (P\ i\ m) \wedge \mathsf{id}_{\mathsf{nat}}(M', \mathsf{s}\ y) \vdash E_2 \Rightarrow x{:}A'.\, P_2\,[E_2'] \\ \Delta, x{:}A'; \lambda i.\, \lambda m.\, (P_1\ i\ m) \vee \exists y{:}\mathsf{nat}.\, (P_2\ i\ m) \vdash E \Rightarrow v{:}B.\, Q\,[E']\end{array}}{\begin{array}{r}\Delta; P \vdash x = \mathsf{case}_A\ M\ \mathsf{of}\ \mathsf{z} \Rightarrow E_1\ \mathsf{or}\ \mathsf{s}\ y \Rightarrow E_2; E \Rightarrow v{:}B.\, (\lambda i.\, \lambda m.\, \exists x{:}A'.\, (Q\ i\ m)) \\ [x = \mathsf{case}_{A'}\ M'\ \mathsf{of}\ \mathsf{z} \Rightarrow E_1'\ \mathsf{or}\ \mathsf{s}\ y \Rightarrow E_2'; E']\end{array}}$$

Finally, we present the rule for the recursion construct $\mathsf{fix}\ f(x : A) : T = \mathsf{dia}\, E\ \mathsf{in}\ \mathsf{eval}\, f\, M$. This construct defines the function f and then immediately applies it to M to obtain a computation. The computation is evaluated and its result is returned as the overall result of the fixpoint construct.

Here the type T must be a Hoare type, and the inference rule must check that the canonical form for T equals $\{R\}x{:}B\{S\}$ for some predicates R and S.

$$\frac{\begin{array}{c}\Delta \vdash A \Leftarrow \mathsf{type}\,[A'] \qquad \Delta, x{:}A' \vdash T \Leftarrow \mathsf{type}\,[T'] \qquad T' = \{R\}y{:}B\{S\} \\ \Delta \vdash M \Leftarrow A\,[M'] \qquad \Delta, i{:}\mathsf{heap}, m{:}\mathsf{heap}, (P\ i\ m) \Longrightarrow [M'/x]^p_A(R_1 * \lambda h.\, \top)(m) \\ \Delta, x{:}A', f{:}\Pi x{:}A'.\, T'; \lambda i.\, \lambda m.\, i = m \wedge (R * \lambda h.\, \top)(m) \vdash E \Leftarrow y{:}B.\, (R \gg S)\,[E'] \\ \Delta, y{:}[M'/x]^p_A(B); P \circ [M'/x]^p_A(R \gg S) \vdash F \Rightarrow z{:}C.\, Q\,[F']\end{array}}{\begin{array}{r}\Delta; P \vdash y = \mathsf{fix}\ f(x{:}A){:}T = \mathsf{dia}\, E\ \mathsf{in}\ \mathsf{eval}\, f\, M; F \Rightarrow z{:}C.\, (\lambda i.\, \lambda m.\, \exists y{:}[M'/x]^p_A(B).(Q\ i\ m)) \\ [y = \mathsf{fix} f(x{:}A'){:}T' = \mathsf{dia}\, E'\ \mathsf{in}\ \mathsf{eval}\, f\, M'; F']\end{array}}$$

Before M can be applied to the recursive function, and the obtained computation executed, we need to check that the main precondition P implies $R * \lambda h.\, \top$. This means that the heap contains a fragment that satisfies R, so that the computation obtained from the recursive function can actually be executed. After the recursive call we are in a heap that is changed according to the proposition $R \gg S$, because this predicate describes in a most general way the behavior of the computation obtained by the recursive function. Thus, we proceed to test F with a precondition $P \circ (R \gg S)$. Of course, because the recursive calls are started using M for the argument x, we need to substitute the canonical form M' for x everywhere.

6 Substitution principles and other properties

The development of the meta-theory of HTT is split into two stages. First, we need to prove the substitution principles and the associated lemmas for the fragment containing canonical forms only. Then, using these results, we can prove the substitution principle for general forms.

In fact, it may be said that HTT can be viewed as really consisting of two layers. The first layer contains only canonical forms, and is the carrier of the semantics of HTT. The second layer contains general forms (and in particular, contains the constructor $M{:}A$), and is used as a convenience in programming, where we do not necessarily want to only work with canonical forms. But, it should be clear that it is the first layer that is the most important.

Even if the general structure of the meta theory of HTT is inherited from the previous work, the development in the current paper is slightly more complicated because of the additions of higher-order features, like the types prop and mono. For example, one complication that arises here is that the function expand now becomes mutually recursive with the hereditary substitutions, leading to an entanglement of the identity and substitution principles.

First of all, we start the development by noting, without a formal statement, that the HTT judgments all satisfy the basic structural properties of weakening, contraction and (dependency preserving) exchange of variable ordering in the context.

We proceed then to establish a basic lemma about the expansion of variables, which shows that in well-typed contexts, a variable $x{:}A$ behaves generally like the expansion $\mathsf{expand}_A(x)$. We will use this property to justify our abuse of notation to abbreviate $\mathsf{expand}_A(x)$ with x only. For example, we write $\mathsf{xid}_{A,B}(x, y)$ instead of $\mathsf{xid}_{A,B}(\mathsf{expand}_A(x), \mathsf{expand}_B(y))$. We have already used this convention in Section 5 when we formulated the axioms and the inference rules of the assertion logic, and the type rules for computations.

In the lemma, we assume that the involved expressions are canonical. We make the canonicity assumption explicit by requiring that each term is well typed and equal to its own canonical form.

Lemma 9 (Properties of variable expansion)
Suppose that $\mathsf{expand}_A(x)$ exists. Then the following holds:

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 $[\mathsf{expand}_A(x)/x]^k_A(K) = N' :: B'$ is an intro term, then $B' = B$ and $N' = \mathsf{expand}_B(K)$.*
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 $[\mathsf{expand}_A(x)/x]^a_A(B) = B$.*
5. *If $\Delta \vdash M \Leftarrow A\,[M]$, then $[M/x]^m_A(\mathsf{expand}_A(x)) = M$.*
6. *If $\Delta; P \vdash E \Leftarrow x{:}A.\,Q\,[E]$, then $\langle E/x\rangle_A(\mathsf{expand}_A(x)) = E$.*

Proof: By mutual nested induction, first on $m(A)$, and then on the structure of the expressions involved. In other words, we will invoke the induction hypotheses either with a strictly smaller metric, but possibly larger expression, or with the same metric, but strictly smaller expression. The characteristic case of the lemma is 1(b), and we present its proof in more detail.

In this case, we know $\mathsf{head}(K) = x$. When $K = x$, the statement is obvious. Consider $K = L\ M$.

