Hoare Type Theory, Polymorphism and Separation

ALEKSANDAR NANEVSKI and GREG MORRISETT
Harvard University
{aleks,greg}@eecs.harvard.edu

LARS BIRKEDAL
IT University of Copenhagen
birkedal@itu.dk

Abstract

We consider the problem of reconciling a dependently typed functional language with imperative features such as mutable higher-order state, pointer aliasing, and non-termination. We propose Hoare Type Theory (HTT), which incorporates Hoare-style specifications into types, making it possible to statically track and enforce correct use of side effects.

The main feature of HTT is the Hoare type $\{P\}x{:}A\{Q\}$ specifying computations with precondition P and postcondition Q that return a result of type A. Hoare types can be nested, combined with other types, and abstracted, leading to a smooth integration with higher-order functions and type polymorphism.

We further show that in the presence of type polymorphism, it becomes possible to interpret the Hoare types in the "small footprint" manner, as advocated by Separation Logic, whereby specifications tightly describe the state required by the computation.

We establish that HTT is sound and compositional, in the sense that separate verifications of individual program components suffice to ensure the correctness of the composite program.

1 Introduction

The static type systems of today's programming languages, such as Java, ML and Haskell, provide a degree of lightweight specification and verification that has proven remarkably effective at eliminating a class of coding errors. Equally important, 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* to provide precise specifications of these requirements. For example, we can specify that a vector has a certain length, by using the type $\mathsf{vector}(n)$ that depends on the length n. Inner product can then be given the signature

$$\mathsf{inner\_prod} : \Pi n{:}\mathsf{nat}.\ \mathsf{vector}(n) \times \mathsf{vector}(n) \to \mathsf{nat}$$

signifying that the argument vectors must be of equal length.

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. An alternative approach is to consider some form of *program logic*, such as Hoare's original logic (Hoare, 1969) or the more recent forms of Separation Logic (O'Hearn *et al.*, 2001; Reynolds, 2002; O'Hearn *et al.*, 2004), 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 to support the abstraction mechanisms such as higher-order functions, polymorphism, and modules.

In this paper, we propose a new approach that smoothly combines dependent types and a Hoare-style logic for a language with higher-order functions and imperative commands (i.e., core, polymorphic 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. Hoare types can be viewed as a refinement of the concept of *monad* (Moggi, 1989; Moggi, 1991; Peyton Jones & Wadler, 1993; Wadler, 1998), which is extensively used in simply-typed functional programming to statically track the computational effects. In the dependent setting, monadic isolation is crucial for ensuring soundness, as it prevents the effects from polluting the logical properties of the underlying pure language. At the same time, it makes it possible for encapsulated effectful terms to freely appear in type dependencies and specifications. Furthermore, Hoare types can be nested, combined with other types, and abstracted within terms, types and predicates alike, thus improving upon the data abstraction and information hiding mechanisms of the original Hoare Logic, and leading to a unified system for programming, specification and reasoning about programs.

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 phase performs a combination of basic type-checking and verification-condition generation, and is completely automatic. The second phase must then show the validity of the generated verification-conditions. The conditions can be fed to an automated theorem prover, discharged by hand, or even ignored, making it possible to use HTT in various ways, ranging from a simple bug-finding tool that can discover some, but not necessarily all errors, to a full-scale program verification and certification framework.

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 (Detlefs *et al.*, 1998; Leino *et al.*, 2000), SPLint (Evans & Larochelle, 2002), Spec# (Barnett *et al.*, 2004), Cyclone (Jim *et al.*, 2002) and JML (Leavens *et al.*, 1999; Burdy *et al.*, 2005), that provide support for extended static checking of programs.

The semantics of program heaps in HTT is based on the treatment of functional arrays of McCarthy (1962) and Cartwright and Oppen (1978). In this classical approach, pre- and postconditions in Hoare triples describe the whole heap, rather than just the heap fragment that any particular program requires. Moreover, the postconditions must explicitly describe how the heap in which the program terminates differs from the heap in which it started. Keeping track of whole heaps may be cumbersome as it requires careful tracking of location inequalities (i.e., lack of aliasing). It is much better to simply assert those properties of the beginning and the ending heap that are actually influenced by the computation, and automatically assume that all the unspecified heap portions remain invariant. This is known as the "small footprint" approach to specification, and has been advocated recently by the work on Separation Logic.

We note that in the presence of type polymorphism, the approach based on functional arrays can already define the connectives from Separation Logic. We thus endow the Hoare types with the small-footprint interpretation, and show that the customary inference rules from Separation Logic (e.g., the Frame rule) are admissible in HTT. Moreover, 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.* (2004). 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.

We prove that HTT is sound and compositional, in the sense that separate verifications of individual program modules suffice to ensure the correctness of the composite program. In other words, the types of HTT are strong enough to serve as adequate interfaces for program components, so that verification does *not* require whole-program reasoning. As customary in type theory, compositionality is expressed via substitution principles. We contrast this to the statement of compositionality for ESC/Modula-3 (Leino & Nelson, 2002), which requires a significantly more involved statement and proof.

We start the presentation with the overview of the syntactic features of HTT (Section 2), and then discuss the type system (Section 3), and the equational reasoning (Section 4). We establish the basic syntactic properties, like substitution principles (Section 5), then define a call-by-value operational semantics (Section 6), and prove the soundness of HTT with respect to this operational semantics using denotational methods (Section 7). We close with the discussion of related and future work.

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 Hoare-style 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 and verification condition generation, 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 (Hofmann, 1995).

Most of the 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. 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. In the current paper, conditionals are considered impure even though they are a customary component of most pure functional languages. This design decision is justified by the well-known complications in the equational reasoning about conditionals (Ghani, 1995; Altenkirch *et al.*, 2001), which are related to the laws for eta expansion. We plan to address these issues in the future work, but for now, simply avoid them by placing the conditionals into the impure fragment.

The split between pure and impure fragments is a familiar one in functional programming. For example, the programming language Haskell (Peyton Jones, 2003) uses monads 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 generic monadic laws (Moggi, 1991) for reasoning about the impure code.

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 conditions for an effectful computation by calculating strongest postconditions. If the verification condition is provable, then the computation matches its specification. The verification conditions are computed during the first phase of typechecking, in a process that is mutually recursive with normalization and equational reasoning, as formalized in Section 3. However, the conditions can be proved in the second phase, so that the complexity and undecidability of proving does not have any bearing on normalization and equational reasoning.

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

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

Terms. Terms form the purely functional fragment of HTT. They are further 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 Hoare types is $\mathsf{do}\ E$, which encapsulates the effectful computation E and suspends its evaluation. The name of the constructor is chosen to closely resemble the familiar Haskell-style do-notation for writing effectful programs. The corresponding elimination form $x \leftarrow K; E$ activates a suspended computation determined by K. However, this elim form is not a term, but a computation, and is described in more detail below. We also list the intro term $\mathsf{eta}_\alpha\ K$, which is not supposed to be used in programming, but is needed internally during typechecking. The reader can safely ignore this constructor for the time being; its significance will become apparent in Section 3.

The separation into intro and elim terms facilitates bidirectional typechecking (Pierce & Turner, 2000), whereby most of the type information can be omitted from the terms, and inferred automatically. When type information must be supplied explicitly, the constructor $M{:}A$ 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 (Section 4), 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 does not use the constructor $M{:}A$ may not contain beta redexes.

Computations. Computations form the effectful fragment of HTT, and can be described intuitively as semicolon-separated lists of commands, terminated with a return value. The commands are executed in the order in which they appear in the list, and usually bind their result to a variable. For example, the computation $x \Leftarrow c$ executes the command c and binds the result to x. As already described before, $x \leftarrow K$ executes the computation encapsulated in K, thus performing all the side effects that may have been suspended in K. The command $x =_A M$ does not perform any side effects, but is simply a syntactic sugar for the usual let-binding of $M{:}A$ to x.

It is important that variables in HTT are *immutable*, as customary in functional programming languages, but in contrast to the usual treatment of variables in Hoare Logic. The scope of the variables bound in computations (x in the above examples) extends to the right, until the end of the block enclosed by the nearest do.

The return value of a computation is specified by the construct return M. When considered by itself, return M creates a trivial computation that does not perform any side effects, but immediately returns the value M. The computations return M and $x \leftarrow K; E$ correspond to the monadic operations of *unit* and *bind*, respectively.

