Polymorphism and Separation in Hoare Type Theory

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

Lars Birkedal
IT University of Copenhagen
birkedal@itu.dk

Abstract

In previous work, we proposed a *Hoare Type Theory* (HTT) which combines effectful higher-order functions, dependent types and Hoare Logic specifications into a unified framework. However, the framework did not support polymorphism, and failed to provide a modular treatment of state in specifications. In this paper, we address these shortcomings by showing that the addition of polymorphism alone is sufficient for capturing modular state specifications in the style of Separation Logic. Furthermore, we argue that polymorphism is an essential ingredient of the extension, as the treatment of higher-order functions requires operations not encodable via the spatial connectives of Separation Logic.

Categories and Subject Descriptors F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs

General Terms Languages, Verification

Keywords Type Theory, Hoare Logic, Separation Logic

1. Introduction

The static type systems of today's programming languages, such as Java and Haskell, provide a degree of lightweight specification and verification that has proven remarkably effective at eliminating a class of coding errors. Furthermore, these type systems have scaled to cover and integrate with necessary linguistic features such as higher-order functions, objects, and imperative references and arrays.

Nevertheless, there is a range of errors, such as array-index-out-of-bounds and division-by-zero, which are not caught by today's type systems. And of course, there are higher-level correctness issues, such as invariants or protocols on mutable data structures, that fall well outside the range where types are effective.

An alternative approach to address these issues is to utilize a form of *dependent* types in conjunction with refinements (i.e., a type theory) to provide precise specifications of these requirements. Dependent types work well with higher-order features and are convenient for capturing relations on functional data structures, but do not work so well in the presence of side-effects, such as state updates and non-termination. Yet another approach is to consider some form of *program logic*, such as Hoare's original logic [12] or the more recent forms of Separation Logic [30, 36, 31], which are tuned for specifying and reasoning about imperative programs. However, these logics do not integrate *into* the type system. Rather, specifications, such as invariants on data structures or refinements on types, must be separately specified as pre- and postconditions on expressions that manipulate these data. In turn, this makes it difficult to scale the logics so that they integrate well with linguistic abstraction mechanisms such as higher-order functions, polymorphism, and modules.

In previous work [26], we demonstrated a new approach that smoothly integrates dependent types and a Hoare-style logic for a language with higher-order functions and imperative commands (i.e., core, monomorphic ML.) The key mechanism is a distinguished type constructor of Hoare (partial) triples $\{P\}x{:}A\{Q\}$, which serves to simultaneously isolate and describe the effects of imperative commands. Intuitively, such a type can be ascribed to a stateful computation if when executed in a heap satisfying the precondition P, the computation diverges or results in a heap satisfying the postcondition Q and returns a value of type A. The monadic isolation of effects is crucial for ensuring the soundness of the dependent types, and makes it possible to safely abstract over refined computations within terms, types, and assertions.

As with any sufficiently rich specification system, checking that HTT programs respect their types is generally undecidable. However, type-checking in HTT is carefully designed to split into two independent phases: The first performs a combination of basic type-checking and verification-condition generation, both of which are decidable. The second phase must then show the validity of the generated verification-conditions. These conditions can either be ignored, fed to an automated theorem prover, or even discharged by hand. This makes it possible to provide various levels of correctness assurance, and to gracefully scale the complexity of verification.

We believe that the HTT approach enjoys many of the benefits and few of the drawbacks of the alternatives mentioned above. In particular, we believe HTT is the right foundational framework for modeling emerging tools, such as ESC/Java [10, 19], SPLint [11], Spec# [2], and Cyclone [16] that provide support for extended static checking of programs.

Nevertheless, if we are to model these rich languages, the current formulation of HTT falls short in several ways. First, the language of HTT does not support polymorphism, which is necessary for Java, ML or Cyclone. Second, the approach to specifying program heaps—which in HTT is based on functional arrays of Cartwright and Oppen [7] and McCarthy [23]—is itself not modular. Pre- and postconditions in HTT describe the whole heap, rather than just the heap fragment that any particular program requires. Furthermore, the postconditions must explicitly describe how the heap in which the program terminates differs from the heap in which it started. Keeping track of both heaps in the postcondition is cumbersome as it requires careful tracking of location inequalities (i.e., lack of aliasing.) It is much better to simply assert the properties of the ending heap, and automatically assume that all unspecified disjoint heap portions remain invariant throughout the

ICFP'06 September 16–21, 2006, Portland, Oregon, USA.

computation. This is known as the "small footprint" approach to specification, and has been advocated recently by the work on Separation Logic.

In this paper, we extend HTT with type polymorphism (including abstraction over Hoare triples) and small footprints. It is interesting that these two additions significantly overlap. At first, we considered simply replacing the functional arrays with the ideas from Separation Logic, but then we realized that in the presence of polymorphism, functional arrays could already define the separation connectives of spatial conjunction and implication, that are needed to describe heap disjointness [30].

Not only that, but in order to accommodate higher-order functions, we needed additional operators that are not expressible using the separation connectives, but that are definable in the presence of polymorphism. Thus, functional arrays with polymorphism are utilized in an essential way to obtain the small footprints.

An important example that becomes possible in HTT, but is formally not admitted in Separation Logic, is naming and explicitly manipulating individual fragments of the heap. We contend that it is useful to be able to do so directly. In particular, it alleviates the need for an additional representation of heaps in assertions as was used in the verification of Cheney's garbage collection algorithm in Separation Logic by Birkedal et al. [5]. An additional feature admitted by polymorphism is that HTT can support strong updates, whereby a location can point to values of different types in the course of the execution.

Most of this paper presents and discusses the typing rules of the extended HTT, including the important meta-theoretic properties of the system. We also sketch a call-by-value operational semantics for the language, and a proof that the type system is sound with respect to this semantics. The proof depends on the soundness of the assertion logic, which we establish using denotational methods. The full technical development, including the proofs, is available in the accompanying report [27].

2. Syntax and overview

A crucial operation in any type system is comparing types for equality. In the case of dependent types, which we use in HTT to express partial correctness, types can contain terms, so type equality must compare terms as well, which is an undecidable problem in any Turing complete language (in fact, it is not even recursively enumerable). It is therefore crucial for HTT that we select equations on terms that strike the balance between the preciseness and decidability of the equality relation. In this choice, we are guided by the decision to separate typechecking from proving of program specifications. We introduce two different notions: *definitional equality*, which is coarse but decidable, and is employed during typechecking, and *propositional equality*, which is fine but undecidable and is used only in proving. The split into definitional and propositional equalities is a customary way to organize equational reasoning in type theories [13].

Almost all of HTT design is geared towards facilitating a formulation of a decidable definitional equality (propositional equality can be arbitrarily complex, so it does not require as much attention). For example, we split the HTT programs into two fragments: pure and impure – precisely in order to separate the concerns about equality. The pure fragment consists of higher-order functions, and constructs for type polymorphism. It admits the usual term equations of beta reduction and eta expansion. We do not include conditionals into the pure fragment because they do not allow an easy use of eta expansion. The impure fragment contains the constructs usually found in first-order imperative languages: allocation, lookup, strong update, deallocation of memory, conditionals and loops (in HTT formulated as recursion). All of these constructs admit reasoning in the style of Hoare Logic by pre- and postconditions, so we use the Hoare type $\{P\}x{:}A\{Q\}$ to classify the impure programs.