By typing:

$$\begin{array}{rcl} L & \Rightarrow & \Pi y{:}B_1.\,B_2 \\ M & \Leftarrow & B_1 \\ B & = & [M/x]^a_{B_1}(B_2) \end{array}$$

By i.h. on L and M:

$$\begin{array}{rcl} [\mathsf{expand}(x)/x]^k_A(L) & = & \mathsf{expand}_{\Pi y:B_1.\,B_2}(L) \ = \ \lambda y.\,\mathsf{expand}_{B_2}(L\ \mathsf{expand}_{B_1}(y)) :: \Pi y{:}B_1.\,B_2 \\ [\mathsf{expand}(x)/x]^m_A(M) & = & M \end{array}$$

As a consequence of the first equation, we also get by the termination theorem that $m(\Pi y{:}B_1.\,B_2) \leq m(A)$.

Now, we compute:

$$\begin{array}{rcl} [\mathsf{expand}(x)/x]^k_A(L\ M) & = & [M/y]^m_{B_1}(\mathsf{expand}_{B_2}(L\ \mathsf{expand}_{B_1}(y)) :: [M/y]^a_{B_1}(B_2) \\ & = & \mathsf{expand}_B(L\ ([M/y]^m_{B_1}(\mathsf{expand}_{B_1}(y)))) :: B \quad \text{because } [M/y]^a_{B_1}(B_2) = B \\ & = & \mathsf{expand}_B(L\ M) :: B \quad \text{by i.h on } m(B_1) < m(\Pi y{:}B_1.\,B_2) \leq m(A) \\ & = & \mathsf{expand}_B(K) :: B \end{array}$$

which proves the case.

■

The next lemma allows us to remove a proposition from a context if the proposition is provable. One may view this property as a substitution principle for propositions. It is worth noting that this property is not mutually dependent with the other substitution principles, because in HTT we deal with provability, but not with proofs themselves. Thus, the removal of a proposition from a context does not impose any structural changes to the involved judgments, and thus there is no need to invoke any other substitution principles in the proof of this lemma.

Lemma 10 (Propositional substitution principles)
Suppose that $\Delta \Longrightarrow R$. Then the following holds.

1. *If $\Delta, R \vdash K \Rightarrow B\,[K]$, then $\Delta \vdash K \Rightarrow B\,[K]$.*
2. *If $\Delta, R \vdash N \Leftarrow B\,[N]$, then $\Delta \vdash N \Leftarrow B\,[N]$.*
3. *If $\Delta, R; P \vdash E \Leftarrow y{:}B.\,Q\,[E]$, then $\Delta; P \vdash E \Leftarrow y{:}B.\,Q\,[E]$.*
4. *If $\Delta, R \vdash B \Leftarrow \mathsf{type}\,[B]$, then $\Delta \vdash B \Leftarrow \mathsf{type}\,[B]$.*
5. *If $\Delta, R \Longrightarrow Q$, then $\Delta \Longrightarrow Q$.*

Proof: By straightforward induction on the structure of the first given derivation in each case. ■

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 Compositionality (which we called "Preservation of History" in the previous papers), 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 $R \circ P$ and a new postcondition $R \circ Q$.

Lemma 11 (Properties of computations)
Suppose that $\Delta; P \vdash E \Leftarrow x{:}A.\,Q\,[E']$. Then:

1. Weakening consequent. *If $\Delta, x{:}A, i{:}\mathit{heap}, m{:}\mathit{heap}, Q\ i\ m \Longrightarrow R\ i\ m$, then $\Delta; P \vdash E \Leftarrow x{:}A.\,R\,[E']$.*
2. Strengthening precedent. *If $\Delta, i{:}\mathit{heap}, m{:}\mathit{heap}, R\ i\ m \Longrightarrow P\ i\ m$, then $\Delta; R \vdash E \Leftarrow x{:}A.\,Q\,[E']$.*
3. Composition.*If $\Delta \vdash R \Leftarrow \mathit{heap}{\rightarrow}\mathit{heap}{\rightarrow}\mathsf{prop}\,[R]$, then $\Delta; (R \circ P) \vdash E \Leftarrow x{:}A.\,(R \circ Q)\,[E']$.*

Proof: To prove weakening of consequent, from $\Delta; P \vdash E \Leftarrow x{:}A.\,Q\,[E']$ we know that there exists a predicate $S : {:}\mathsf{heap}{\rightarrow}\mathsf{heap}{\rightarrow}\mathsf{prop}$, such that $\Delta; P \vdash E \Rightarrow x{:}A.\,S$ where $\Delta, x{:}A, i{:}\mathsf{heap}, m{:}\mathsf{heap}, S\ i\ m \Longrightarrow Q\ i\ m$. Applying the rule of cut, we get $\Delta, x{:}A, i{:}\mathsf{heap}, m{:}\mathsf{heap}, S\ i\ m \Longrightarrow R\ i\ m$, and thus $\Delta; P \vdash E \Leftarrow x{:}A.\,R\,[E']$.

Strengthening precedent and composition are proved by induction on the structure of E. In both statements, the characteristic case is $E = \mathsf{let\ dia}\ y = K\ \mathsf{in}\ F$. In this case, from the typing of E we obtain: $\Delta \vdash K \Rightarrow \{R_1\}y{:}B\{R_2\}\,[N']$ where $\Delta, i{:}\mathsf{heap}, m{:}\mathsf{heap}, (P\ i\ m) \Longrightarrow (R_1 * \lambda m'.\,\top)\ m$, and $\Delta, y{:}B; P \circ (R_1 \gg R_2) \vdash F \Rightarrow x{:}A.\,S\,[F']$ where also $\Delta, x{:}A, i{:}\mathsf{heap}, m{:}\mathsf{heap}, \exists y{:}B.\,(S\ i\ m) \Longrightarrow (Q\ i\ m)$, 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. $\Delta, i{:}\mathsf{heap}, m{:}\mathsf{heap}, R\ i\ m \Longrightarrow (R_1 * \lambda m'.\,\top)(m)$, and
2. $\Delta, y{:}B; R \circ (R_1 \gg R_2) \vdash F \Rightarrow x{:}A.\,S'\,[F']$ for some $S'{:}\mathsf{heap}{\rightarrow}\mathsf{heap}{\rightarrow}\mathsf{prop}$
   such that $\Delta, x{:}A, i{:}\mathsf{heap}, m{:}\mathsf{heap}, \exists y{:}B.\,(S'\ i\ m) \Longrightarrow (Q\ i\ m)$.