We illustrate the commands of HTT with the following example. Consider the simple ML-like function $f{:}\mathsf{nat}{\rightarrow}\mathsf{unit}$, defined as $\lambda x{:}\mathsf{nat}.$ if $!y = 0$ then $y := x$ else $(\,)$, where $y{:}\mathsf{ref\ nat}$. The HTT code that implements function f and then immediately applies it to 1 can be written as the following effectful computation.

```
f  =nat→unit  λx. do (t₁  ⇐ !nat y;
                      t₂  ⇐ if_unit (t₁ == z) then
                                 t₃ ⇐ (y :=nat x);
                                 return t₃
                              else return( );
                      return t₂);
t₄  ←  f (s z);
return t₄
```

The syntax of computations is very explicit and rather cumbersome, as every intermediate result must be named with a temporary variable (t_1–t_4 above). Shortly, we will introduce a number of syntactic conventions to reduce the clutter. But first, we describe the semantics of the HTT commands.

The commands $!_\tau\, M$ and $M :=_\tau N$ perform a read from and write into the memory location M, respectively. The index τ is the type of the value being read or written. Unlike most statically typed functional languages, HTT supports *strong updates.* That is, if x is a location storing a nat, then it can be updated with a value of any type, not just another nat. Correspondingly, memory reads from one and the same location may return differently typed values. To ensure safety, the type system must ensure statically, as a precondition, that the contents of the location matches the expected index type τ. On the other hand, the precondition for an update simply requires that the updated location exists (i.e., is allocated), but does not say anything about the type of the old value. The command $\mathsf{alloc}_\tau\ M$ allocates a fresh memory location, initializes it with $M{:}\tau$, and returns the address of the location. Dually, $\mathsf{dealloc}\ M$ frees the location at address M, under the precondition that the location exists. In the conditionals if and case, the index A is the type of the branches. The fixpoint command $\mathsf{fix}\ f(y{:}A){:}B = \mathsf{do}\,E\,\mathsf{in}\,\mathsf{eval}\,f\ M$ first obtains the function f such that $f(y) = \mathsf{do}(E)$, and then evaluates the encapsulated computation $f\ M$. Here f and y may appear in the body E. The types A and B are the argument and result type of f, respectively, and B may depend on the variable y. The type B must be a Hoare type, in order to capture that the recursive definition of f may lead to divergence.

Returning to our example program, we now introduce the following abbreviations. First, we represent natural numbers in their decimal rather than unary form. Second, we omit the variable x in $x \Leftarrow (M :=_\tau N)$ and $x \leftarrow \mathsf{dealloc}\ M$, as x is of unit type. Third, computations of the form $x \Leftarrow c; \mathsf{return}(x)$ are abbreviated simply as c, to avoid the spurious variable x. Similarly, $x \leftarrow K; \mathsf{return}(x)$ is abbreviated as $\mathsf{eval}\ K$. Fourth, we omit the index type in operations, when that type is clear from the context. Under such conventions, the above program may be abbreviated as

$$\begin{array}{l} f = \lambda x.\ \mathsf{do}\ (t_1 \Leftarrow\ !y; \\ \qquad\qquad\qquad \mathsf{if}\ (t_1 == 0)\ \mathsf{then}\ y := x\ \mathsf{else}\ \mathsf{return}(\,)); \\ \mathsf{eval}(f\ 1) \end{array}$$

Types. Types include the primitive types of booleans and natural numbers, unit type 1, dependent functions $\Pi x{:}A.B$, and polymorphic types $\forall\alpha.A$. We write $A \to B$ to abbreviate $\Pi x{:}A.\ B$ when B does not depend on x.

The Hoare 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 *ghost variables* or *logic variables*, and can appear only in the assertions, but not in the programs. Similarly, 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, $\Psi.X.\{P\}x{:}\tau\{Q\}$ is a monotype when τ is a monotype, even if Ψ, P and Q contain polymorphic types. This kind of 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 (Section 4), which is a crucial component of type equality. Therefore, we leave the treatment of impredicative polymorphism to future work.

Note that allowing polymorphic quantification in the assertions of the Hoare types, or in the types of ghost variables, does not change the predicative nature of HTT. The type system will be formulated so that ghost variables and the assertions do not influence the computational behavior or equational properties of effectful computations.

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.

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. Any monotype is allowed, including higher-order function types, and computation types, so that HTT heaps implement higher-order store.

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 $N{:}\tau$.

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

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

In English, $\subset\supset$ is the logical equivalence (and $\supset$ is the logical implication), hid 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 (O'Hearn *et al.*, 2001; Reynolds, 2002; O'Hearn *et al.*, 2004). All of these are relative to the free variable mem, which denotes the current heap fragment of reference.

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

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 $N{:}\tau$. *Separating conjunction* $P * Q$ holds iff P and Q hold of disjoint subheaps of mem. $P \mathrel{-\!\!*} Q$ holds of mem if any extension by a heap satisfying P, produces a heap satisfying Q. $\mathsf{this}(H)$ is true iff mem equals H.

We 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, $\forall X.A$ and $\exists X.A$ stands for iterated quantification over heap variables of the context X. We also abbreviate $\Diamond A$ instead of $\{\top\}x{:}A\{\top\}$.

2.1 Examples

Small and large footprints. In this example, we illustrate the "small footprint" semantics of Hoare types by considering several different specifications that can be ascribed to the HTT allocation primitive. The shortest one is the following.

$$\mathsf{alloc} : \forall\alpha.\ \Pi x{:}\alpha.\ \{\mathsf{emp}\}y{:}\mathsf{nat}\{y \mapsto_\alpha x\}$$

It states that allocation does not touch any existing heap locations (precondition emp), and returns an address y which is initialized with the supplied value $x{:}\alpha$ (postcondition $y \mapsto_\alpha x$). Implicit in the specification is that y must be fresh, because the precondition prohibits alloc from working with existing locations.

We point out that the definitions of the predicates emp and $\mapsto$ require a free variable mem to denote the *current heap*. In other words, the precondition and the postcondition of Hoare types are parametrized with respect to a heap variable mem. In the precondition, mem denotes the beginning heap of the computation, and in the postcondition, mem is the ending heap. The variable mem is bound in both the preconditions and the postconditions of the Hoare type. That is, if we made the scope of variables in Hoare types explicit, the syntax of Hoare types would be

$$\{\mathsf{mem}.\ P\}x{:}A\{\mathsf{mem}.\ Q\}.$$

However, to reduce clutter, we omit the bindings of mem, as they can be assumed implicitly, and simply write $\{P\}x{:}A\{Q\}$.

Returning now to the function alloc, the precondition emp in its specification does not mean that alloc can only work if the global heap is empty. Rather, the small footprint specifications require that the global heap contains a subheap satisfying the precondition. The subheap is then changed by the execution of the effectful computation, so that it satisfies the postcondition. The remaining, unspecified, part of the global heap is guaranteed to remain invariant. In the particular case of alloc, the precondition emp means that alloc can be executed in *any* heap, since any heap trivially contains an empty subheap.

For the sake of illustration, let us consider a slightly less permissive specification for alloc, one which allows execution only in heaps with *at least* one boolean location.

$$\mathsf{alloc}' : \forall\alpha.\ \Pi x{:}\alpha.\ z{:}\mathsf{nat}, v{:}\mathsf{bool}.\ \{z \mapsto_{\mathsf{bool}} v\}y{:}\mathsf{nat}\{y \mapsto_\alpha x * z \mapsto_{\mathsf{bool}} v\}$$

The ghost variables z and v denote the assumed existing location and its contents, respectively, and the specification insists that the contents of z is not changed by the execution. Notice the use of separating conjunction in the postcondition to specify that z and y belong to disjoint heap portions (i.e., are not aliased), and hence y is fresh.

Instead of listing the pre-existing locations in the precondition, and then repeating them in the postcondition, as in alloc' above, we can simply name the heap encountered before allocation by using a ghost heap variable (say h), and then explicitly specify in the postcondition that h is not changed.

$$\mathsf{alloc}'' : \forall\alpha.\ \Pi x{:}\alpha.\ h{:}\mathsf{heap}.\ \{\mathsf{this}(h)\}y{:}\mathsf{nat}\{(y \mapsto_\alpha x) * \mathsf{this}(h)\}$$

Thus heap variables allow us to express some of the invariance that one may express in higher-order separation logic (Biering *et al.*, 2005).

For comparison, let us now consider the specification of alloc in the classical style (Cartwright & Oppen, 1978), which, by contrast, we can call *large footprint* specifications. Since alloc can run in any initial heap h, the precondition is trivial. The ending heap is obtained as $\mathsf{upd}_\alpha(h, y, x)$, but we also know that y is fresh, that is $y \notin h$. Using again ghost variables to name the initial heap, we can write this specification as

$$\mathsf{alloc}''' : \forall\alpha.\ \Pi x{:}\alpha.\ h{:}\mathsf{heap}.\ \{\mathsf{this}(h)\}y{:}\mathsf{nat}\{\mathsf{this}(\mathsf{upd}_\alpha(h, y, x)) \wedge y \notin h\}.$$

By explicitly naming various heap fragments with ghost variables, HTT can freely switch between the small and large footprint specifications. We believe this to be a very desirable property. For example, it is obvious that the types for alloc, alloc'' and alloc''' are all equivalent, but the small footprint specification as in alloc is the most convenient for programming, when we want to express non-aliasing and freshness of locations. On the other hand, large footprint specification as in alloc''' may be more parsimonious in cases where aliasing is allowed, as we will illustrate in subsequent examples. Finally, it will turn out that the intermediate form of specification as in alloc'' is the easiest one to connect to the assertion logic of HTT, and we will adopt this intermediate form to define the semantics of Hoare types. The small footprint specification can then be viewed as a simple syntactic variant, as we only need to introduce a fresh ghost variable to name the untouched part of the heap.