The split between pure and impure fragments is a familiar one in functional programming. For example, it is the driving idea behind the programming language Haskell [33], which uses monads [24, 17, 40], to classify impure code. It should therefore not come as a surprise that the Hoare type in HTT is a monad, and that we admit the usual monadic laws [34, 25] for reasoning about the impure code.

However, it may be interesting that HTT monads take a slightly bigger role than to simply serve as type markers for effects. The HTT monadic judgments actually formalize the process of generating the verification condition for an effectful computation by calculating strongest postconditions. If the verification condition is provable, then the computation matches its specification [29]. The verification condition is computed during typechecking, but it can be proved separately, so that the complexity and undecidability of proving does not have any bearing on the typechecker.

We also note that verification conditions are obtained from the computation in a syntax-directed and compositional manner, so that an HTT computation can be seen as (part of) a proof of its specification[1] — there is no need for whole-program reasoning.

We next present the syntax of HTT and comment on the various constructors.

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

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

The separation into intro and elim terms facilitates bidirectional typechecking [35], whereby most of the type information can be omitted from the terms, and inferred automatically. When type information must be supplied explicitly, the elim term $M{:}A$ can be used. In the typing rules in Section 3, $M{:}A$ will indicate direction switch during bidirectional typechecking. More importantly for our purposes, this kind of formulation also facilitates equational reasoning via hereditary substitutions (described below), as it admits a simple syntactic criterion for normality with respect to beta reduction. For example, the reader may notice that an HTT term which

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

does not use the constructor $M{:}A$ may not contain beta redexes. This is the primary reason why we do not use the more familiar monadic constructs return and bind in this presentation.

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

Each computation is a semicolon-separated list of commands. The primitive commands are as follows (where x is always a bound variable): (1) $x = \mathsf{alloc}_\tau(M)$ allocates space in the heap and initializes it with $M{:}\tau$. The address of the allocated space is returned in x, and is guaranteed to be "fresh". (2) $[M]_\tau = N$ updates the heap so that the location M points to the term $N{:}\tau$. To perform this operation, we must prove that the location M is allocated, but we need not establish that it holds a value of type τ. That is, the operation supports *strong updates*—the ability to change the contents of a location to a value of arbitrary type. (3) $x = [M]_\tau$ looks up the term that the current heap assigns to the location M, and binds the result to x. To perform this operation, we must prove that the location M indeed points to a term of type τ in the current heap. (4) $\mathsf{dealloc}(M)$ frees the heap space pointed to by M. To perform this operation, we must prove that M is allocated. (5) $x = \mathsf{if}_A(M, E_1, E_2)$ is a conditional which executes the computation E_1 or E_2 depending on the value of the Boolean term M. The resulting value is stored in x. (6) $x = \mathsf{fix}_A(M, f.y.E)$ is a recursion construct. It first computes the least fixpoint of the equation $f = \lambda y.\,\mathsf{dia}\ E$, immediately applies it to the initial value M, and the resulting computation is activated to compute a result which gets bound to x. (7) The computation that simply consists of an intro term M is the trivial computation that just returns M as its result. (8) The computation $\mathsf{let\ dia}\ x = K\ \mathsf{in}\ E$ activates the computation that is encapsulated and suspended by K, binds the result to x and proceeds to evaluate E, achieving the sequential composition of K and E. The construct is the elimination form for the Hoare types in HTT. A suspended computation can only be activated by another computation, and thus once we enter the effectful fragment of the language, we cannot get out. This is a characteristic property of monadic type systems [24, 40]. In the literature, the let dia construct is often denoted as let val or bind.

Types. The types of HTT include the primitive types of Booleans and natural numbers, unit type 1, dependent functions $\Pi x{:}A.\,B$, Hoare triples $\Psi.X.\{P\}x{:}A\{Q\}$, and polymorphic types $\forall\alpha.\,A$. We write $A \to B$ to abbreviate $\Pi x{:}A.\,B$ when B does not depend on x, and $\Diamond A$ to abbreviate $\{\top\}x{:}A\{\top\}$.

The type $\Psi.X.\{P\}x{:}A\{Q\}$ specifies an effectful computation with a precondition P and a postcondition Q, returning a result of type A. The variable x names the return value of the computation, and Q may depend on x. The contexts Ψ and X list the variables and heap variables, respectively, that may appear in both P and Q, thus helping relate the properties of the beginning and the ending heap. In the literature on Hoare Logic, these are known under the name of *logic variables*. As usual in the literature, logic variables can only appear in the assertions, but not in the programs. Also, in our setting, the type A cannot contain any variables from Ψ and X.

The type $\forall\alpha.\,A$ polymorphically quantifies over the *monotype* variable α. For our purposes, it suffices to define a monotype as any type that does not contain polymorphic quantification, except in the assertions. For example, $\Psi.X.\{P\}x{:}A\{Q\}$ is a monotype when A is a monotype, even if Ψ, P and Q contain polymorphic types. Note that allowing polymorphism in the assertions does not change the predicative nature of HTT. The type system will be formulated so that logic variables and the assertions do not influence the computational behavior or equational properties of effectful computations: if two terms of some Hoare type are semantically equal, then they are equal under any other Hoare type to which they may belong.

Predicative polymorphism (quantification over monotypes) is sufficient for modeling languages such as Standard ML, but not more recent languages such as Haskell. However, extending HTT to support impredicative polymorphism seems difficult as it significantly complicates the termination argument for normalization (see below), which is a crucial component of type equality. Therefore, we leave the treatment of impredicative polymorphism to future work.

Heaps and locations. In this paper, we model memory locations as natural numbers. One advantage of this approach is that it supports some forms of pointer arithmetic which is needed for languages such as Cyclone. We model heaps as finite functions, mapping a location N to a pair (τ, M) where τ is the monotype of M. In this case we say that N *points to* M, or that M is the *contents* of location N, or that the heap *assigns* M to the location N.

We introduce the following syntax for heaps: empty denotes the empty heap, and $\mathsf{upd}_\tau(H, M, N)$ is the heap obtained from H by updating the location M so that it points to N of type τ, while retaining all the other assignments of H.

Heap terms and variables play a prominent role in our encoding of assertions about (propositional) equality and disjointness of heaps. If heaps could hold values of polymorphic type, then encoding these properties would require impredicative quantification. Consequently, we limit heaps to hold only values of monotype.

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

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

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

Hld is the heap equality, $M \in H$ iff the heap H assigns to the location M, share states that H_1 and H_2 agree on the location M, and splits states that H can be split into disjoint heaps H_1 and H_2.

We next define the assertions familiar from Separation Logic [30, 36, 31]. All of these are relative to the free variable mem, which denotes the current heap fragment of reference.