The sequent (1) follows by the rule of cut, from the assumption $R\,i\,m \Longrightarrow P\,i\,m$ and the sequent $P\,i\,m \Longrightarrow (R_1 * \lambda m'.\,\top)(m)$ obtained from the typing of E. To derive (2), we first observe that $\Delta, y{:}B; P \circ (R_1 \gg R_2) \vdash F \Rightarrow x{:}A.\,S\,[F']$ implies $\Delta, y{:}B; P \circ (R_1 \gg R_2) \vdash F \Leftarrow x{:}A.\,S\,[F']$, by the inference rule $\mathsf{consequent}$, and using the initial sequent $S\ i\ m \Longrightarrow S\ i\ m$. It is also easy to show that the sequent $\Delta, i{:}\mathsf{heap}, m{:}\mathsf{heap}, (R \circ (R_1 \gg R_2))\ i\ m \Longrightarrow (P \circ (R_1 \gg R_2))\ i\ m$ is derivable, after first expanding the definition of the operator "$\circ$" on the left, and then on the right. Now, by induction hypothesis on F, we have $\Delta, y{:}B; R \circ (R_1 \gg R_2) \vdash F \Leftarrow x{:}A.\,S\,[F']$.

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

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

3. $\Delta, i{:}\mathsf{heap}, m{:}\mathsf{heap}, (R \circ P)\ i\ m \Longrightarrow (R_1 * \lambda m'.\,\top)(m)$, and

4. $\Delta, y{:}B; (R \circ P) \circ (R_1 \gg R_2) \vdash F \Rightarrow x{:}A.\, S'\,[F']$ where
   $\Delta, x{:}A, i{:}\mathsf{heap}, m{:}\mathsf{heap}, \exists y{:}B.\,(S'\ i\ m) \Longrightarrow (R \circ Q)\ i\ m$.

Sequent (3) follows by cut from the sequents $(R \circ P)\ i\ m \Longrightarrow \exists h{:}\mathsf{heap}.\, P\ h\ m$ and $\exists h{:}\mathsf{heap}.\, P\ h\ m \Longrightarrow (R_1 * \lambda m'.\,\top)(m)$. The first sequent is trivially obtained after expanding the definition of "$\circ$". To prove the second sequent, we eliminate the existential on the left to obtain the subgoal $P\ i\ m \Longrightarrow (R_1 * \lambda m'.\,\top)(m)$. But this subgoal is already given as a consequence of the typing of E. To derive (4), we apply the induction hypothesis on the typing derivation for F, to obtain $\Delta, y{:}B; R \circ (P \circ (R_1 \gg R_2)) \vdash F \Leftarrow x{:}A.\,(R \circ S)$. This gives us $\Delta, y{:}B; (R \circ P) \circ (R_1 \gg R_2) \vdash F \Leftarrow x{:}A.\,(R \circ S)$ by using strengthening of precedent and associativity of "$\circ$" (which is easy to show).

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

The other cases of Composition are proved in a similar way relying on the properties that $R \circ (\lambda i.\,(P\ i * X)) = \lambda i.\,((R \circ P)\ i * X)$ (in the case of alloc) and $R \circ (\lambda i.\,\lambda m.\,(P\ i\ m \wedge X\ m)) = \lambda i.\,\lambda m.\,((R \circ P)\ i\ m \wedge X\ m)$ (in the case of lookup). Both of these equations are easy to prove. ■

The substitution principle for canonical forms is now mutually recursive with the identity principle, because the definition of hereditary substitutions is mutual recursive with the definition of eta expansion.

Lemma 12 (Canonical identity and substitution principles)

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

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

   (a) 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

      i. if $[M/x]^k_A(K) = K'$ is an elim term, then $\Delta, \Delta_1' \vdash K' \Rightarrow B'\,[K']$

      ii. if $[M/x]^k_A(K) = N' :: C'$ is an intro term, then $C' = B'$ and $\Delta, \Delta_1' \vdash N' \Leftarrow B'\,[N']$.

   (b) 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 $\Delta, \Delta_1' \vdash [M/x]^m_A(N) \Leftarrow B'\,[[M/x]^m_A(N)]$.

   (c) If $\Delta, x{:}A, \Delta_1; P \vdash E \Leftarrow y{:}B.\, Q\,[E]$, and $y \notin \mathsf{FV}(M)$, and the predicates $P' = [M/x]^p_A(P)$ and $Q' = [M/x]^p_A(Q)$ and the type $B' = [M/x]^a_A(B)$ exist and are well-formed (i.e., $\Delta, \Delta_1' \vdash P' \Leftarrow \mathsf{heap}\to\mathsf{heap}\to\mathsf{prop}\,[P']$, $\Delta, \Delta_1' \vdash B' \Leftarrow \mathsf{type}\,[B']$ and $\Delta, \Delta_1', y{:}B' \vdash Q' \Leftarrow \mathsf{heap}\to\mathsf{heap}\to\mathsf{prop}\,[Q']$), then $\Delta, \Delta_1'; P' \vdash [M/x]^e_A(E) \Leftarrow y{:}B'.\, Q'\,[[M/x]^e_A(E)]$.

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

   (e) If $\Delta, x{:}A, \Delta_1 \Longrightarrow P$, and the assertion $P' = [M/x]^p_A(P)$ exists and is well-formed (i.e., $\Delta, \Delta_1' \vdash P' \Leftarrow \mathsf{prop}\,[P']$) then $\Delta, \Delta_1' \Longrightarrow P'$.

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

Proof: By nested induction, first on the metric $m(A)$, and then, for the variable substitution principle on the derivation of the first typing or sequent judgment in each case, and for the monadic substitution principle on the typing derivation for E.

For the identity principle (Statement 1), 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. $\Delta, i{:}\mathsf{heap}, m{:}\mathsf{heap}, (\mathsf{this}\ i\ m) \wedge (P * \lambda m'.\, \top)(m) \Longrightarrow (P * \lambda m'.\, \top)(m)$

2. $\Delta, x{:}B, i{:}\mathsf{heap}, m{:}\mathsf{heap}, \exists y{:}B.\, \exists h{:}\mathsf{heap}.\, \mathsf{id}_{\mathsf{heap}}(i, h) \wedge (P*\lambda m'.\, \top)(h) \wedge ((P \gg [y/x]Q)\ h\ m) \wedge \mathsf{id}_B(x, y) \Longrightarrow (P \gg Q)\ i\ m$

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 axioms for conjunction.

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 $\mathsf{id}_{\mathsf{heap}}(i, m) \wedge P$ actually implies $P \gg Q$. In the statement of the sequent we have used the previously stated syntactic convention to abbreviate $\mathsf{expand}_B(x)$ and $\mathsf{expand}_B(y)$ by x and y respectively, and similarly for the heaps i, h, m.

The sequent can easily be proved by noticing that $\mathsf{id}_B(x, y)$ and $(P \gg [y/x]Q)\ h\ m$ imply $(P \gg Q)\ h\ m$, and then $\mathsf{id}_{\mathsf{heap}}(i, h)$ leads to $(P \gg Q)\ i\ m$, as required.