Diverging computation. In HTT, the term language is pure. Non-termination is an effect, and is relegated to the fragment of impure computations. Hence, any recursive program in HTT will have a monadic type, which will prevent recursion from unrolling during typechecking. We can write a diverging computation of an arbitrary Hoare type $\{P\}x{:}A\{Q\}$ as follows.

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

diverge is a suspended computation which, when forced, first sets up a recursive function $f(y:1) = \mathsf{do}\ (\mathsf{eval}\ (f\ y))$. The function is applied to ( ) to obtain another suspended computation $\mathsf{do}\ (\mathsf{eval}\ f\ \ (\,))$, which is immediately forced by eval, to trigger another application to ( ), another suspended computation, another forcing, and so on, ad infinitum.

Polymorphism and higher-order functions. 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 $\alpha\,\mathsf{ref} \times \alpha\,\mathsf{ref} \rightarrow 1$. This type is an underspecification, of course, as it does not describe what the function does. In HTT, we can be more precise. Furthermore, we can use strong updates to swap locations pointing to values of different types. One possible definition of swap is presented below.

$$\begin{array}{lcl} \mathsf{swap} & : & \forall\alpha.\ \forall\beta.\ \Pi x{:}\mathsf{nat}.\ \Pi y{:}\mathsf{nat}. \\ & & \quad m{:}\alpha, n{:}\beta.\ \{x \mapsto_\alpha m * y \mapsto_\beta n\}r{:}1\{x \mapsto_\beta n * y \mapsto_\alpha m\} \\ & = & \Lambda\alpha.\ \Lambda\beta.\ \lambda x.\ \lambda y.\ \mathsf{do}\ (t_1 \Leftarrow !_\alpha\, x;\ t_2 \Leftarrow !_\beta\, y; \\ & & \qquad\qquad y :=_\alpha t_1;\ x :=_\beta t_2; \\ & & \qquad\qquad \mathsf{return}(\,)) \end{array}$$

The function takes two monotypes α and β, two locations x and y and produces a computation which reads 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 $m{:}\alpha$ and $n{:}\beta$ respectively, for some ghost variables m and n. The locations must not be aliased, due to the use of spatial conjunction 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 (Smith *et al.*, 2000) and ATS with stateful views (Zhu & Xi, 2005). However, in HTT, like in Separation Logic, we can include the aliasing case as well.

One possible small-footprint specification which covers both aliasing and non-aliasing is as follows.

$$\begin{array}{lcl} \mathsf{swap}' & : & \forall\alpha.\ \forall\beta.\ \Pi x{:}\mathsf{nat}.\ \Pi y{:}\mathsf{nat}. \\ & & \quad m{:}\alpha, n{:}\beta.\ \ \{(x \mapsto_\alpha m * y \mapsto_\beta n) \vee (x \mapsto_\alpha m \wedge y \mapsto_\beta n)\}\ r{:}1 \\ & & \qquad\qquad\quad \{(x \mapsto_\beta n * y \mapsto_\alpha m) \vee (x \mapsto_\beta m \wedge y \mapsto_\alpha n)\} \end{array}$$

The second disjuncts in the pre- and post-condition of the above specification, use $\wedge$ instead of spatial conjunction, 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 switch to a large-footprint style and admit an arbitrary heap in the assertions, but then explicitly state the invariance of the heap fragment not containing x and y.

$$\begin{array}{lll}\mathsf{swap}'' & : & \forall\alpha.\ \forall\beta.\ \Pi x{:}\mathsf{nat}.\ \Pi y{:}\mathsf{nat}. \\ & & \quad m{:}\alpha, n{:}\beta, h{:}\mathsf{heap}.\ \{(x \hookrightarrow_\alpha m) \wedge (y \hookrightarrow_\beta n) \wedge \mathsf{this}(h)\}\ r{:}1 \\ & & \qquad \{\mathsf{this}(\mathsf{upd}_\beta(\mathsf{upd}_\alpha(h, y, m), x, n))\}\end{array}$$

As mentioned before, the large footprint style may be less verbose in the case of specifications that admit aliasing. For example, expressing that swapping the same locations twice in a row does not change anything can be achieved as follows.

$$\begin{array}{lll}\mathsf{identity} & : & \forall\alpha.\ \forall\beta.\ \Pi x{:}\mathsf{nat}.\ \Pi y{:}\mathsf{nat}. \\ & & \quad h{:}\mathsf{heap}.\ \{x \hookrightarrow_\alpha - \wedge y \hookrightarrow_\beta - \wedge \mathsf{this}(h)\} r{:}1 \{\mathsf{this}(h)\} \\ & = & \Lambda\alpha.\ \Lambda\beta.\ \lambda x.\ \lambda y.\ \mathsf{do}\ (t_1 \leftarrow \mathsf{swap}''\ \alpha\ \beta\ x\ y; \\ & & \qquad t_2 \leftarrow \mathsf{swap}''\ \beta\ \alpha\ x\ y; \\ & & \qquad \mathsf{return}(\,))\end{array}$$

This function uses swap'' to generate a computation for swapping x and y, and then activates it twice using the monadic bind to the temporary variables t_1 and t_2. A small footprint specification would have to explicitly repeat in the postcondition that the contents of x and y remains unchanged, whereas the large footprint specification can simply postulate the equality with the initial heap.

We further note that $\mathsf{identity}$ is completely parametric in swap'', in the sense that any other function with the same specification as swap'' could have been used instead. Thus, it is possible to *abstract* the variable swap'' from $\mathsf{identity}$, and obtain a function whose type combines Hoare specifications in a higher-order way.

As a further example of scoping in Hoare types, as well as the use of higher-order abstraction over computations, let us consider the following specification that may be ascribed to a standard looping constructor until. Here until is parametrized by the loop invariant I, which is an assertion that depends on the current memory mem in the pre- and post-conditions of the Hoare types. until takes as argument the loop body which includes the computation of the loop guard as well.

$$\begin{array}{lll}\mathsf{until}_I & : & \{I(\mathsf{mem})\}b{:}\mathsf{bool}\{I(\mathsf{mem})\} \to \{I(\mathsf{mem})\}b{:}\mathsf{bool}\{I(\mathsf{mem}) \wedge b = \mathsf{true}\} \\ & = & \lambda e.\ \mathsf{do}(\mathsf{fix}\, f(r:1) : \{I(\mathsf{mem})\}b{:}\mathsf{bool}\{I(\mathsf{mem}) \wedge b = \mathsf{true}\} = \\ & & \qquad \mathsf{do}(b \leftarrow e; \mathsf{if}\ b\ \mathsf{then\ true\ else\ eval}\ f(\,)) \\ & & \quad \mathsf{in\ eval}\ f(\,))\end{array}$$

Because currently HTT lacks the ability to polymorphically abstract over predicates, like the loop invariant I above, we cannot create a generic until constructor. Instead, a separate copy of until must be created for each specific loop invariant. A similar requirement would also appear in the specification of the usual polymorphic functionals like map and fold over inductively defined types.

Thus, it is important to extend HTT with abstraction over predicates. Such extension would not only allow for generic programs like until above, but would also increase the expressive power of the assertion logic to higher order. We forego any further discussion of this research direction, but refer to (Nanevski *et al.*, 2007) for a description of some initial steps towards extending HTT to the full power of the Extended Calculus of Constructions.

3 Type system

Typechecking judgments in HTT (as in any other type theory) require testing if two terms are definitionally equal. In HTT, the tests for definitional equality involve *normalization*. Two terms are reduced by normalization into their respective *canonical forms* and are deemed equal only if the canonical forms are syntactically the same, modulo α-conversion. The formulation of HTT is somewhat unusual, however, as normalization is undertaken simultaneously with type checking. That is, each typing judgment not only decides if some expression has a required type, but also computes as output the canonical form of the expression. That way, whenever two well-typed expressions must be checked for definitional equality, their canonical forms are readily available for syntactic comparison.

In the following section, we introduce the equations and operations that are used in the normalization algorithm of HTT; a more thorough discussion of this process will be given in Section 4. We then proceed with the definition of HTT typing rules.