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

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

The operation $[H/h]$ used in the above definitions substitutes the heap H for the heap variable h into heaps and assertions. The substitution commutes with most of the constructors, except that it leaves terms and types invariant. This is justified as terms and types will not depend on free heap variables.

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

Monadic and hereditary substitutions. The equational theory of HTT is based on the usual beta and eta reductions for the various type constructors. The most interesting equations are the ones dealing with Hoare types. These equations should capture the properties of sequential composition of effectful computations. To that end, we define the operation of *monadic substitution* $\langle E/x{:}A\rangle F$, which composes E and F sequentially. The operation is defined by induction on the structure of E.

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

Now we can specify the beta and eta equations for the Hoare types.

$$\mathsf{let\ dia}\ x = (\mathsf{dia}\ E){:}(\Psi.X.\{P\}y{:}A\{Q\})\ \mathsf{in}\ F \Longrightarrow_\beta \langle E/x{:}A\rangle F$$

$$M{:}\Psi.X.\{P\}x{:}A\{Q\} \Longrightarrow_\eta \mathsf{dia}\ (\mathsf{let\ dia}\ y = M{:}\Psi.X.\{P\}x{:}A\{Q\}\ \mathsf{in}\ y)$$

where $y \notin \mathsf{FV}(M{:}\Psi.X.\{P\}x{:}A\{Q\})$. The definition of monadic substitution and the corresponding reduction and expansion are taken directly from the work of Pfenning and Davies [34]. Pfenning and Davies show that these equations are equivalent to the standard monadic equational laws [25], with the benefit that the monadic substitution subsumes the associativity laws of [25], thus simplifying the equational theory.

The general strategy that HTT employs in the equational reasoning is to reduce the expressions to their canonical form (defined below), and then compare the canonical forms for alpha equivalence. This reduction is carried out during type checking, as will be explained in Section 3.

A term is in canonical form if it is beta-normal (i.e. it contains no beta redexes), and eta-long (i.e., all of its intro subterms are eta expanded). For example, if $f{:}(\mathsf{nat}{\to}\mathsf{nat}){\to}(\mathsf{nat}{\to}\mathsf{nat}){\to}\mathsf{nat}$ and $g{:}\mathsf{nat}{\to}\mathsf{nat}$, then the canonical version of the term $f\ \ g$ is $\lambda h.\ f\ (\lambda y.\ g\ y)\ (\lambda x.\ h\ x)$. This definition of canonicity accounts for both beta and eta equations. In order to treat polymorphism, we also need to add a new term constructor $\mathsf{eta}_\alpha\ K$ which is only used in canonical forms, and serves to record that K should be eta expanded, once α is substituted with a concrete monotype.

The main insight, due to Watkins et al. [41], is that conversion to canonical forms can be defined on (possibly) ill-typed terms, and can be shown to terminate. This is important, as it will allow us to avoid the mutual dependency between equational reasoning and typechecking, which is one of the main sources of complexity in dependent type theories.

At the center of the development are *hereditary substitutions* [41], which are defined only on canonical forms, and preserve canonicity. For example, in places where an ordinary capture-avoiding substitution creates a redex like $(\lambda x.\ M)\ N$, a hereditary substitution continues by immediately substituting N for x in M. This may produce another redex, that is immediately reduced initiating another hereditary substitution and so on. To ensure termination, hereditary substitutions are parametrized by a metric based on types, which decreases as the substitution proceeds.

Space precludes us from presenting the formal definition of hereditary substitutions here (see [27] for details), but they have the form $[M/x]^*_A(-)$, and they substitute the canonical form M for a variable x into a given argument. The superscript $*$ ranges over $\{k, m, e, a, p, h\}$ and determines the syntactic domains of the argument (elim terms, intro terms, computations, types, assertions and heaps, respectively). The subscript A is a putative type of M, and is used to ensure the termination of the substitution. We also need a monadic hereditary substitution $\langle E/x\rangle_A(-)$, and a monotype substitution $[\tau/\alpha]^*(-)$. The later performs an on-the-fly eta expansion with respect to τ of any subterms in the argument of the form $\mathsf{eta}_\alpha\ K$.

The substitutions are defined by nested induction, first on the structure of A, and then on the structure of the term being substituted into (in case of the monadic substitution, we use the substituted computation instead). In other words, we either go to a smaller type, in which case the expressions may become larger, or the type remains the same, but the expressions decrease. Note that without the restriction to predicative polymorphism, types could actually grow after a substitution, hence our restriction to polymorphism over monotypes.

Theorem 1 (Termination of hereditary substitutions) *$[M/x]^*_A(-)$, $\langle E/x\rangle_A(-)$ and $[\tau/\alpha]^*(-)$ terminate, either by returning a result, or failing in a finite number of steps.*

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

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

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

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

One possible specification which covers both aliasing and non-aliasing has the precondition $(\mathsf{x} \mapsto_\alpha \mathsf{m} * \mathsf{y} \mapsto_\beta \mathsf{n}) \vee (\mathsf{x} \mapsto_\alpha \mathsf{m} \wedge \mathsf{y} \mapsto_\beta \mathsf{n})$, with the symmetric postcondition. The second disjunct uses $\wedge$ instead of $*$, and can be true only if the heap contains exactly one location, thus forcing $x = y$. This specification is interesting because it precisely describes the smallest heap needed for swap as the heap containing only x and y.

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

of the invariance that one may express in higher-order separation logic [4].

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

$$\begin{array}{l}\mathsf{identity} : \forall\alpha.\forall\beta.\Pi\mathsf{x}{:}\mathsf{nat}.\Pi\mathsf{y}{:}\mathsf{nat}.\\ \qquad \mathsf{h}.\{\mathsf{x} \hookrightarrow_\alpha \text{-} \wedge \mathsf{y} \hookrightarrow_\beta \text{-} \wedge \mathsf{this(h)}\}\ \mathsf{r} : 1\ \{\mathsf{this(h)}\}\\ = \quad \lambda\mathsf{x}.\ \lambda\mathsf{y}.\ \mathsf{dia}(\mathsf{let\ dia\ u} = \mathsf{swap}\ \alpha\ \beta\ \mathsf{x}\ \mathsf{y}\\ \qquad\qquad\qquad\qquad \mathsf{dia\ v} = \mathsf{swap}\ \beta\ \alpha\ \mathsf{x}\ \mathsf{y}\ \mathsf{in}\ ())\end{array}$$

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

3. Type system

The type system of HTT consists of the following judgments.

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

The judgments on the right deal with formation and canonicity of variable contexts, assertion contexts, assertions, types, monotypes and heaps. In these judgments, the output is always the canonical version of the main input (Δ' is canonical for Δ, P' is canonical for P, etc). When checking assertion contexts (Γ propctx), Γ is required to be canonical, so there is no need to return the output.

The judgments on the left side of the above table are the primary ones, and are explicitly oriented to symbolize whether the type or the assertion are given as input or are synthesized as output. This is a characteristic feature of bidirectional typechecking [35], which we here employ for both terms and computations.