In case $A = \Sigma x{:}B_1.\, B_2$, we need to show that

$$\begin{array}{rcl}
\mathsf{expand}_{A_1}(\mathsf{fst}\ K) & \Leftarrow & A_1\,[\mathsf{expand}_{A_1}(\mathsf{fst}\ K)] \\
\mathsf{expand}_{[\mathsf{expand}_{A_1}(\mathsf{fst}\ K)/x]^a_{A_1}(A_2)}(\mathsf{snd}\ K) & \Leftarrow & [\mathsf{expand}_{A_1}(\mathsf{fst}\ K)/x]^a_{A_1}(A_2)\,[\mathsf{expand}_{[\mathsf{expand}_{A_1}(\mathsf{fst}\ K)/x]^a_{A_1}(A_2)}(\mathsf{snd}\ K)]
\end{array}$$

The first judgment follows by induction on $m(A_1) < m(A)$. The second judgment is proved first by induction on $m(A_1) < m(A)$ in order to establish the existence of the type $[\mathsf{expand}_{A_1}(\mathsf{fst}\ K)/x]^a_{A_1}(A_2)$, and then by induction on $m([\mathsf{expand}_{A_1}(\mathsf{fst}\ K)/x]^a_{A_1}(A_2)) < m(A)$ in order to establish that the expansion is well typed. Here the inequality $m([\mathsf{expand}_{A_1}(\mathsf{fst}\ K)/x]^a_{A_1}(A_2)) < m(A)$ that justifies the induction step is obtained by the Termination theorem (Theorem 3, case 4).

For the Variable Substitution Principle (Statement 2), we also present several cases.

First, the case of (c) 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 $\Delta, x{:}A, \Delta_1, i{:}\mathsf{heap}, m{:}\mathsf{heap}, (P\ i\ m) \Longrightarrow (R_1 * \lambda m'.\, \top)(m)$, and $\Delta, x{:}A, \Delta_1, z{:}C; P \circ (R_1 \gg R_2) \vdash F \Leftarrow y{:}B.\, Q\,[F]$ and $\Delta, \Delta_1'; \lambda i.\, \lambda m.\, (\mathsf{this}\ i\ m) \wedge (R_1' * \lambda m'.\, \top)(m) \vdash E_1 \Leftarrow z{:}C'.\, R_1' \gg R_2'\,[E_1]$. By induction hypothesis on K, we know that $[M/x]K = diaE_1 :: \{R_1'\}z{:}C'\{R_2'\}$, and thus, $m(C') < m(\{R_1'\}z{:}C'\{R_2'\}) < m(A)$ by Theorem 3. And, of course, by definition $E' = \langle E_1/z\rangle_C(F')$.

From the typing of E_1, by Composition (Lemma 11), $\Delta, \Delta_1'; P' \circ (\lambda i.\, \lambda m.\, (\mathsf{this}\ i\ m) \wedge (R_1' * \lambda m'.\, \top)(m)) \vdash E_1 \Leftarrow z{:}C'.\, P' \circ (R_1' \gg R_2')\,[E_1]$. From the typing of F, by induction hypothesis, $\Delta, \Delta_1', z{:}C'; P' \circ (R_1' \gg R_2') \vdash F' \Leftarrow y{:}B'.\, Q'\,[F']$. By induction hypothesis on $m(C') < m(A)$ and from the above two judgments, by monadically substituting E_1 for z in F', we obtain $\Delta, \Delta_1'; P' \circ (\lambda i.\, \lambda m.\, (\mathsf{this}\ i\ m) \wedge (R_1' * \lambda m'.\, \top)(m)) \vdash E' \Leftarrow y{:}B'.\, Q'\,[E']$.

Finally, by induction hypothesis on the derivation of the sequent $P\ i\ m \Longrightarrow (R_1 * \lambda m'.\, \top)(m)$ we obtain $P'\ i\ m \Longrightarrow (R_1' * \lambda m'.\, \top)(m)$, and therefore also $P'\ i\ m \Longrightarrow (P' \circ (\lambda i.\, \lambda m.\, (\mathsf{this}\ i\ m) \wedge (R_1' * \lambda m'.\, \top)(m)))\ i\ m$. Now we can apply strengthening of the precedent (Lemma 11) to derive the required $\Delta, \Delta_1'; P' \vdash E' \Leftarrow y{:}B'.\, Q'\,[E']$.

Yet another interesting case is (b) when $N = \mathsf{eta}_K\ L$. Let us consider the subcase when $\mathsf{head}(K) = x$ and $\mathsf{head}(L) \neq x$.

In this case by typing, we know that $\Delta, x{:}A, \Delta_1 \vdash L \Rightarrow K\,[L]$ where $K = B$ and $\Delta, x{:}A, \Delta_1 \vdash K \Leftarrow \mathsf{type}\,[K]$. Because K is a term, we also know that $\Delta, x{:}A, \Delta_1 \vdash K \Leftarrow \mathsf{mono}\,[K]$.

By i.h. on K, we know that $[M/x]^k_A(K) = B' :: \mathsf{mono}$. By i.h. on L, we know that $\Delta, \Delta_1' \vdash L' \Rightarrow B'$.

By definition, $[M/x](\mathsf{eta}_K\ L) = \mathsf{expand}_{B'}(L')$. Now, by appealing to the identity principle as an induction hypothesis with $m(B') = m([M/x]K) < m(\mathsf{mono}) \leq m(A)$, we obtain $\mathsf{expand}_{B'}(L') \Leftarrow B'\,[\mathsf{expand}_{B'}(L')]$, which is precisely what we needed to show.

Another interesting case is (a) when $K = \mathsf{snd}\ L$.

By typing derivation, we know that $\Delta, x{:}A, \Delta_1 \vdash L \Rightarrow \Sigma y{:}B_1.\, B_2\,[L]$ and $B = [\mathsf{expand}_{B_1}(\mathsf{fst}\ (L))/y]_{B_1}(B_2)$.

Now, we have two subcases: $\mathsf{head}(K) = \mathsf{head}(L) = x$ and $\mathsf{head}(K) = \mathsf{head}(L) \neq x$.

- subcase $\mathsf{head}(K, L) = x$.

In this case, we know that $[M/x]L = N' :: \Sigma y{:}B_1'.\, B_2'$, and by i.h. on L:
$\Delta, \Delta_1' \vdash N' \Leftarrow \Sigma y{:}B_1'.\, B_2'\,[N']$.

By inversion, it must be $N' = (N_1', N_2')$, where $\Delta, \Delta_1' \vdash N_2' \Leftarrow [N_1'/y]_{B_1'}(B_2')\,[N_2']$

Now, by definition $[M/x](\mathsf{snd}\ K) = N_2'$, so it only remains to be proved that $[N_1'/y]_{B_1'}(B_2') = [M/x]B = [M/x][\mathsf{expand}_{B_1}(\mathsf{fst}\ L)/y]_{B_1}(B_2)$.