3.1 Equational reasoning

The most important equations in the definitional equality of HTT are the beta and eta equalities associated with each type constructor. We orient the equations, to emphasize their use as rewriting rules in the normalization algorithm. As customary, beta equality is used as a reduction, and eta equality as an expansion.

For example, the function type $\Pi x{:}A.\ B$ gives rise to the following familiar beta reduction and eta expansion.

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

Here, of course, the notation $[T/x]M$ denotes a capture-avoiding substitution of the expression T for the variable x in the expression M. In the above equations, the participating terms are decorated with type annotations, because otherwise the results may not be well-formed with respect to the HTT syntax from Section 2.

Equations associated with the type $\forall\alpha.\ A$ are also standard.

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

where $\mathsf{FTV}(K)$ denotes the free monotype variables of K.

In the case of the unit type, we do not have any beta reduction (as there are no elimination constructors associated with this type), but only one eta expansion.

$$K \quad \longrightarrow_{\eta} \quad (\,)$$

The equations for the Hoare type require an auxiliary operation of *monadic substitution*, $\langle E/x{:}A\rangle F$, which sequentially composes the computations E and F. The result is a computation which should, intuitively, evaluate as E *followed by* F, where the free variable $x{:}A$ in F is bound to the value of E. Monadic substitution is defined by induction on the structure of E, as follows.

$$\begin{array}{rcll}
\langle \mathsf{return}\ M/x{:}A\rangle F & = & [M{:}A/x]F & \\
\langle y \leftarrow K; E/x{:}A\rangle F & = & y \leftarrow K; \langle E/x{:}A\rangle F & \text{choosing } y \notin \mathsf{FV}(F) \\
\langle y \Leftarrow c; E/x{:}A\rangle F & = & y \Leftarrow c; \langle E/x{:}A\rangle F & \text{choosing } y \notin \mathsf{FV}(F) \\
\langle y =_B M; E/x{:}A\rangle F & = & y =_B M; \langle E/x{:}A\rangle F & \text{choosing } y \notin \mathsf{FV}(F)
\end{array}$$

Now the equations associated with Hoare types can be defined as follows.

$$\begin{array}{rcl}
x \leftarrow (\mathsf{do}\ E : \{P\}x{:}A\{Q\}); F & \longrightarrow_{\beta} & \langle E/x{:}A\rangle F \\
K & \longrightarrow_{\eta} & \mathsf{do}\ (x \leftarrow K; \mathsf{return}\ x)
\end{array}$$

The monadic substitution is modeled directly after the operation introduced by Pfenning and Davies (2001), who also show that the beta and eta equations above are equivalent to the unit and associativity laws for a generic monad (Moggi, 1991).

The normalization algorithm of HTT does not use the described eta expansions in an arbitrary way, but relies on a very specific strategy. To capture this strategy, we define an auxiliary function expand_A, which iterates over the type A and expands the given argument (elim or intro term), according to each encountered type constructor.

$$\begin{array}{lcll}
\mathsf{expand}_a(K) & = & K & \text{if } a \text{ is } \mathsf{nat} \text{ or } \mathsf{bool} \\
\mathsf{expand}_\alpha(K) & = & \mathsf{eta}_\alpha\ K & \text{if } \alpha \text{ is a monotype variable} \\
\mathsf{expand}_1(K) & = & (\,) & \\
\mathsf{expand}_{\forall\alpha.\,A}(K) & = & \Lambda\alpha.\ \mathsf{expand}_A(K\ \alpha) & \text{choosing } \alpha \notin \mathsf{FTV}(K) \\
\mathsf{expand}_{\Pi x{:}A_1.\,A_2}(K) & = & \lambda x.\ \mathsf{expand}_{A_2}(K\ M) & \text{where } M = \mathsf{expand}_{A_1}(x) \\
 & & & \text{choosing } x \notin \mathsf{FV}(K) \\
\mathsf{expand}_{\{P\}x{:}A\{Q\}}(K) & = & \mathsf{do}\ (x \leftarrow K; E) & \text{where } E = \mathsf{return}(\mathsf{expand}_A(x)) \\
\mathsf{expand}_A(N) & = & N & \text{if } N \text{ is not elim}
\end{array}$$

The definition of expand exposes the significance of the constructor $\mathsf{eta}_\alpha\ K$. This constructor records that, once the type variable α is instantiated with some concrete type, then K should be expanded, on-the-fly, with respect to this type.

Beta reductions too are used in the normalization algorithm according to a very specific strategy, implemented via a set of auxiliary functions called *hereditary substitutions* (Watkins *et al.*, 2004) that operate on canonical terms only. 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, which is immediately reduced by initiating another hereditary substitution and so on.

The definition and properties of hereditary substitutions are discussed in more detail in Section 4. Here we only note that hereditary substitutions 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 as termination metric for the substitution. We also need a monadic hereditary substitution $\langle E/x\rangle_A(-)$, and a monotype hereditary substitution $[\tau/\alpha]^*(-)$. The later performs an on-the-fly eta expansion with respect to τ of any subterms of the form $\mathsf{eta}_\alpha\ K$ in the argument.

We mention two more capture-avoiding substitutions, $[H/h]G$ and $[H/h]P$, which substitute the heap H for the variable h in the *heap* G and *assertion* P, respectively. We will never substitute heaps into other kinds of expressions, so we do not define similar operations for other syntactic domains. The substitutions simply commute with all the constructors, leaving intact the subexpressions which are not heaps or assertions, as illustrated in the definitional clause below (which is typical).

$$[H/h](\mathsf{upd}_\tau(G, M, N)) \quad = \quad \mathsf{upd}_\tau([H/h]G, M, N)$$

Not substituting into τ, M and N is justified because HTT types, terms and computations are *not* allowed to contain free heap variables.

There are several more reductions that factor into definitional equality. For example, $\mathsf{s}\ M + N$ is not canonical, as it can be simplified into $\mathsf{s}\ (M + N)$. The later is simpler, because it makes more of the structure apparent (e.g., just by looking at the head constructor of $\mathsf{s}\ (M + N)$, we know that the expression must be nonzero). Similarly, addition $\mathsf{z} + N$ reduces to N, multiplication $\mathsf{s}\ M \times N$ reduces to $N + (M \times N)$, and multiplication $\mathsf{z} \times N$ reduces to z. These reductions are required in order for the normalization to agree with the evaluation of nat expressions, so that the actual numerals are the only closed canonical forms of type nat.

We close this section with the comment that the rules described here certainly do not constitute a reasoning system that is complete in any sense. For example, algebraic laws like commutativity or associativity of addition, or reasoning principles like induction, cannot be inferred. We will make such reasoning principles available in the *propositional equality* defined in the following section.

3.2 Typing rules

The type system of HTT consists of the following judgments.

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

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 (Ψ' is canonical for Ψ, P'

is canonical for P, etc). When checking assertion contexts ($\Gamma\ \mathsf{pctx}$), Γ is required to be canonical, so there is no need to return the output.

The judgments on the left are the main 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 (Pierce & Turner, 2000), which we here employ for both terms and computations.

For example, $\Delta \vdash K \Rightarrow A\,[N']$ takes an elim form K 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 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 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 and outputs the canonical form E', if Q is a postcondition (not necessarily the strongest) for E with respect to P, and A is the type of the return value of E.

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. Here Δ is a variable context, X is a heap context, and Γ_1, Γ_2 are sets of assertions. As usual, the sequent is valid 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 need two auxiliary functions to compute canonical forms of application and type specialization. The functions $\mathsf{apply}_A(M, N)$ and $\mathsf{spec}(M, \tau)$ reduce the applications $M\ N$ and $M\ \tau$, if M is a function or type abstraction, respectively. Here, M, N and τ are assumed to be canonical.

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

The typing rules now formalize the intuition that intro terms can be 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 can be read from the context, the type of the whole application can always be inferred by instantiating the type of x with T_i.

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

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

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

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

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

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

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

For example, $\Pi\mathsf{I}$ checks that term $\lambda x.\ M$ has the given function type, and if so, returns the canonical form $\lambda x.\ M'$. In $\Pi\mathsf{E}$ 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 N' actually be a lambda abstraction. 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 syntactically equal. The canonical form synthesized from K in the premise, may be an elim term (because it is generated by a judgment for elim terms), but we need to use it in the conclusion as an intro term. The switch from an elim term to the equivalent intro term 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 $x{:}\mathsf{nat}{\rightarrow}\mathsf{nat} \vdash x \Leftarrow \mathsf{nat}{\rightarrow}\mathsf{nat}\,[\lambda y.\ x\ y]$.

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 use the constructor eta to remember that expansion must be done as soon as α is instantiated with a concrete monotype. The constructor eta is applicable only to canonical forms; notice how its typing rule insists that the argument K equals its own canonical form. Thus, eta is never used in the source programs, but is only required internally, for equational reasoning.