For example, the judgment $\Delta \vdash K \Rightarrow A\,[N']$ takes an elim form K and input context Δ and outputs the type A of K and the canonical form N'. On the other hand, $\Delta \vdash N \Leftarrow A\,[N']$ takes an intro form N and input context Δ and input type A, and outputs the canonical form N' if N matches A.

The judgment $\Delta; P \vdash E \Rightarrow x{:}A.\,Q\,[E']$ takes a computation E, input context Δ, input assertion P, and input type A, and outputs the strongest postcondition Q for E with respect to the precondition P, and the canonical form E' of E. Symmetrically, $\Delta; P \vdash E \Leftarrow x{:}A.\,Q\,[E']$ takes computation E, input context Δ, input assertions P and Q and input type A, and outputs the canonical form E', if Q is a postcondition (not necessarily the strongest) for E with respect to P. The canonical form E' is computed using only the beta and eta rules for the type constructors. Other kinds of equational reasoning, like arithmetic or unrolling of recursive calls, are not part of definitional equality, and hence does not factor into the computation of canonical forms.

The judgment $\Delta; X; \Gamma_1 \Longrightarrow \Gamma_2$ formalizes the sequent calculus for the assertion logic, which is a classical multi-sorted logic with polymorphism. The Δ is a variable context, X is a heap context, and Γ_1, Γ_2 are sets of assertions. As usual in sequent calculi, the judgment holds if for every instantiation of the variables in Δ and X such that the conjunction of assertions in Γ_1 holds, the disjunction of assertions in Γ_2 holds as well.

The input and output contexts and types in all the above judgments are always assumed canonical.

Terms. We only discuss selected rules here, and refer to the accompanying technical report [27] for the treatment of the primitive types nat and bool and their corresponding operations. We fist need several auxiliary functions which deal with beta reduction and eta expansion. The functions $\mathsf{apply}_A(M, N)$ and $\mathsf{spec}(M, \tau)$ normalize the applications $M\ N$ and $M\ \tau$, respectively, if these applications contain a redex. The function $\mathsf{expand}_A(N)$ eta expands the term N with respect to A. We note that the results of eta expansion are invariant with respect to the possible assertions that may appear in A, so that we can assume that A is a simple type. Here M, N and τ are assumed canonical.

$$\begin{array}{lcll}\mathsf{apply}_A(K, M) & = & K\ M & \text{if } K \text{ is an elim term} \\ \mathsf{apply}_A(\lambda x.\,N, M) & = & N' & \text{where } N' = [M/x]^m_A(N) \\ \mathsf{apply}_A(N, M) & & \text{fails} & \text{otherwise} \\ & & & \\ \mathsf{spec}(K, \tau) & = & K\ \tau & \text{if } K \text{ is an elim term} \\ \mathsf{spec}(\Lambda\alpha.\,M, \tau) & = & [\tau/\alpha]^m(M) & \\ \mathsf{spec}(N, \tau) & & \text{fails} & \text{otherwise} \\ & & & \\ \mathsf{expand}_a(K) & = & K & \text{if } a \text{ is nat or bool} \\ \mathsf{expand}_\alpha(K) & = & \mathsf{eta}_\alpha\ K & \\ \mathsf{expand}_1(K) & = & () & \\ \mathsf{expand}_{\forall\alpha.\,A}(K) & = & & \text{where } \alpha \notin \mathsf{FTV}(K) \\ \quad \Lambda\alpha.\,\mathsf{expand}_A(K\ \alpha) & & & \\ \mathsf{expand}_{A_1 \to A_2}(K) & = & & \text{where } M = \mathsf{expand}_{A_1}(x) \\ \quad \lambda x.\,\mathsf{expand}_{A_2}(K\ M) & & & \text{and } x \notin \mathsf{FV}(K) \\ \mathsf{expand}_{\diamond A}(K) & = & & \text{where } M = \mathsf{expand}_A(x) \\ \quad \mathsf{dia}\ (\mathsf{let\ dia}\ x = K\ \mathsf{in}\ M) & & & \\ \mathsf{expand}_A(N) & = & N & \text{if } N \text{ is not elim}\end{array}$$

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

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

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

$$\frac{\Delta, \alpha \vdash M \Leftarrow A\,[M']}{\Delta \vdash \Lambda\alpha.\,M \Leftarrow \forall\alpha.\,A\,[\Lambda\alpha.\,M']}\ \forall\mathsf{I}^\alpha$$

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

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

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

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

As described before, intro terms are checked against a supplied type, and elim terms can synthesize their type. The latter holds because elim terms are generally of the form $x\ T_1\ T_2 \cdots T_n$, applying a variable x to a sequence of intro terms or types T_i. Since the type of x is declared in the context of the judgment, the type of the whole application can always be inferred by instantiating the type of x with T_i.

The typing rules now make it explicit how the typing information flows through the system. For example, ΠI checks that term $\lambda x.\,M$ has the given function type, and if so, returns the canonical form $\lambda x.\,M'$. In ΠE we first synthesize the canonical type $\Pi x{:}A.\,B$ and the canonical form N' of the function part of the application. Then the synthesized type is used in checking the argument part of the application. The result type of the whole application is synthesized using hereditary substitutions in order to remove the dependency of the type B on the variable x. Finally, we compute the canonical form of the whole application, using the auxiliary function apply to reduce the term $N'\ M'$ should this term

actually be a redex. Similar description applies to the rules for polymorphic quantification.

In the rule $\Leftarrow\!\Rightarrow$, we need to synthesize the canonical type for the ascription $M{:}A$. This type should clearly be the canonical version of A, under the condition that M actually has this type. Thus, we first test that A is well-formed and compute its canonical form A', and then proceed to check M against A'. If M and A' match, we obtained the canonical version M' of M. Then M' and A' are returned as the output of the judgment.

In the rule $\Rightarrow\!\Leftarrow$, we are checking an elim term K against a canonical type B. But K can already synthesize its canonical type A, so we simply need to check that A and B are actually equal canonical types. The canonical form synthesized from K in the premise, may be an elim form (because it is generated by a judgment for elim forms), but we need to use it in the conclusion as an intro form. The switch from an elim form to the equivalent intro form is achieved by eta expansion with respect to the supplied type B. For example, if $x{:}\mathsf{nat}{\to}\mathsf{nat}$ is a variable in context, then its canonical form is $\lambda y.\, x\ y$, and we could use the rule $\Rightarrow\!\Leftarrow$ to derive the judgment $x{:}\mathsf{nat}{\to}\mathsf{nat} \vdash x \Leftarrow \mathsf{nat}{\to}\mathsf{nat}\,[\lambda y.\, x\ y]$.

When the types A and B in the rule $\Rightarrow\!\Leftarrow$ are equal to some type variable α, we cannot eta expand the canonical forms, so we simply remember that expansion must be done whenever α is instantiated with a concrete monotype (please see the definition of the auxiliary function expand). This is why we introduced the constructor $\mathsf{eta}_\alpha\ K$ which is used only in canonical terms. $\mathsf{eta}_\alpha\ K$ is an intro term, because its occurrences are always generated when using the rule $\Rightarrow\!\Leftarrow$ to switch from elim into intro terms.