To prove the last equation, we first notice that

$$\begin{aligned} [M/x]_A(\mathsf{expand}_{B_1}(\mathsf{fst}\ L)) &= [[M/x]_A(\mathsf{fst}\ L)/z]_{[M/x](B_1)}(\mathsf{expand}_{[M/x](B_1)} z) \\ &= [N_1'/z]_{[M/x](B_1)}(\mathsf{expand}_{[M/x](B_1)} z) \\ &= N_1' \quad \text{by Lemma 9.5 on properties of variable expansion} \end{aligned}$$

Now by composition of hereditary substitutions, we can push $[M/x]$ inside in the expression $[M/x][\mathsf{expand}_{B_1}(\mathsf{fst}\ L)/y]_{B_1}(B_2)$ to obtain the equivalent: $[N_1'/y]_{B_1'}(B_2')$. But this is precisely what we wanted to show.

- subcase $\mathsf{head}(K) = \mathsf{head}(L) \neq x$ is similar, but we end up having to show that

$$[M/x]_A(\mathsf{expand}_{B_1}(\mathsf{fst}\ L)) = \mathsf{expand}_{[M/x]_A(B_1)}(\mathsf{fst}\ [M/x]L).$$

But this property follows from the composition of hereditary substitutions.

■

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 12.

Lemma 13 (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 $\Delta \vdash A' \Leftarrow \mathsf{type}\,[A']$.*

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

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

In this case, by the typing derivation, we know that $\Delta \vdash K \Rightarrow B\,[M']$ and $A = B$ and $N' = \mathsf{exp}_A(M')$. Now, if M' is an introduction term, then $N' = M'$ and the result immediately follows by induction hypothesis 2. On the other hand, if $M' = K'$ is an elimination term, then $N' = \mathsf{expand}_A(K')$ and by induction hypothesis 1, $\Delta \vdash K' \Rightarrow A\,[K']$, and then by the identity principle (Lemma 12.1), $\Delta \vdash \mathsf{expand}_A(K') \Leftarrow A\,[\mathsf{expand}_A(K')]$. ■

Finally, we can state the substitution principles for the general forms.

Lemma 14 (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 $\Delta; P \vdash E \Leftarrow x{:}A'.\,Q\,[E']$ 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. The proofs are largely similar to the proofs of the canonical substitution principles.

One distinction from the canonical forms analogue is that the case $N = \mathsf{eta}_\alpha\ K$ does not arise here, because general forms are assumed not to contain the constructor eta. ■

7 Operational semantics

In this section we consider the operational semantics of the canonical fragment of HTT. We focus on the canonical fragment, because canonical forms carry the real meaning of terms in HTT, while the general forms can be viewed merely as a convenience for describing programs in a more concise way.

When we want to evaluate a general form, we first convert it into its canonical equivalent, and then evaluate that canonical equivalent according to the formalism presented in this section.

If one really wants to define an operational semantics directly on general forms, that is certainly possible, and we refer the reader to our previous work on the first-order version of HTT [32, 33], where we presented a call-by-value operational semantics for the general forms. However, general forms introduce a lot of clutter, which may obscure the nature of the system.

That is why in this section we focus on canonical forms. This means that all the pure terms that may be encounter during the evaluation have already been fully reduced, so that we only need to consider the evaluation of effectful computations.

The evaluation judgment for the canonical operational semantics has the form $\chi_1 \triangleright E_1 \longrightarrow \chi_2 \triangleright E_2$. Here, of course, E_1 is a (canonical) computation, χ_1 is the heap in which the evaluation of E_1 starts, E_2 is the computation obtained after one step, and χ_2 is the new heap obtained after one step.

The heaps χ_1 and χ_2 belong to a new syntactic category of *run-time heaps* defined as follows.

$$\textit{Run-time heaps} \quad \chi \quad ::= \quad \cdot \mid \chi, n \mapsto_\tau M$$

Here we assume that n is a numeral, τ is a canonical small type, and $M{:}\tau$ is also canonical. Furthermore, each numeral n appears at most once in the run-time heap.

Run-time heaps are – as their name says – a run-time concept, unlike heaps from Section 2 which are expressions used for reasoning in the assertion logic. That the two notions actually correspond to each other is the statement of the *Heap Soundness* lemma, given later in this section, which shows that HTT correctly reasons about its objects of interest (in this case, the run-time heaps).

Clearly, each run-time heap can be seen as a partial function mapping a natural number n into a pair (τ, M). Thus, we write $\chi(n) = (\tau, M)$ if the heap χ assigns the value $M{:}\tau$ to the location n, and we write $\chi[n \mapsto_\tau M]$ for a heap obtained by updating the location n so that it points to the value $M{:}\tau$.

There is an obvious embedding $(\chi)^+$ of run-time heaps into HTT heaps given as

$$\begin{array}{rcl} (\cdot)^+ & = & \mathsf{empty} \\ (\chi, n \mapsto_\tau N)^+ & = & \mathsf{upd}\, \chi^+\, n\, \tau\, N \end{array}$$

Moreover, this embedding respects the operations $\chi(n)$ and $\chi[n \mapsto_\tau M]$ which become respectively seleq and upd. Thus, we will abuse the notation and freely use run-time heaps as if they were HTT heaps when we need them in the HTT assertions. Whenever χ appears in an assertion, it is assumed implicitly converted into an HTT term by the embedding χ^+.

We will use the name *abstract machine* for the pair $\chi \triangleright E$, and we use μ and variants to range over abstract machines. The rules of the evaluation judgment $\mu_1 \longrightarrow \mu_2$ are given below.

$$\frac{\chi(n) = (\tau, M)}{\chi \triangleright x = !_\tau\, n; E \longrightarrow \chi \triangleright [M/x]^e_\tau(E)}$$

$$\frac{\chi(n) = (\tau, M)}{\chi \triangleright n :=_\sigma N; E \longrightarrow \chi[n \mapsto_\sigma N] \triangleright E}$$

$$\chi \triangleright x = \mathsf{if}_A\ \mathsf{true\ then}\ E_1\ \mathsf{else}\ E_2; E \longrightarrow \chi \triangleright \langle E_1/x\rangle_A(E)$$

$$\chi \triangleright x = \mathsf{if}_A\ \mathsf{false\ then}\ E_1\ \mathsf{else}\ E_2; E \longrightarrow \chi \triangleright \langle E_2/x\rangle_A(E)$$

$$\chi \triangleright x = \mathsf{case}_A\ \mathsf{z\ of}\ z \Rightarrow E_1\ \mathsf{or\ s}\ x \Rightarrow E_2; E \longrightarrow \chi \triangleright \langle E_1/x\rangle_A(E)$$

$$\chi \triangleright x = \mathsf{case}_A\ \mathsf{s}\ n\ \mathsf{of}\ z \Rightarrow E_1\ \mathsf{or\ s}\ y \Rightarrow E_2; E \longrightarrow \chi \triangleright \langle [n/y]^e_{\mathsf{nat}}(E_2)/x\rangle_A(E)$$

$$\frac{C = \{R_1\}z{:}B\{R_2\} \qquad N = \lambda w.\, \mathsf{dia}\ (\mathsf{fix}\ f(y{:}A){:}C = \mathsf{dia}\, F\ \mathsf{in\ eval}\ f\, w)}{\chi \triangleright x = \mathsf{fix}\ f(y{:}A){:}C = \mathsf{dia}\, F\ \mathsf{in\ eval}\ f\, M; E \longrightarrow \chi \triangleright \langle [M/y]^e_A [N/f]^e_{\Pi y{:}A.\, C}(F)/x\rangle_B(E)}$$