Computations. The judgment $\Delta; P \vdash E \Rightarrow x{:}A.\ Q\,[E']$ generates the strongest postcondition Q for the program E with respect to the precondition P (although approaches based on other kinds of predicate transformers, like the weakest preconditions, are possible). However, unlike in the Hoare type, where pre- and postconditions are unary relations over the heap mem, here the assertions P and Q are *binary relations on heaps*, and the typing rule for introduction of Hoare types will mediate the switch from unary to binary relations. Since P and Q are binary relations, we make them depend on two free heap variables: init, which denotes a specific heap in the past of the computation, and mem, which denotes the current heap. The computation E describes how the heap mem from the precondition is modified into the heap mem in the postcondition, one effectful step at a time, while init is a fixed heap of reference, shared by both P and Q. In other words, P describes how mem relates to init before the execution of E, and Q describes how mem relates to init after the execution of E.

Alternatively, the computation E may be viewed semantically as a relation between the input and output heaps. Then the typing judgment simply serves to translate the computational syntax of E into the corresponding binary relation (here, the postcondition Q). Intuitively, P is a relation that the translation starts with, and Q is the relation that the translation ends with, thus capturing the semantics of E. In addition, P has to be strong enough to guarantee that the execution of E will never get stuck. Translating programs into relations on heaps is a well-known approach to formulating Hoare Logic (Greif & Meyer, 1979), and we adapt it here to a type-based system.

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

$$R_1 \multimap R_2 \quad = \quad \forall h{:}\mathsf{heap}.\ ([\mathsf{init}/\mathsf{mem}](R_1 * \mathsf{this}(h))) \supset R_2 * \mathsf{this}(h)$$

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. For example, $R_1 \multimap R_2$ captures the transformation incurred on the heap by the execution of an unspecified computation with type $\{R_1\}x{:}A\{R_2\}$. Notice how the definition of $\multimap$ relies on naming the heap h by means of universal quantification in order to state its invariance. This corresponds to the equivalence of types $\{R_1\}x{:}A\{R_2\}$ and $h{:}\mathsf{heap}.\{R_1 * \mathsf{this}(h)\}x{:}A\{R_2 * \mathsf{this}(h)\}$ that we commented on in Section 2.1.

Now consider a suspended computation $\mathsf{do}\ E : \Psi.X.\{R_1\}x{:}A\{R_2\}$. 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 $x \leftarrow M; E$, if M is a do-suspended computation. Here A, M and E are assumed canonical.

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

We start with the rules for the general monadic operations, and then proceed with the individual effectful commands.

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

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

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

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

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 semantics 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 $K \Rightarrow \Psi.X.\{R_1\}x{:}A\{R_2\}$ can be sequentially composed with another computation E. The composition is meaningful if the following are satisfied. First, the assertion logic must establish that P ensures that the current heap contains a fragment satisfying 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.

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

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

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

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

$$\frac{\Delta \vdash A \Leftarrow \mathsf{type}\,[A'] \qquad \Delta \vdash M \Leftarrow A'\,[M'] \qquad \Delta; P \vdash [M{:}A/x]E \Rightarrow y{:}B.\ Q\,[E']}{\Delta; P \vdash x =_{A} M; E \Rightarrow y{:}B.\ Q\,[E']}$$

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.

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 emp. The side condition is the sequent $P \Longrightarrow M' \hookrightarrow -$ showing that M' was allocated.

Typechecking the command $x =_A M; E$ reduces to typechecking the substitution $[M/x]E$, reflecting that assignment is just a syntactic sugar for substitution.

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

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

A similar rule is given for case.

Finally, we present the rule for recursion. The body $\mathsf{do}\ E$ of the recursive function may depend on the function itself (variable f) and one argument (variable x). We also require an initial value M to which the recursive function is immediately applied. 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.

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

Before M can be applied to the recursive function, and the obtained computation executed, we need to check that the main precondition P implies $\exists \Psi.X.(R_1 * \top)$, 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 first-order classical multi-sorted logic with equality and universal and existential polymorphic quantification over monotypes. The sorts include bools, nats, functions and type functions with extensionality, effectful computations and heaps. The calculus includes the rules of cut, initial sequents, structural rules of weakening, contraction and exchange, and left and right rules for each propositional connective; all of these are standard (Girard *et al.*, 1989) so we do not present them here.

Definitional equality is embedded into propositional equality by postulating the following rules.

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

$$\frac{\Delta, x{:}A; X \vdash P \Leftarrow \mathsf{prop}\,[P] \qquad \Delta; X; \Gamma_1, \mathsf{id}_A(M, N) \Longrightarrow [M/x]^p_A(P), [N/x]^p_A(P), \Gamma_2}{\Delta; X; \Gamma_1, \mathsf{id}_A(M, N) \Longrightarrow [M/x]^p_A(P), \Gamma_2}$$

The first rule postulates that propositional equality is reflexive. In combination with the requirement that all terms appearing in sequents are *in canonical form*, this rule in effect embeds definitional equality as a subrelation of propositional equality. The second rule axiomatizes that equal terms can be substituted in an arbitrary context P.

The last equality rule does not admit extensionality of functions. The terms M and N must depend only on the variables in Δ and X, while functions require an additional variable. Hence, we introduce a separate rule for function extensionality, and similarly, a separate rule for type abstraction.

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

$$\frac{\Delta, \alpha; X; \Gamma_1 \Longrightarrow \mathsf{id}_B(M, N), \Gamma_2}{\Delta; X; \Gamma_1 \Longrightarrow \mathsf{id}_{\forall \alpha.\, B}(\Lambda \alpha.\, M, \Lambda \alpha.\, N), \Gamma_2}$$

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

Heaps are axiomatized as partial functions from nats to values as follows.

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

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

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

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

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 (McCarthy, 1962), 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. Stating the full functionality of heaps requires a new assertion for *heterogeneous equality* (aka. John Major equality, as introduced by McBride (1999)), which allows equating terms at different underlying types. We leave this extension for future work.

Finally, the rules for the primitive types include the standard Peano axioms for natural numbers, including the induction principle, and a similar set of rules for booleans.

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

$$\frac{\begin{array}{c}\Delta, x{:}\mathsf{nat}; X \vdash P \Leftarrow \mathsf{prop}\,[P] \\ \Delta \vdash M \Leftarrow \mathsf{nat}\,[M] \qquad \Delta, x{:}\mathsf{nat}; X; \Gamma_1, P \Longrightarrow [\mathsf{s}\ x/x]^p_{\mathsf{nat}}(P), \Gamma_2\end{array}}{\Delta; X; \Gamma_1, [z/x]^p_{\mathsf{nat}}(P) \Longrightarrow [M/x]^p_{\mathsf{nat}}(P), \Gamma_2}$$

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

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

Peano axioms usually include equations on primitive operations like $+$ and $\times$. Such equations are not required here, as they are already incorporated into definitional equality. Similarly, we do not postulate any specific equations over effectful computations. Currently, HTT only supports the generic monadic unit and associativity laws (Moggi, 1991), and these too are already a part of definitional equality.

Example. Here we 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 in Figure 1, and annotate it with assertions (preceded by "$--$") 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) $\mathsf{sum}(r, n) = \mathsf{id}_{\mathsf{nat}}(2 \times r, n \times (n+1))$ denoting that $r = 1 + \cdots + n$; (2) $I = i \leq n \wedge \exists t{:}\mathsf{nat}.\ a \mapsto_{\mathsf{nat}} t \wedge \mathsf{sum}(t, i)$ will be the loop invariant during the summation; (3) $Q = a \mapsto_{\mathsf{nat}} - \wedge \mathsf{sum}(x, n)$ asserts what holds upon the exit from the loop. 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 0, as specified in $\mathsf{eval}\,f\,0$, and the loop terminates when i reaches n.