Of course, once $\mathsf{eta}_\alpha\ K$ is introduced, we need to be able to typecheck it, and we use the rule eta for that. Notice how this rule insists that K is canonical by requiring in the premise that K equals its own canonical form.

Computations. The judgment $\Delta; P \vdash E \Rightarrow x{:}A.\, Q\,[E']$ translates the program E into a corresponding binary relation on heaps.

Intuitively, the precondition P is a relation that the translation starts with, and the postcondition Q is the relation that captures the semantics of E. In addition, the precondition P has to be strong enough to guarantee that the execution of E will never get stuck. The assertions P and Q use the heap variables init and mem to stand for the input and the output heaps of the computations.

In order to define the small footprint semantics of the Hoare types, we first need two new connectives. The *relational composition* $P \circ Q = \exists h{:}\mathsf{heap}.\, [h/\mathsf{mem}]P \wedge [h/\mathsf{init}]Q$, expresses temporal sequencing of heaps. The informal reading of $P \circ Q$ is that Q holds of the current heap, which is itself obtained from another past heap of which P holds.

The *difference operator* on assertions is defined as $R_1 \multimap R_2 = \forall h{:}\mathsf{heap}.\, [\mathsf{init}/\mathsf{mem}](R_1 * \mathsf{this}(h)) \supset R_2 * \mathsf{this}(h)$ where R_i are assumed to have a free variable mem, but not init. The informal reading of $R_1 \multimap R_2$ is that the heap mem is obtained from the initial heap init by replacing a fragment satisfying R_1 with a new fragment which satisfies R_2. The rest of the heaps init and mem agrees. It is not specified, however, which particular fragment of init is replaced. If there are several fragments satisfying R_1, then each of them could have been replaced, but the replacement is always such that the result satisfies R_2. The operator $\multimap$ is used in the typing judgments to describe a difference between two successive heaps of the computation. Notice how the definition of $\multimap$ relies on naming the heap h by means of universal quantification in order to state its invariance. We could not define an operator with this semantics using the spatial connectives $*$ and $\mathrel{-\!\!*}$ alone.

Now consider a suspended computation $\mathsf{dia}\ E$ with the Hoare type $\Psi.X.\{R_1\}x{:}A\{R_2\}$. Intuitively, the computation and the type should correspond if the following three requirements are satisfied: (1) Assuming that the initial heap can be split into two disjoint parts h_1 and h_2 such that R_1 holds of h_1, then E does not get stuck if executed in this initial heap. Moreover, E never touches h_2 (not even for a lookup); in other words, h_2 is not in the footprint of E. (2) Upon termination of E, the fragment h_1 is replaced with a new fragment which satisfies R_2, while h_2 remains unchanged. (3) The split into h_1 and h_2 is not decided upon before E executes, and need not be unique. We only know that if a split is possible, then the execution of E defines one such split, but which split is chosen may depend on the run-time conditions. Whichever values h_1 and h_2 end up taking, however, we know that (2) holds.

The above requirements define what it means for the specification in the form of Hoare type $\Psi.X.\{R_1\}x{:}A\{R_2\}$ to possess the small footprint property. We argue next that the requirements are satisfied by E if we can establish that $\Delta; P \vdash E \Leftarrow x{:}A.\, Q$, where $P = \mathsf{this}(\mathsf{init}) \wedge \exists\Psi.X.(R_1 * \top)$ and $Q = \forall\Psi.X.R_1 \multimap R_2$.

The assertion P is related to the requirements (1) and (3). Indeed, P states that the initial heap can be split into h_1 and h_2 so that h_1 satisfies R_1 and h_2 satisfies $\top$, as required. In order to ensure progress, the typing judgment will allow E to touch only locations whose existence can be proved. Because there is no information available about h_2 and its locations (knowing $\top$ amounts to knowing nothing), E will be restricted to working with h_1 only. The split into h_1 and h_2 is arbitrary, satisfying an aspect of (3).

The assertion Q is related to the requirements (2) and (3). After unraveling the definition of the $\multimap$ operator, Q essentially states that any split into h_1 and h_2 that E may have induced on init results in a final heap where h_1 is replaced with a fragment satisfying R_2, while h_2 remains unchanged. The invariance of h_2 is precisely what (2) requires, and the parametricity of R_2 with respect to the split is the remaining aspect of (3).

Before we can state the inference rules of the computation judgments, we need an auxiliary function $\mathsf{reduce}_A(M, x.\, E)$ which reduces the term $\mathsf{let\ dia}\ x = M\ \mathsf{in}\ E$, if it contains a redex. Here A, M and E are assumed canonical.

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

We can now present the typing rules for computations. We start with the general monadic fragment, and then proceed with the rules for the individual commands.

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

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

$$\frac{\Delta; \mathsf{this}(\mathsf{init}) \wedge \exists\Psi.X.(R_1 * \top) \vdash E \Leftarrow x{:}A.\, \forall\Psi.X.R_1 \multimap R_2\,[E']}{\Delta \vdash \mathsf{dia}\ E \Leftarrow \Psi.X.\{R_1\}x{:}A\{R_2\}\,[\mathsf{dia}\ E']}\ \{\ \}\mathsf{I}$$

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

The rule consq allows the weakening of the strongest postcondition R into an arbitrary postcondition Q, assuming that R implies Q. The rule comp types the trivial computation that immediately returns the result $x = M$ and performs no changes to the heap. The precondition is simply propagated into the postcondition, but the postcondition must also assert the equality between M and (the canonical form of) x. The rule $\{\ \}\mathsf{I}$ defines the small footprint se-

mantics of Hoare types. This is achieved with using the premise $\Delta; P \vdash E \Leftarrow x{:}A.\, Q$, for P and Q as discussed before.

The rule $\{\ \}\mathsf{E}$ describes how a suspended computation $K \Rightarrow \{R_1\}x{:}A\{R_2\}$ can be sequentially composed with another computation E. The composition is meaningful if the following are satisfied. First, the the assertion logic must establish that the precondition P ensures that the current heap contains a fragment satisfying the precondition R_1, as required by K. In other words, we need to show that $P \Longrightarrow \exists\Psi.X.(R_1 * \top)$. Second, the computation E needs to check against the postcondition obtained after executing K. The latter is taken to be $P \circ \forall\Psi.X.R_1 \multimap R_2$, expressing that the execution of K changed the heap P by replacing a fragment satisfying R_1 with a new fragment satisfying R_2. The normal form of the whole computation is obtained by invoking the auxiliary function reduce. We emphasize that the type B in the conclusion of the $\{\ \}\mathsf{E}$ rule is an *input* of the typing judgments, and is by assumption well-formed in the context Δ. In particular, x does not appear in B, so no special considerations are needed passing from the premise of the rule to the conclusion. No such assumptions are made about the postcondition Q, which is an output of the judgment, so we need to existentially abstract x in the postcondition of the conclusions, to avoid dangling variables. A similar remark applies to the rules for the specific effectful constructs for allocation, lookup, strong update and deallocation that we present next.