Before we can state the theorems, we need a judgment for typing of abstract machines. Given $\mu = \chi \triangleright E$, the typing judgment $\vdash \mu \Leftarrow x{:}A.\, Q$ holds iff $\vdash \chi \Leftarrow \mathsf{heap}\,[\chi]$ and $\cdot; \lambda i.\, \lambda m.\, \mathsf{id}_{\mathsf{heap}}(m, \chi) \vdash E \Leftarrow x{:}A.\, Q\,[E]$. In this section we only work with canonical forms, so we will omit the canonical expressions returned as the output of the typing judgments, since they are guaranteed to be equal to the input expressions.

Theorem 15 (Preservation)
If $\mu_0 \longrightarrow \mu_1$ *and* $\vdash \mu_0 \Leftarrow x{:}A.\, Q$*, then* $\vdash \mu_1 \Leftarrow x{:}A.\, Q$.

Proof: By case analysis on the evaluation judgment.

case $\mu_0 = \chi \triangleright y = !_\tau\, n; E$. Then $\chi(n) = (\tau, M)$ and $\mu_1 = \chi \triangleright [M/x]^e_\tau(E)$. In this case, by the typing of μ_0 we know that

1. $\vdash M \Leftarrow \tau$
2. $y{:}\tau; \lambda i.\, \lambda m.\, \mathsf{id}_{\mathsf{heap}}(m, \chi) \wedge (n \hookrightarrow_\tau y)(m) \vdash E \Leftarrow x{:}A.\, Q$

We need to show that $\vdash \mu_1 \Leftarrow x{:}A.\, Q$, i.e., that $\cdot; \lambda i.\, \lambda m.\, \mathsf{id}_{\mathsf{heap}}(m, \chi) \vdash [M/x]^e_\tau(E) \Leftarrow x{:}A.\, Q$. But this follows from (2) by the substitution principle, and then by strengthening precedent. We just need to show that $\mathsf{id}_{\mathsf{heap}}(m, \chi)$ implies $\mathsf{id}_{\mathsf{heap}}(m, \chi) \wedge (n \hookrightarrow_\tau M)(\chi)$. The last implication follows by the substitution of equals for equals, and the assumption $\chi(n) = (\tau, M)$, which obviously implies $(n \hookrightarrow_\tau M)(\chi)$, which can easily be shown by the induction on the size of χ.

case $\mu_0 = \chi \triangleright y :=_\sigma N; E$. Then $\chi(n) = (\tau, M)$, and $\mu_1 = \chi[n \mapsto_\sigma N] \triangleright E$. In this case, by typing of μ_0, we have

1. $\cdot; (\lambda i. \lambda m. \mathsf{id}_{\mathsf{heap}}(m, \chi)) \circ (n \mapsto_\tau - \multimap n \mapsto_\sigma N) \vdash E \Leftarrow x{:}A. Q$

We need to show that $\vdash \mu_1 \Leftarrow x{:}A. Q$, i.e., that $\cdot; \lambda i. \lambda m. \mathsf{id}_{\mathsf{heap}}(m, \chi[n \mapsto_\sigma N]) \vdash E \Leftarrow x{:}A. Q$. But this again immediately follows from (1) by strengthening of the precedent, because $\mathsf{id}_{\mathsf{heap}}(m, \chi[n \mapsto_\sigma N]) \supset \exists h{:}\mathsf{heap}. \mathsf{id}_{\mathsf{heap}}(h, \chi) \wedge (n \mapsto_\tau - \multimap n \mapsto_\sigma N)\ h\ m$ (which is unrolling of the composition above). Indeed, we know by assumption that $\chi(n) = (\tau, M)$, so there is a subheap of χ containing only n of which $n \mapsto_\tau M$ and hence $n \mapsto_\tau -$ holds. After the update, we have simply replaced that subheap with a new one of which $n \mapsto_\sigma N$ holds, while keeping everything else intact. Thus, the heaps $h = \chi$ and $m = \chi[n \mapsto_\sigma N]$ validate the proposition $(n \mapsto_\tau - \multimap n \mapsto_\sigma N)\ h\ m$.

case $\mu_0 = \chi \triangleright y = \mathsf{fix}\, f(z{:}B){:}C = \mathsf{dia}\ F\ \mathsf{in}\ \mathsf{eval}\, f\, M; E$. Then $\mu_1 = \chi \triangleright \langle [M/z][N/f](F)/y\rangle(E)$, where N is as described in the typing rule. In this case, by typing of μ_0, we know

1. $C = \{R_1\} y{:}D \{R_2\}$
2. $z{:}B; f{:}\Pi z{:}B. C; R_1 * \top \vdash F \Leftarrow y{:}D. R_1 \gg R_2$ (we abuse the notation slightly here and omit the excessive lambda abstraction of i and m in the precondition)
3. $y{:}[M/z]D; (\lambda i. \lambda m. \mathsf{id}_{\mathsf{heap}}(m, \chi)) \circ [M/z](R_1 \gg R_2) \vdash E \Leftarrow x{:}A. Q$.
4. $\Longrightarrow \mathsf{id}_{\mathsf{heap}}(m, \chi) \supset [M/z](R_1 * \top)$

From (2) we can conclude that $N \Leftarrow \Pi z{:}B. C$ (where N is as defined in the typing rule), and then by the substitution principle $\cdot; [M/z](R_1 * \top) \vdash [M/z][N/f]F \Leftarrow y{:}[M/z]D. [M/z](R_1 \gg R_2)$.

By the compositionality of computations, if we take $R = \lambda i. \lambda m. \mathsf{id}_{\mathsf{heap}}(m, \chi)$, we get $\cdot; R \circ [M/z](R_1 * \top) \vdash [M/z][N/f](F) \Leftarrow y{:}[M/z]D. R \circ [M/z](R_1 \gg R_2)$. From here, and the monadic substitution principle over (3), we get $\cdot; R \circ [M/z](R_1 * \top) \vdash \langle [M/z][N/f](F)/y\rangle(E) \Leftarrow x{:}A. Q$.