The following sequents are generated during typechecking, and they constitute the verification conditions that should be discharged in order to validate the program: (1) $P_2 \Longrightarrow a \hookrightarrow_{\mathsf{nat}} -$, so that a can be looked up, (2) $P_5 \Longrightarrow a \hookrightarrow -$ so that a can be updated, (3) $P_6 \Longrightarrow [i+1/i]I * \top$, so that the computation obtained from $f(i+1)$ can be executed, (4) $P_8 \wedge \mathsf{id}_{\mathsf{nat}}(x, t) \Longrightarrow I \multimap Q$, so that the fixpoint satisfies

```
sumfunc  :  $\Pi n{:}\mathsf{nat}.\ \{\mathsf{emp}\}r{:}\mathsf{nat}\{\mathsf{emp} \wedge \mathsf{sum}(r, n)\}$
         =  $\lambda n.$ do (−− $P_0 : \{\mathsf{this}(\mathsf{init})\}$
                  $a = \mathsf{alloc}_{\mathsf{nat}}(0);$
                  −− $P_1 : \{\mathsf{this}(\mathsf{init}) * (a \mapsto_{\mathsf{nat}} 0)\}$
                  $x = \mathsf{fix}\ f(i : \mathsf{nat}) : \{I\}x{:}\mathsf{nat}\{Q\} =$
                        do (−− $P_2 : \{\mathsf{this}(\mathsf{init}) \wedge (I * \top)\}$
                            $s = !_{\mathsf{nat}}\, a;$
                            −− $P_3 : \{P_2 \wedge a \hookrightarrow_{\mathsf{nat}} s\}$
                            $t = \mathsf{if}_{\mathsf{nat}}(i == n)$ then
                                    −− $P_4 : \{P_3 \wedge \mathsf{id}_{\mathsf{nat}}(i, n)\}$
                                    return($s$)
                                else
                                    −− $P_5 : \{P_3 \wedge \neg\mathsf{id}_{\mathsf{nat}}(i, n)\}$
                                    $a :=_{\mathsf{nat}}\ s + i + 1;$
                                    −− $P_6 : \{P_5 \circ (a \mapsto_{\mathsf{nat}} - \multimap a \mapsto_{\mathsf{nat}} s + i + 1)\}$
                                    $x \leftarrow f\ (i + 1);$
                                    −− $P_7 : \{P_6 \circ ([i + 1/i]I \multimap Q)\}$
                                    return($x$)
                            −− $P_8 : \{(P_4 \wedge \mathsf{id}_{\mathsf{nat}}(t, s)) \vee (\exists x{:}\mathsf{nat}.P_7 \wedge \mathsf{id}_{\mathsf{nat}}(t, x))\}$
                            return($t$))
                    in eval $f$ 0;
                  −− $P_9 : \{P_1 \circ ([0/i]I \multimap Q)\}$
                  dealloc($a$);
                  −− $P_{10} : \{P_9 \circ (a \mapsto_{\mathsf{nat}} - \multimap \mathsf{emp})\}$
                  return($x$))
```

Fig. 1. Annotated code for summation. Lines preceded by "−−" describe what is true at the respective program points.

the prescribed postcondition, (5) $P_9 \Longrightarrow a \hookrightarrow -$ so that a can be deallocated, and (6) $P_{10} \wedge \mathsf{id}_{\mathsf{nat}}(r, x) \Longrightarrow \mathsf{emp} \multimap (\mathsf{emp} \wedge \mathsf{sum}(r, n))$, so that sumfunc has the required postcondition. It is not too hard to see that all these sequents are valid.

4 Hereditary substitutions

An HTT 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.

The main insight, due to Watkins et al. (2004), is that conversion to canonical forms (i.e., normalization) can be defined on possibly ill-typed terms, and can be shown to terminate using a simple syntactic argument. The current section is a self-contained presentation of these ideas, augmented with a treatment of predicative polymorphism (which is not a difficult extension).

At the center of the development are *hereditary substitutions*, which are capture-avoiding substitutions defined only on canonical terms. Because of the restriction to canonical terms, hereditary substitutions cannot produce intermediate or final results which contain beta redexes. Thus, whenever an ordinary substitution would create a beta redex, hereditary substitution must immediately reduce it. This reduction may create another redex, which must be reduced as well, and so on. Hereditary substitutions may therefore be viewed as implementing a very specific normalization strategy, which lends itself to a simple proof of termination.

In the current paper, the definitional equality of HTT terms does not depend on the full HTT type, but only on its dependency-free version. For example, two terms that are equal at some Hoare type are equal at any other Hoare type that they belong to. Thus, when computing with canonical forms, we can ignore the assertions from the Hoare types. With this in mind, given an HTT type A, we define the *shape* A^- to be the simple type obtained by erasing the dependencies.

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

We impose an ordering on shapes and write $S_1 \leq S_2$ and $S_1 < S_2$, if S_1 is a subexpression of S_2 (proper subexpression in the second case). Here we consider that proper subexpressions of a quantified type also include all the substitution instances obtained by replacing the bound type variable with a *simple* monotype. This is clearly a well-founded ordering, as instantiating a type quantifier with a simple monotype decreases the overall number of quantifiers.

We define following hereditary substitutions: (1) $[M/x]^*_S(-)$ substitute the canonical term M for x in the argument. Here $* \in \{k, m, e, a, p, h\}$ ranges over elim terms, intro terms, computations, types, assertions and heaps, respectively, and determines the domain of the argument of the substitution. The index S is the putative shape of the type of M, and will serve as the termination metric for the substitution; (2) $\langle E/x\rangle_S(F)$ is the hereditary version of the *monadic substitution*, and (3) $[\tau/\alpha]^*(-)$ is the hereditary *type* substitution. To reduce clutter, we will frequently write $[M/x]^*_A(-)$ and $\langle E/x\rangle_A(F)$, instead of $[M/x]^*_{A^-}(-)$ and $\langle E/x\rangle_{A^-}(F)$, correspondingly.

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

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

We next present the characteristic cases of the definitions of hereditary substitutions. The substitution into elim terms may return either another elim term K, or an intro term N. In the later case, N is annotated with the shape S of its putative type.

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

Notice that the substitution into $K\ N$ and $K\ \tau$ may fail to be defined depending on what is returned as a result of substituting into K. For example, a failure will appear if $[M/x]^k_S(K)$ returns an intro term which is not a lambda abstraction, or if the returned shape is not a function type. We also note that the definition does not require a case for substitution into elim terms $N{:}A$, because this kind of terms cannot appear in a canonical form.

Next we present several cases from the hereditary substitution into intro terms.

$$\begin{array}{llll}
[M/x]^m_S(K) & = & K' & \text{if } [M/x]^k_S(K) = K' \\
[M/x]^m_S(K) & = & N' & \text{if } [M/x]^k_S(K) = N' :: S' \\
[M/x]^m_S(\mathsf{eta}_\alpha\ K) & = & \mathsf{eta}_\alpha\ K' & \text{if } [M/x]^k_S(K) = K' \\
[M/x]^m_S(\mathsf{eta}_\alpha\ K) & = & \mathsf{eta}_\alpha\ K' & \text{if } [M/x]^k_S(K) = \mathsf{eta}_\alpha\ K' :: \alpha \\
[M/x]^m_S(()) & = & () & \\
[M/x]^m_S(\lambda y.\ N) & = & \lambda y.\ N' & \text{where } [M/x]^m_S(N) = N' \\
 & & & \text{choosing } y \notin \mathsf{FV}(M) \text{ and } y \neq x \\
[M/x]^m_S(\Lambda\alpha.\ N) & = & \Lambda\alpha.\ N' & \text{where } [M/x]^m_S(N) = N' \\
 & & & \text{choosing } \alpha \notin \mathsf{FTV}(M) \text{ and } \alpha \neq x \\
[M/x]^m_S(\mathsf{do}\ E) & = & \mathsf{do}\ E' & \text{if } [M/x]^e_S(E) = E'
\end{array}$$

We omit the clauses for substitution into primitive operations, as they are not particularly interesting. In fact, the only interesting case here is substituting into $\mathsf{eta}_\alpha\ K$, which fails if substituting into the subterm K does not return $\mathsf{eta}_\alpha\ K'$ indexed by the same type variable α. Any other intro term would restrict which types could be substituted for α, thus violating the parametricity of type abstraction.

The characteristic cases of hereditary substitution into computations follow.

$$\begin{array}{llll}
[M/x]^e_S(\mathsf{return}\, N) & = & \mathsf{return}\, N' & \text{if } [M/x]^m_S(N) = N' \\
[M/x]^e_S(y \leftarrow K; E) & = & y \leftarrow K'; E' & \text{if } [M/x]^k_S(K) = K' \text{ and } [M/x]^e_S(E) = E' \\
 & & & \text{choosing } y \notin \mathsf{FV}(M) \text{ and } y \neq x \\
[M/x]^e_S(y \leftarrow K; E) & = & F' & \text{if } [M/x]^k_S(K) = \mathsf{do}\ F :: \diamond S_1 \\
 & & & \text{and } [M/x]^e_S(E) = E' \text{ and } F' = \langle F/y \rangle_{S_1}(E') \\
 & & & \text{choosing } y \notin \mathsf{FV}(M) \text{ and } y \neq x
\end{array}$$

We omit the clauses for primitive computations as they are straightforward. Also, we do not require a clause for substitution into $y =_A N; E$, because this computation is not canonical (indeed it reduces to $[N/y]^e_A(E)$). The only interesting case arises when the substitution into the branch K of $y \leftarrow K; E$ returns a do-suspended computation. That creates a redex which is immediately reduced by invoking a monadic hereditary substitution.