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

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

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

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

In the case of allocation, E is checked against the assertion $P * (x \mapsto_{\tau'} M')$, which describes the state after the allocation, and is the strongest postcondition for allocation with respect to P. The assertion simply states that the newly allocated memory whose address is stored in x is disjoint from any already allocated memory described in P.

In the case of lookup, the strongest postcondition states that the heap has not changed (i.e., P still holds) but we have the additional knowledge that the variable x stores the looked up value. The variable x is expanded because we only consider assertions in canonical form. In order to ensure progress, we must prove the sequent $P \Longrightarrow M' \hookrightarrow_{\tau'} -$ showing that the location M' actually exists in the current heap, and points to a value of an appropriate type.

It is important to notice that proving the sequent $P \Longrightarrow M' \hookrightarrow_{\tau'} -$ may be postponed, as it does not influence the other premises. The sequent can be seen as part of the verification condition which is generated during typechecking. This property will be true of all the sequents involved in the computation judgments.

The strongest postcondition for update states that the heap has changed by replacing some assignment $M' \mapsto -$ with an assignment $M' \mapsto_{\tau'} N'$. A prerequisite is to prove the sequent $P \Longrightarrow M' \hookrightarrow -$, thus showing that M' was allocated with an arbitrary type (hence the update is strong).

The strongest postcondition for deallocation states that the heap has changed by replacing the assignment $M' \mapsto -$ with empty. The side condition is the sequent $P \Longrightarrow M' \hookrightarrow -$ showing that M' was allocated.

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

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

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

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

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

Sequents. The sequent calculus is a standard formulation of a first-order classical multi-sorted logic with equality and universal and existential polymorphic quantification over monotypes. The sorts include bools, nats (in Peano axiomatization), functions and type functions with extensionality, effectful computations and heaps. The axiomatization of bools, nats, functions and type functions is standard, and we currently do not consider any specific reasoning principles about computations, except propositional equality. Here, we only present the axioms related to heaps, and refer to [27] for the rest of the rules.

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

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

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

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

The first rule states that an empty heap does not contain any assignments. The second and the third rule implement the McCarthy axioms for functional arrays [23], relating the seleq and upd functions. The fourth axiom asserts a version of heap functionality: a heap may assign at most one value to a location, for each given type.

We would prefer a slightly stronger fourth axiom here, which would state that a heap assigns at most one type and value to a location, instead of at most one value for each type. As an illustration, in our previous example we used the assertion $P = x \mapsto_\alpha m \wedge y \mapsto_\beta n$ to specify a heap which contains exactly one location thus forcing x and y to be aliases. While $x = y$ could be derived from P, we cannot derive that $\alpha = \beta$ and $m = n$ with our weak fourth axiom.

Obviously, stating the full functionality of heaps requires new assertions for equality of types and for equality of terms at different types [22], which we leave for future work.

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

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

```
sumfunc : Πn:nat. {emp} r : nat {emp ∧ sum(r, n)} =
 λn. dia(a = alloc_nat(0);
        P0:{this(init) * (a ↦_nat 0)}
        x = fix(0, f. i.
                  P1:{this(init) ∧ (I * ⊤)}
                  s = [a]_nat;
                  P2:{P1 ∧ a ↪_nat s}
                  t = if eq(i, n) then
                        P3:{P2 ∧ Id_nat(i, n)}
                        s
                      else
                        P4:{P2 ∧ ¬ Id_nat(i, n)}
                        [a]_nat = s+i+1;
                        P5:{P4 ∘ (a ↦_nat - -∘ a ↦_nat s+i+1)}
                        let dia x = f (i+1)
                        in
                          P6:{P5 ∘ ([i+1/i]I -∘ Q)}
                          x
                        end;
                  P7:{(P3 ∧ Id_nat(t, s)) ∨
                      (∃x:nat. P6 ∧ Id_nat(t, x))}
                  t);
        P8:{P0 ∘ ([0/i]I -∘ Q)}
        dealloc(a);
        P9:{P8 ∘ (a ↦_nat - -∘ emp)}
        x);
```

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

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

4. Properties

In this section, we present the most characteristic properties of HTT. The formal development is too extensive to be included here, and the interested reader is referred to the accompanying technical report [27] for the complete statements of all the theorems and all the proofs.

Theorem 2 (Relative decidability of type checking)
Given an oracle for deciding the validity of assertion logic sequents, all the typing judgments of the HTT are decidable.

The proof of Theorem 2 exploits the fact that the typing judgments of HTT, including the computation judgments, are syntax directed, so that typechecking the premises always involves typechecking smaller expressions. Premises may also involve deciding equality of types, or computing hereditary substitutions or deciding sequents of the assertion logic. As the first two kinds of premises are decidable, according to Theorem 1, the conclusion follows.

It should be possible to remove the assumption about the oracle by extending the HTT terms with certificates for the sequents, in the style of Proof-Carrying Code [29]. With this extension, a computation judgment of HTT will contain all the information needed to establish its own derivation, as the derivation is completely guided by the syntax of the computation. In the terminology of Martin-Löf [21], the judgments become *analytic*, or self-evident. Alternatively, we can say that an HTT computation can be seen as a proof of its own specification, and thus the effectful fragment of HTT establishes the Curry-Howard correspondence [15] between computations and specification proofs.

The next lemma restates in the context of HTT the usual properties of Hoare Logic, like weakening of the consequent and strengthening of the precedent. Also included is the frame rule from Separation Logic which embodies the small footprint property by stating that the computation cannot change any heap fragment disjoint from the footprint.

Lemma 3 (Properties of computations)
Suppose that $\Delta; P \vdash E \Leftarrow x{:}A.\, Q\, [E']$. *Then:*

1. Weakening Consequent. *If* $\Delta, x{:}A; \mathit{init}, \mathit{mem}; Q \Longrightarrow R$, *then* $\Delta; P \vdash E \Leftarrow x{:}A.\, R\, [E']$.
2. Strengthening Precedent. *If* $\Delta; \mathit{init}, \mathit{mem}; R \Longrightarrow P$, *then* $\Delta; R \vdash E \Leftarrow x{:}A.\, Q\, [E']$.
3. Frame. *If* $\Delta \vdash \mathit{dia}\ F \Leftarrow \Psi.X.\{R_1\}x{:}A\{R_2\}\, [F']$, *and* $\Delta, \Psi; X, \mathit{mem} \vdash R \Leftarrow \mathsf{prop}\, [R]$, *then* $\Delta \vdash \mathit{dia}\ F \Leftarrow \Psi.X.\{R_1 * R\}x{:}A\{R_2 * R\}\, [F']$.
4. Preservation of History. *If* $\Delta; \mathit{init}, \mathit{mem} \vdash R \Leftarrow \mathsf{prop}\, [R]$, *then* $\Delta; R \circ P \vdash E \Leftarrow x{:}A.\, (R \circ Q)\, [E']$.