But, by using (4), we know that $\mathsf{id}_{\mathsf{heap}}(m, \chi) \supset (R \circ [M/z](R_1 * \top))\ i\ m$, and hence, by strengthening precedent: $\cdot; \mathsf{id}_{\mathsf{heap}}(m, \chi) \vdash \langle [M/z][N/f](F)/y\rangle(E) \Leftarrow x{:}A. Q$. But this is precisely the required $\vdash \mu_1 \Leftarrow x{:}A. Q$.

The rest of the cases involving the conditionals are proved in a similar, straightforward fashion. ■

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

Notice that in the evaluation rules we must occasionally 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.

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 16 (Heap soundness)
The assertion logic of HTT is heap sound iff for every run-time heap χ and numeral n, the existence of a derivation of $m{:}\mathit{heap} \vdash \mathit{id}_{\mathit{heap}}(m, \chi) \supset (n \hookrightarrow_\tau -)(m)$ implies that $\chi(n) = (\tau, M)$ for some canonical expression M such that $\vdash M \Leftarrow \tau$.

Notice that the opposite of Heap soundness, i.e. that if $\chi(n) = (\tau, M)$ implies $\mathsf{id}_{\mathsf{heap}}(m, \chi) \supset (n \hookrightarrow_\tau -)(m)$ is easy to prove (say by induction on the size of χ), and we have already used this fact in the proof of the Preservation theorem.

However, the Heap Soundness itself is much harder. The definition of heap soundness correspond to the side conditions that need to be derived in the typing rules for the primitive commands of lookup and update. Heap soundness essentially shows that the assertion logic soundly reasons about run-time heaps, so that facts established in the assertion logic will be true during evaluation. If the assertion logic proves that $n \hookrightarrow_\tau -$, then the evaluation will be able to associate a term M with this location, which is needed, for example, in the evaluation rule for lookup.

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

Theorem 17 (Progress)
Suppose that the assertion logic of HTT is heap sound. If $\vdash \chi_0 \triangleright E_0 \Leftarrow x{:}A.\, Q$, *then either* $E_0 = N$, *or* $\chi_0 \triangleright E_0 \longrightarrow \chi_1 \triangleright E_1$.

Proof: The proof is by straightforward case analysis. The only interesting cases are when the first command in E_0 is a lookup or an update. One of these command may fail to make a step if the premise of its corresponding evaluation rule is not satisfied. However, by Heap Soundness, we immediately conclude that the premises in these two rules must be satisfied if $\chi_0 \triangleright E_0$ is well-typed (as is assumed). No other evaluation rules have any premises (the premises in the rule for fix are merely notational abbreviations, not real conditions), so they are guaranteed to make a step. ■

8 Heap Soundness

In this section we show that the assertion logic is heap sound. We do so by means of a simple, and somewhat crude, set-theoretic semantics of HTT. Our set-theoretic model depends, as the denotational model in our previous work [33] did, on the observation that the assertion logic does not include axioms for computations; reasoning about computations is formalized via the typing rules and soundness of those is proved above via progress and preservation assuming soundness of the assertion logic (heap soundness).

Thus in our set-theoretic model, we chose to simply interpret the type $\{P\}x{:}A\{Q\}$ as a one-element set, emphasizing that the assertion logic cannot distinguish between different computations. Given this basic decision we are really just left with interpreting a type theory similar to the Extended Calculus of Constructions with a (assertion) logic on top. The type theory has two universes (mono and other types) and is similar to the Extended Calculus of Construction (ECC), except that the mono universe is not impredicative. Hence we can use a simplified version of Luo's model of ECC [23, Ch. 7] for the types. Thus our model is really fairly standard and hence we only include a sketch of it here.

As in [23, Ch. 7] our model takes place in ZFC set theory with infinite inaccessible cardinals κ_0, κ_1, ... (see *loc. cit.* for details). The universe mono is the set of all sets of cardinality smaller than κ_0. The type nat is interpreted as the set of natural numbers, bool as the set of booleans, $\Pi x{:}A.\, B$ as dependent product in sets, $\Sigma x{:}A.\, B$ as dependent sum in sets. Predicates P on a type are interpreted as subsets in the classical way. Finally, of course, $\{x{:}A.\, P\}$ is just interpreted as the set-theoretic subset given by the interpretation of P.

Thus we clearly get a sound model of classical higher-order logic and the assertion logic is clearly heap sound.

9 Conclusions, related and future work

In this paper we present an extension our Hoare Type Theory (HTT), with higher-order predicates, and allow quantification over abstract predicates at the level of terms, types and assertions. This significantly increased the power of the system to encompass definition of inductive predicates, abstraction of program invariants, and even first-class modules that can contain not only types and terms, but also specifications of properties that the types and the terms are required to satisfy.

Technically, the main additions are dependent sums, subset types and the types prop and mono. Elements of the type prop are assertions, and thus quantifying over them provides the power of higher-order logic. Similarly, elements of mono are (small) types, and we can abstract over and compute with them. The later provides HTT with enough power to express some important programming features from mainstream module systems, like abstract types, structures and signatures and functions over them (i.e. functors).

From the point of view of application, we have shown that abstracting higher-order predicates at the level of terms, leads to a type system that can express ownership, sharing and invariants on local state of higher-order functions and abstract datatypes.

Recently, several variants of Hoare logics for higher-order languages have appeared. Berger et al. [4, 16] define a logic for PCF with references, and Krishnaswami [19] defines a logic for core ML extended with a monad. Also Birkedal et al. [6] defines a Higher-Order Separation Logic for reasoning about ADTs in first-order programs. Neither of these logics considers strong updates, pointer arithmetic or source language features like type polymorphism, modules, or dependent types. While Hoare Logics (for first- or higher-order languages) can certainly adequately represent a large class of program properties, we believe that a type theoretic formulation like HTT has certain advantages. In particular, a Hoare logic cannot really interact with the type system of the underlying language. It is not possible to abstract over specifications in the source programs, aggregate the logical invariants of the data structures with the data itself, compute with such invariants, or nest the specifications into larger specifications or types. These features are essential ingredients for data abstraction and information hiding, and, in fact, a number of researchers have tried to address this shortcoming and add such abstraction mechanisms to Hoare-like logics, usually without formal semantical considerations. Examples include bug-finding and verification tools and languages like Spec# [2], SPLint [11], and Cyclone [17], and ESC/Java [10]. As an illustration, in ESC/Java, the implementation of an ADT can be related to the interface by so called *abstraction dependencies* [21], which roughly corresponds to the invariants that we pair up with code in HTT.