Hereditary monadic substitution is a simple adaptation of the ordinary monadic substitution defined in Section 3.1.

$$\begin{array}{rcll} \langle \mathsf{return}\ M/x\rangle_A(F) & = & F' & \text{if } [M/x]^e_A(F) = F' \\ \langle y \leftarrow K; E/x\rangle_A(F) & = & y \leftarrow K; F' & \text{if } \langle E/x\rangle_A(F) = F', \text{ choosing } y \notin \mathsf{FV}(F) \\ \langle y \Leftarrow c; E/x\rangle_A(F) & = & y \Leftarrow c; F' & \text{if } \langle E/x\rangle_A(F) = F', \text{ choosing } y \notin \mathsf{FV}(F) \\ \langle E/x\rangle_A(F) & & \text{fails} & \text{otherwise} \end{array}$$

Hereditary (mono)type substitutions $[\tau/\alpha]^*(-)$ in most cases simply commute with the constructors, and the only interesting case is:

$$[\tau/\alpha]^m(\mathsf{eta}_\alpha\ K) \quad = \quad \mathsf{expand}_\tau(K') \quad \text{if } [\tau/\alpha]^k(K) = K'$$

which expands out the argument expression with respect to τ. Notice that the result must itself be a canonical intro term, as expansion cannot create new redexes (please see the definition of expand in Section 3.1).

We omit the hereditary substitutions into types, propositions and heaps, as they simply commute with all the connectives.

Theorem 1 (*Termination of hereditary substitutions*)

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

Proof

The first part is by induction on K. The second part is trivial in the case of type substitutions, as type substitutions cannot create new redexes, and are hence terminating. In the case of $[M/x]^*_S$ and $\langle E/x\rangle_S$, we proceed by nested induction, first on the index shape S (under the ordering $\leq$) , and second on the structure of the argument we apply the substitution to. In each case of the definition, we either decrease S, or failing that, we apply the function to strict subexpressions of the input. Since $\leq$ is well-founded, this process must terminate.

Note that without the restriction to predicative polymorphism, types could actually grow after a substitution into the elim form $K\ \tau$, and hereditary substitutions may fail to terminate if applied to ill-typed terms. Hence our restriction to predicative polymorphism insists that τ must be a monotype.

In the future work, we plan to extend HTT with impredicative polymorphism. The extension will require a significantly more involved proof of termination, probably based on logical relations, unlike the present case where a simple syntactic argument suffices. □

5 Properties

In this section we present the most characteristic properties of HTT, leading up to the substitution principles.

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.

Proof
The typing judgments of HTT are syntax directed; their premises always involve typechecking smaller expressions, or deciding syntactic equality of types, or computing hereditary substitutions, or deciding sequents of the assertion logic. Checking syntactic equality is obviously a terminating algorithm, and as shown in Theorem 1, hereditary substitutions are terminating as well. Thus, if the validity of each assertion logic sequent can be decided, so too can the typing judgments. $\square$

Theorem 2 essentially states that typechecking of HTT can be divided into the two phases described at the beginning of Section 2. Whenever a sequent is encountered during typechecking, we can use the oracle to decide its validity, or simply view the sequent as a verification condition, and postpone calling the oracle (that is, proving the sequent) until later. Theorem 2 guarantees that if deciding sequents is postponed, then typechecking terminates. Thus, we can divide typechecking into two phases. The first phase checks the underlying simple types, and generates the verification conditions. Since HTT is a higher-order language, this process is non-trivial, as it involves *normalization*. However, by Theorem 2, it is terminating. In the second phase the verification conditions are discharged.

It should be possible to remove the assumption about the oracle in Theorem 2, by extending HTT with certificates for the sequents, in the style of Proof-Carrying Code (Necula, 1997). 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 (1996), the judgments become *analytic*, or self-evident. Alternatively, we can say that an HTT computation can be viewed as a proof of its own specification, and thus the effectful fragment of HTT establishes a correspondence in the style of Curry-Howard isomorphism (Howard, 1980) between computations and specification proofs.

The next lemma connects the elim term K with its eta expansion.

Lemma 3 (*Identity principle*)
If $\Delta \vdash K \Rightarrow A\,[K]$, then $\Delta \vdash \mathsf{expand}_A(K) \Leftarrow A\,[\mathsf{expand}_A(K)]$.

Proof
By induction on the structure of A, using an auxiliary lemma about expansion of variables (omitted here), which roughly states that if $x{:}A$ is a free variable in a well-typed expression N, then hereditarily substituting $\mathsf{expand}_A(x)$ for x does not change N. Intuitively, the later holds because typing ensures that x is correctly used in N, so that when $\mathsf{expand}_A(x)$ is substituted, the created redexes are immediately hereditarily reduced, preserving the syntactic form of N. $\square$

We proceed to restate in the context of HTT the usual properties of Hoare Logic, such as weakening of the consequent and strengthening of the precedent. Also included is the property of relational composition, which states that a computation does not depend on how the heap in which it executes has been obtained; what matters is only the current state of the heap. In other words, if a computation has a precondition P and postcondition Q (which, in the typing judgments are binary relations on heaps), these can be composed with an arbitrary relation R into a new precondition $R \circ P$ and a new postcondition $R \circ Q$. This property is important for the meta-theory of HTT, because the typing rules compute strongest postcondition by composing binary relations.

Lemma 4 (*Properties of computations*)
Suppose that $\Delta; P \vdash E \Leftarrow x{:}A.\ Q\,[E']$. Then:

1. *Weakening Consequent.* If $\Delta, x{:}A; \mathsf{init}, \mathsf{mem}; Q \Longrightarrow R$, then $\Delta; P \vdash E \Leftarrow x{:}A.\ R\,[E']$.
2. *Strengthening Precedent.* If $\Delta; \mathsf{init}, \mathsf{mem}; R \Longrightarrow P$, then $\Delta; R \vdash E \Leftarrow x{:}A.\ Q\,[E']$.
3. *Relational Composition.* If $\Delta; \mathsf{init}, \mathsf{mem} \vdash R \Leftarrow \mathsf{prop}\,[R]$, then $\Delta; R \circ P \vdash E \Leftarrow x{:}A.\ (R \circ Q)\,[E']$.

Proof
Weakening of consequent is straightforward. From $\Delta; P \vdash E \Leftarrow x{:}A.\ Q\,[E']$ we know that there exists an assertion S, such that $\Delta; P \vdash E \Rightarrow x{:}A.\ S$ where $\Delta, x{:}A; \mathsf{init}, \mathsf{mem}; S \Longrightarrow Q$. Applying the rule of cut, we get $\Delta, x{:}A; \mathsf{init}, \mathsf{mem}; S \Longrightarrow R$, and thus $\Delta; P \vdash E \Leftarrow x{:}A.\ R\,[E']$.

Strengthening precedent and relational composition are proved by induction on the structure of E, using the identity principles. □

We next use the properties of computations to prove that the Frame Rule from Separation Logic is admissible. The Frame Rule captures the essence of small footprints, and guarantees that a computation cannot touch the portion of the heap that is not described by its precondition.

Lemma 5 (*Frame*)
If $\Delta \vdash \mathsf{do}\ E \Leftarrow \Psi.X.\{P\}x{:}A\{Q\}\,[E']$, and $\Delta, \Psi; X, \mathsf{mem} \vdash R \Leftarrow \mathsf{prop}\,[R]$, then $\Delta \vdash \mathsf{do}\ E \Leftarrow \Psi.X.\{P * R\}x{:}A\{Q * R\}\,[E']$.

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

Notice that that following sequents are derivable.

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

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

We close this section with the statement of the substitution principles, which formalize the interplay between the ordinary substitutions on general terms, and hereditary substitutions on the canonical versions of the same terms. Intuitively, *substitution commutes with normalization.* That is, ordinary substitution followed by normalization, produces the same result as computing the canonical forms first, and then employing hereditary substitutions.

Thus, unlike in most other works on dependent types, where proving soundness usually requires proving strong normalization, here we establish a somewhat weaker, but sufficient, result. Instead of showing that *every* reduction order yields the same canonical form, we only need to know that the particular reduction order implemented by the typechecker, is stable under substitution.

Lemma 6 (*Type substitution principles*)
Suppose that $\Delta \vdash \tau \Leftarrow \mathsf{mono}\,[\tau']$, and denote by $\overline{T}$ the operation of hereditary type substitution $[\tau'/\alpha]^*(T)$, for any kind of expression T. Then the following holds.