We discuss here the last property from Lemma 3, which we call Preservation of History. It essentially states that a computation does not depend on how the heap in which it executes has been obtained, i.e., which sequence of computations lead to its creation. Thus, each precondition P and postcondition Q can always be arbitrarily precomposed with a new assertion R. This is one of the most important

properties of HTT and is indispensable in the meta-theoretic proofs, because it captures the fact that HTT reasons about programs by computing strongest postconditions via relational composition.

5. Operational semantics

In this section we discuss the operational semantics for HTT and the soundness of the type system with respect to the operational semantics. In particular, we argue that if $\Delta; P \vdash E \Leftarrow x{:}A.\, Q$ is derivable in the type system, then it is indeed the case that evaluating E in a heap in which P holds produces a heap in which Q holds (if E terminates).

The operational semantics is only defined for well-typed terms. Since our types correspond to specifications, our approach is different from the traditional approach of Hoare Logic but it is similar to the approach in [6], which also only gives semantics to well-specified programs.

Syntax. We now present the syntactic ingredients for defining a call-by-value, left-to-right operational semantics.

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

The definition of values is standard from mostly functional programming languages. We use l to range over nats when they are used as pointers.

Value heaps are assignments from nats to values, where each assignment is indexed by a type. Value heaps are a run-time concept – and are used in the evaluation judgments to describe the state in which programs execute. This is in contrast to heaps from Section 2 which are used for reasoning in the assertion logic. That the two notions correspond to each other is expressed by our definition of heap soundness that will be given later in this section. We will need to convert a value heap into a heap canonical form, so we introduce the following conversion function.

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

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

A control expression $\kappa \triangleright E$ pairs up a computation E and a continuation κ, so that E provides the initial value with which the execution of κ can start. Thus, a control expression is in a sense a self-contained computation. Control expressions are introduced because they make the call-by-value semantics of the computation $\mathsf{let\ dia}\ x = \mathsf{dia}\ E\ \mathsf{in}\ F$ explicit. Evaluation of this computation is carried out by creating the control expression $x.\, F \triangleright E$; or in other words, first push $x.\, F$ onto the continuation, and proceed to evaluate E.

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

Our theorems require a typing judgment for abstract machines, in order to specify the type of the return value and the properties of the heap in which the abstract machine terminates (if it does). Given $\mu = \chi, \kappa \triangleright E$, we write $\vdash \mu \Leftarrow x{:}A.\, Q$ if we can prove that Q is a postcondition for $\kappa \triangleright E$ with respect to the assertion $[\![\chi]\!]$ generated from χ.

Evaluation. There are three evaluation judgments in HTT; one for elimination terms $K \hookrightarrow_k K'$, one for introduction terms $M \hookrightarrow_m M'$ and one for abstract machines $\chi, \kappa \triangleright E \hookrightarrow_e \chi', \kappa' \triangleright E'$. Each judgment relates an expression with its one-step reduct. The inference rules of the evaluation judgments are straightforward, so we omit them here. We refer to the technical report [27] for the complete formalization.

Soundness. Perhaps somewhat surprisingly for a program logic like HTT, we formulate soundness via Preservation and Progress theorems as often used for simpler type systems. This is a consequence of our decision to formulate HTT as a type theory, rather than as an ordinary Hoare Logic. Of course, our Preservation and Progress theorems are significantly stronger (and also harder to prove) than corresponding theorems for simpler type systems since our types are much more expressive.

Theorem 4 (Preservation)

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

The preservation theorem states that the evaluation step on a well-specified term/abstract machine does not change the specification of the result. In the case of abstract machines, after taking the step, the evaluation is still on its way to produce a value of type A, and terminate in a heap satisfying Q. In the case of pure terms, there is an additional claim that evaluation preserves the canonical form—and thus the equational properties—-of the evaluated term. In other words, normalization is adequate for the operational semantics.

Before we can state the progress theorem, we need to define a property of the assertion logic which we call *heap soundness*.

Definition 5 (Heap soundness)

The assertion logic of HTT is heap sound iff for every value heap χ,

1. *if* $\cdot$; *mem*; *this*$([\![\chi]\!]) \Longrightarrow l \hookrightarrow_\tau -$, *then* $l \mapsto_\tau v \in \chi$, *for some value* v, *and*
2. *if* $\cdot$; *mem*; *this*$([\![\chi]\!]) \Longrightarrow l \hookrightarrow -$, *then* $l \mapsto_\tau v \in \chi$ *for some monotype* τ *and a value* v.

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

Now we can state the Progress theorem, under the assumption of heap soundness; in the following section we prove that the assertion logic of HTT is indeed heap sound.

Theorem 6 (Progress)

Suppose that the assertion logic of HTT is heap sound. Then the following holds.

1. *If* $\cdot \vdash K_0 \Rightarrow A\,[N']$, *then either* $K_0 = v : A$ *or* $K_0 \hookrightarrow_k K_1$, *for some* K_1.
2. *If* $\cdot \vdash M_0 \Leftarrow A\,[M']$, *then either* $M_0 = v$ *or* $M_0 \hookrightarrow_m M_1$, *for some* M_1.
3. *If* $\vdash \chi_0, \kappa_0 \triangleright E_0 \Leftarrow x{:}A.\, Q$, *then either* $E_0 = v$ *and* $\kappa_0 = \cdot$, *or* $\chi_0, \kappa_0 \triangleright E_0 \hookrightarrow_e \chi_1, \kappa_1 \triangleright E_1$, *for some* χ_1, κ_1, E_1.

Example. From the Progress and Preservation theorem it is now clear that sumfunc 10 produces a computation that, if it terminates when executed in an empty heap, returns the value 55 and an empty heap.

6. Heap soundness

In this section we sketch a proof that the assertion logic of HTT is heap sound. We do so by means of a simple denotational semantics of HTT. It is based on the observation that the operational semantics does not depend on HTT types and, likewise, the atomic predicates of the assertion logic do not depend on HTT types, but only on the underlying simple types (which we call *shapes*) obtained after erasing assertions from HTT types. Hence we may devise a simple semantics of the language in which types are interpreted by a domain of values, and in which assertions are interpreted as subsets of the domain of values. For simplicity, we here use a denotational semantics; one could also have made a model directly from the operational semantics and modeled the type of values as ground contextual equivalence classes of terms, but that requires showing operational extensionality properties of functions, which is non-trivial in the presence of general references.

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

Let $\mathrm{MonoTypes}$ denote the set of mono types of HTT. Let N denote the discrete cpo of natural numbers, let B denote the discrete cpo of booleans with elements true and false, and let 1 denote the one-element cpo with element $*$. Finally, let Loc be a copy of N. Recall that $\mathbf{pCpo}$ is bilimit compact and complete. Hence there is a canonical solution to the following recursive domain equations:

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

where the ordering of $\Sigma_{L \in P_{\mathit{fin}}(\mathit{Loc})}(L \to V)$ only relates records (heaps) with equal domain; two records with equal domain are ordered pointwise.