The work on dependently typed systems with stateful features, has mostly focused on how to appropriately restrict the language of types so that effects do not pollute the types. If types only depend on pure terms, it becomes possible to use logical reasoning about them. Such systems have mostly employed singleton types to enforce purity. Examples include Dependent ML by Xi and Pfenning [44], Applied Type Systems by Xi [43] and Zhu and Xi [45], and a type system for certified binaries by Shao et al. [40]. HTT differs from all these approaches, because types are allowed to depend on monadically encapsulated effectful computations. We also mention the theory of type refinements by Mandelbaum et al. [25], which reasons about programs with effects, by employing a restricted fragment of linear logic. The restriction on the logic limits the class of properties that can be described, but is undertaken to preserve decidability of type checking.

Finally, we mention that HTT may be obtained by adding effects and the Hoare type to the Extended Calculus of Constructions (ECC) [23]. There are some differences between ECC and the pure fragment of HTT, but they are largely inessential. For example, HTT uses classical assertion logic, whereas ECC is intuitionistic, but consistent with classical extensions. The later has been demonstrated in Coq [26] which implements and subsumes ECC. A related property is that ECC interprets each proposition as a type of its proofs, while in the current paper, there is no explicit syntax for proofs; proofs are discovered by invoking, as an oracle, the judgment that formalizes the assertion logic. Another difference is that HTT contains only two type universes (small and large types), while ECC is more general, and contains the whole infinite tower. However, we do not expect that the proof terms, intuitionism, or the full universe tower would be particularly difficult to combine with HTT.

This opens a question if HTT can perhaps be shallowly embedded into ECC or Coq, so that HTT functions are represented by Coq functions, and a Hoare type is represented as a function space from heaps to heaps. We believe that this embedding is not possible, but do plan to investigate the issue further. The main problem is that Hoare types contain a certain impredicativity in their definition. For example, the type $\{P\}x{:}\tau\{Q\}$ is small (if τ is small), even though it classifies computations on heaps. But heap is a large type because heaps can contain pointers to any small type, including small Hoare types. The definition of Hoare types (and Hoare types only) thus exhibits a certain impredicativity which is essential for representing higher-order store. While Coq can soundly support an impredicative type universe Set, it requires that the type of propositions does not belong to this universe. In such a setting, quantification over local state, as we have done it in Section 3, will produce results that belong to a strictly larger universe, and thus cannot be stored in the heap. If only first-order store is considered, than Coq can certainly support Hoare-like specifications. An example can be found in the work of Filliâtre [12], who develops a Hoare-like logic for a language without pointer aliasing, where assertions can be drawn from Coq (or, for that matter, from several other logical frameworks, like PVS, Isabelle/HOL, HOL Light, etc).

The described impredicativity and disparity in size does not lead to unsoundness in HTT because computations are encapsulated within the monad. The monad prevents the unrolling of effectful computations during normalization, but we pay for this by not having as rich an equational reasoning over computations as we would over usual heap functions. Currently, the only equations over computations that HTT supports are the generic monadic laws [29, 30, 18, 41], and the beta and eta rules for the pure fragment. In the future, we intend to investigate which additional equations over stateful computations can soundly be added to HTT.

References

[1] A. Banerjee and D. A. Naumann. Ownership confinement ensures representation independence in object-oriented programs. *Journal of the ACM*, 52(6):894–960, November 2005.

[2] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In *CASSIS 2004*, Lecture Notes in Computer Science. Springer, 2004.

[3] N. Benton. Abstracting Allocation: The New new Thing. In *International Workshop on Computer Science Logic, CSL'06*, pages ??–??, 2006.

[4] 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.

[5] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions.* Texts in Theoretical Computer Science. Springer Verlag, 2004.

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

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

[8] T. Coquand and C. Paulin-Mohring. Inductively defined types. In P. Martin-Löf and G. Mints, editors, *Proceedings of Colog'88*, volume 417 of *Lecture Notes in Computer Science*. Springer-Verlag, 1990.

[9] R. DeLine and M. Fahndrich. Enforcing high-level protocols in low-level software. In *Conference on Programming Language Design and Implementation, PLDI'01*, pages 59–69, 2001.

[10] D. L. Detlefs, K. R. M. Leino, G. Nelson, and J. B. Saxe. Extended static checking. Compaq Systems Research Center, Research Report 159, December 1998.

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

[12] J.-C. Filliâtre. Verification of non-functional programs using interpretations in type theory. *Journal of Functional Programming*, 13(4):709–745, July 2003.

[13] I. Greif and A. Meyer. Specifying programming language semantics: a tutorial and critique of a paper by Hoare and Lauer. In *Symposium on Principles of Programming Languages, POPL'79*, pages 180–189, New York, NY, USA, 1979. ACM Press.

[14] R. Harper, J. C. Mitchell, and E. Moggi. Higher-order modules and the phase distinction. In *Symposium on Principles of Programming Languages, POPL'90*, pages 341–354, San Francisco, California, January 1990.

[15] J. Harrison. Inductive definitions: automation and application. In *Higher Order Logic Theorem Proving and Its Applications*, volume 971 of *Lecture Notes in Computer Science*, pages 200–213. Springer, 1995.

[16] 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.

[17] 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.

[18] 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.

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

[20] N. Krishnaswami and J. Aldrich. Permission-based ownership: encapsulating state in higher-order typed languages. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 96–106, New York, NY, USA, 2005. ACM Press.

[21] K. R. M. Leino and G. Nelson. Data abstraction and information hiding. *ACM Transactions on Programming Languages and Systems*, 24(5):491–553, 2002.

[22] 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.

[23] Z. Luo. *An Extended Calculus of Constructions.* PhD thesis, University of Edinburgh, 1990.

[24] D. MacQueen. Using dependent types to express modular structure. In *Symposium on Principles of Programming Languages, POPL'86*, pages 277–286, St. Petersburg Beach, Florida, 1986.

[25] 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.

[26] The Coq development team. *The Coq proof assistant reference manual.* LogiCal Project, 2004. Version 8.0.

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

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

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

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

[31] G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):527–568, 1999.

[32] A. Nanevski, G. Morrisett, and L. Birkedal. Polymorphism and separation in Hoare Type Theory. In *International Conference on Functional Programming, ICFP'06*, pages ??–??, Portland, Oregon, 2006.

[33] A. Nanevski, G. Morrisett, and L. Birkedal. Polymorphism and Separation in Hoare Type Theory. Technical Report TR-10-06, Harvard University, April 2006. Available at `http://www.eecs.harvard.edu/~aleks/papers/hoarelogic/httsep.pdf`.

[34] B. Nordström, K. Petersson, and J. M. Smith. *Programming in Martin-Löf Type Theory.* Oxford University Press, 1990.

[35] 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.

[36] 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.

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

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

[39] 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.

[40] 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.

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

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

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

[44] H. Xi and F. Pfenning. Dependent types in practical programming. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 214–227, San Antonio, January 1999.

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