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

Given the above, the actual definition of the semantics, is fairly standard. For example, $[\![\Delta \vdash \lambda x.\, M \Leftarrow \Pi x{:}A.\, B\,[\lambda x.\, M]]\!]_\theta$ is

$$\lambda\rho.\,(\lambda v.\ [\![\Delta, x{:}A \vdash M \Leftarrow B\,[M]]\!]_\theta(\rho, v)),$$

and $[\![\Delta; X \vdash P \wedge Q \Leftarrow \mathsf{prop}\,[P \wedge Q]]\!]_\theta$ is

$$[\![\Delta; X \vdash P \Leftarrow \mathsf{prop}\,[P]]\!]_\theta \cap [\![\Delta; X \vdash Q \Leftarrow \mathsf{heap}\,[Q]]\!]_\theta\,.$$

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

$$\begin{array}{l} [\![\Delta; X \vdash P_1 \wedge \cdots \wedge P_n]\!](\rho, \mu) \\ \quad \subseteq [\![\Delta; X \vdash Q_1 \vee \cdots \vee Q_m]\!](\rho, \mu). \end{array}$$

Theorem 7 (Soundness of Assertion Logic)
All the axioms and rules of the assertion logic are sound with respect to the semantic notion of validity.

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

Theorem 8 (Heap Soundness)
The assertion logic of HTT is heap sound.

Proof: Let χ be a value heap. Here we only sketch the argument for item 1 of heap soundness.

By assumption $\cdot; \mathsf{mem}; \mathsf{Hld}(\mathsf{mem}, [\![\chi]\!]) \Longrightarrow \mathsf{seleq}_\tau(\mathsf{mem}, l, -)$ is derivable, so by logic also $\cdot; \cdot; \cdot \Longrightarrow \mathsf{seleq}_A([\![\chi]\!], l, -)$ is derivable. By soundness of the assertion logic (Theorem 7) and the definition of the semantics of the assertion logic, we have that $[\![\cdot; \cdot; \cdot \Longrightarrow \mathsf{seleq}_A([\![\chi]\!], l, -)]\!](*, *)$ is true. By definition of the interpretation of seleq_A this means that $\exists v \in V.\ [\![[\![\chi]\!]]\!](*, *)(l) = v$. By the definition of $[\![\chi]\!]$ and the semantics of heaps, we have that $l \mapsto_A v_0 \in \chi$, for some value v_0, as required (and $[\![v_0]\!] * $ is the v that exists). ■

Remark 9
Note that the denotational model above does not model predicates as *admissible*[2] subsets, but rather as all subsets. One might have expected admissibility to show up since HTT contains a rule for fixed points (see Section 3) but because the denotational model is so crude (it only models the shape of HTT types, not HTT types themselves) and since it is only used to show heap soundness, while operational methods are used to show soundness of the typing rule for fixed points, we do not need to restrict attention to admissible predicates in the denotational model. We are not aware of similar combinations of models and proof methods for models of higher-order store in the literature.

7. Related work

There has been a significant interest recently in systems for reasoning about effectful higher-order functions. Honda et al. [14, 3] present several Hoare Logics for total correctness, where specifications in the form of Hoare triples are taken as propositions. Krishnaswami [18] proposes a version of Separation Logic for a higher-order typed language. Similarly to HTT, Krishnaswami bases his

[2] A subset of a pointed cpo is admissible if it is pointed and closed under sups of chains.

logic on a monadic presentation of the underlying programming language. Both proposals do not support polymorphism, strong updates, deallocation or pointer arithmetic. Both are Hoare-like Logics, rather than type theories, which means that logic specifications cannot be used in the program syntax to describe the context in which any particular program fragment can appear. On the other hand, Honda et al. have established a notion of contextual completeness for their framework, which we do not have. Both Honda et al. and Krishnaswami allow their specifications to talk about the abstract type of references. In HTT, like in Separation Logic, we use natural numbers instead, as it was not clear how to axiomatize quantification and induction principles over this abstract type in the context of HTT. It is an interesting future work to devise a type system that can use local state in the definition of abstract types.

Shao et al. [37] and Applied Type Systems (ATS) of Xi et al. [42, 44] present dependently typed systems for effectful programs, based on singleton types, but they do not allow effectful terms in the specifications. Both systems encode a notion of pre- and postconditions. In ATS, assertions are drawn from linear logic, and the proofs for pre- and postconditions are embedded within the code. It is interesting that the properties of linear logic actually require the embedding of proofs and code, unlike in HTT where this is optional. For most effectful commands, a precondition must be transformed into a suitable form (usually a linear product) before the postcondition can be computed at all. The proofs are necessary in order to guide this transformation of preconditions. In other words, they cannot separate type-checking into a decidable verification-condition generation phase, and a sequent validity phase. On the other hand, ATS possesses a very powerful mechanism for definition of generalized algebraic datatypes [43], which we have not considered in HTT yet.

Mandelbaum et al. [20] develop a theory of type refinements for reasoning about effectful higher-order functions, whose foundations are very similar to ours. They use a monadic separation between pure and impure fragments, and their type refinements correspond to pre- and postconditions, just like in HTT. There are significant differences as well. For example, the assertion logic of [20] is a very simple fragment of propositional linear logic in order to facilitate decidable typechecking. The simplicity of this fragment avoids the issues related to explicit proofs that we discussed above for [44], but it also makes it unclear if this approach could support full-fledged state with aliasing, which seems to require quantification in the world refinements. A related problem which the authors discuss in their future work is the lack of features in linear logic to express sharing. They suggest that second-order quantification over worlds will remedy the situation, and indeed, our current development of polymorphism for HTT could be seen as supporting this statement.

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

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

8. Future work

In this section we describe some future work that we plan to carry out, involving higher-order assertion logic and local state.

Higher-order assertion logic. The polymorphic multi-sorted first-order assertion logic presented in the current paper is still insufficient for realistic languages and applications. For any practical application, HTT needs internal means of defining new predicates, including inductive ones, and new types of data. At a minimum, one needs assertions that describe lists, trees, dags, etc. that can be used to describe the shape of mutable data structures within the heap. All of these are definable in higher-order logic [8, 32, 39]. For purposes of HTT, the higher-order logic will also require polymorphic quantification over monotypes.

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

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

First, a computation should have more than one result so that it can return the addresses of locally allocated data. Thus, we will require a new type of Hoare triples, with a syntax as in $\Psi.X.\{P\}\Delta, x{:}A\{Q\}$, where Δ is a context of variables that abstracts over the local data of the computation. The variables from Δ can be used in the return type A and in the postcondition Q. This extension may employ some results from the Contextual modal type theory of [28].

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

References

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[27] A. Nanevski, 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`.

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

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

[30] 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.

[31] 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.

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

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

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

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

[36] 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.

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

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

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

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

[41] 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.

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

[43] H. Xi, C. Chen, and G. Chen. Guarded recursive datatype constructors. In *Symposium on Principles of Programming Languages, POPL'03*, pages 224–235, New Orleans, January 2003.

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