1. If $\Delta, \alpha, \Delta_1 \vdash K \Rightarrow B\,[O]$, then $\Delta, \overline{\Delta_1} \vdash [\tau/\alpha]K \Rightarrow \overline{B}\,[\overline{O}]$.
2. If $\Delta, \alpha, \Delta_1 \vdash N \Leftarrow B\,[O]$, then $\Delta, \overline{\Delta_1} \vdash [\tau/\alpha]N \Leftarrow \overline{B}\,[\overline{O}]$.
3. If $\Delta, \alpha, \Delta_1; P \vdash E \Leftarrow y{:}B.\ Q\,[F]$, and $y \notin \mathsf{FV}(M)$, then $\Delta, \overline{\Delta_1}; \overline{P} \vdash [\tau/\alpha]E \Leftarrow y{:}\overline{B}.\ \overline{Q}\,[\overline{F}]$.
4. If $\Delta, \alpha, \Delta_1; X; \Gamma_1 \Longrightarrow \Gamma_2$, then $\Delta, \overline{\Delta_1}; X; \overline{\Gamma}_1 \Longrightarrow \overline{\Gamma}_2$.

Lemma 7 (*Term substitution principles*)
Suppose that $\Delta \vdash A \Leftarrow \mathsf{type}\,[A']$ and $\Delta \vdash M \Leftarrow A'\,[M']$, and denote by $\overline{T}$ the operation of hereditary substitution $[M'/x]^*_{A'}(T)$, for any kind of expression T. Then the following holds.

1. If $\Delta, x{:}A', \Delta_1 \vdash K \Rightarrow B\,[O]$, then $\Delta, \overline{\Delta_1} \vdash [M{:}A/x]K \Rightarrow \overline{B}\,[\overline{O}]$.
2. If $\Delta, x{:}A', \Delta_1 \vdash N \Leftarrow B\,[O]$, then $\Delta, \overline{\Delta_1} \vdash [M{:}A/x]N \Leftarrow \overline{B}\,[\overline{O}]$.
3. If $\Delta, x{:}A', \Delta_1; P \vdash E \Leftarrow y{:}B.\ Q\,[F]$, and $y \notin \mathsf{FV}(M)$, then $\Delta, \overline{\Delta_1}; \overline{P} \vdash [M{:}A/x]E \Leftarrow y{:}\overline{B}.\ \overline{Q}\,[\overline{F}]$.
4. If $\Delta, x{:}A, \Delta_1; X; \Gamma_1 \Longrightarrow \Gamma_2$, then $\Delta, \overline{\Delta_1}; \Psi; \overline{\Gamma}_1 \Longrightarrow \overline{\Gamma}_2$.
5. If $\Delta; P \vdash E \Leftarrow x{:}A'.\ Q\,[E']$ and $\Delta, x{:}A'; Q \vdash F \Leftarrow y{:}B.\ R\,[F']$, where $x \notin \mathsf{FV}(B, R)$, then $\Delta; P \vdash \langle E/x{:}A\rangle F \Leftarrow y{:}B.\ R\,[\langle E'/x\rangle_A(F')]$.

Most of the substitution principles are simple adaptations of the fairly standard principles encountered in any dependent type theory. However, it may be interesting to point out here that the monadic substitution principle (Lemma 7.5) actually states that the *sequential composition rule* usually encountered in Hoare Logics is *admissible* in HTT. Indeed, Lemma 7.5 states that a computation E with a postcondition Q can be sequentially composed with another computation F, if F has a precondition Q.

Both type and term substitution principles are proved by simultaneous induction of the main derivations, after generalizing to include the formation judgments of HTT. The proofs makes essential use of auxiliary *hereditary substitution principles* (omitted here), which have similar statement as the lemmas above, but involve only canonical forms.

6 Operational semantics

In this section we define the call-by-value, left-to-right operational semantics for HTT and prove that the type system is sound 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 HTT types correspond to specifications, our approach is different from the traditional approach of Hoare Logic but it is similar to the approach in (Birkedal *et al.*, 2005), which also only gives semantics to well-specified programs.

Syntax. The syntactic domains required by the operational semantics are as follows.

$$\begin{array}{lcll}
\textit{Values} & v, l & ::= & (\,) \mid \lambda x.\ M \mid \Lambda \alpha.\ M \mid \mathsf{do}\ E \mid \mathsf{true} \mid \mathsf{false} \mid \mathsf{z} \mid \mathsf{s}\ v \\
\textit{Value heaps} & \chi & ::= & \cdot \mid \chi, l \mapsto_\tau v \\
\textit{Continuations} & \kappa & ::= & \cdot \mid x{:}A.\ E; \kappa \\
\textit{Control expressions} & \rho & ::= & \kappa \triangleright E \\
\textit{Abstract machines} & \mu & ::= & \chi, \kappa \triangleright E
\end{array}$$

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.

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

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 $x \leftarrow \mathsf{do}\ E; F$ 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. We present selected rules in Figure 2.

For example, in the evaluation of intro terms, if the intro term is obtained by coercion from an elim term, we invoke the judgment for elim terms. If the returned result is of the form $v : A$, we remove the type annotation. This prevents accumulation of type annotations, as in $v : A_1 : \cdots : A_n$. In the evaluation of the abstract machine $\chi, \kappa \triangleright E$, we first reduce E to a value, which is plugged into the continuation κ to proceed. Occasionally we must check that the types given at the input abstract machine are well-formed, so that the output abstract machine is well-formed as well. The outcome of the evaluation, however, does not depend on type information, if we assume that the input to evaluation is well formed.

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 8 (*Preservation*)

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

Proof

The first two statements are proved by simultaneous induction on the evaluation judgment, using inversion on the typing derivation, and substitution principles. The third statement is proved by case analysis on the evaluation judgment, using the first two statements. We also tacitly use an auxiliary replacement lemma (omitted here), which roughly states that replacing the computation E in the control expression $\kappa \triangleright E$ with another computation F preserves the typing of the control expression, as long as E and F have the same postconditions, and thus both provide the same precondition for the execution of κ. $\Box$

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

Evaluation of elim terms

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

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

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

Evaluation of intro terms

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

Evaluation of abstract machines

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

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

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

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

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

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

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

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

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

$$\frac{}{\chi, \kappa \triangleright x = \mathsf{case}_A\,(\mathsf{z})\ \mathsf{of}\ \mathsf{z}.E_1\ \mathsf{or}\ \mathsf{s}\,y.E_2; E \hookrightarrow_e \chi, x{:}A.\ E; \kappa \triangleright E_1}$$

$$\frac{}{\chi, \kappa \triangleright x = \mathsf{case}_A\,(\mathsf{s}\,v)\ \mathsf{of}\ \mathsf{z}.E_1\ \mathsf{or}\ \mathsf{s}\,y.E_2; E \hookrightarrow_e \chi, x{:}A.\ E; \kappa \triangleright [v{:}\mathsf{nat}/y]E_2}$$

$$\frac{N = \lambda z.\ \mathsf{do}\ (y = \mathsf{fix}\, f(x{:}A){:}B = \mathsf{do}\ E\, \mathsf{in\, eval}\, f\ z; y) \qquad B = \Psi.X.\{R_1\}y{:}C\{R_2\}}{\begin{array}{c}\chi, \kappa \triangleright y = \mathsf{fix}\, f(x{:}A){:}B = \mathsf{do}\ E\, \mathsf{in\, eval}\, f\ v; F \hookrightarrow_e \\ \chi, (y{:}[v{:}A/x]C.\ F; \kappa) \triangleright [v{:}A/x, N{:}\Pi x{:}A.B/f]E\end{array}}$$

Fig. 2. Selected evaluation rules (omitted are primitive arithmetic operations and non-redex cases of abstract machines).

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

1. if $\cdot; \mathsf{mem}; \mathsf{this}(\llbracket\chi\rrbracket) \Longrightarrow l \hookrightarrow_\tau -$, then $l \mapsto_\tau v \in \chi$, for some value v, and
2. if $\cdot; \mathsf{mem}; \mathsf{this}(\llbracket\chi\rrbracket) \Longrightarrow l \hookrightarrow -$, then $l \mapsto_\tau v \in \chi$ for some monotype τ and a value v.

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. 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. 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 10 (*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 $\vdash \chi_0, \kappa_0 \rhd E_0 \Leftarrow x{:}A.\ Q$, then either $E_0 = v$ and $\kappa_0 = \cdot$, or $\chi_0, \kappa_0 \rhd E_0 \hookrightarrow_e \chi_1, \kappa_1 \rhd E_1$, for some χ_1, κ_1, E_1.

Proof
By straightforward case analysis on the involved expressions, employing inversion on the typing derivations, using heap soundness in the cases of third statement involving the primitive effectful commands for allocation, lookup and update. □

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.

7 Heap soundness

In this section we sketch a proof that the assertion logic is heap sound. For the most part, the assertion logic is a standard first-order logic with polymorphism. However, it also admits reasoning about heaps and monadic computations. We need to show that the presence of heaps and computations does not cause unsoundness, and we do so by means of a simple, and somewhat crude set-theoretic model of HTT.

Our model depends on the observation that the assertion logic does not include axioms for computations; reasoning about computations is formalized via the typing rules and soundness of those is proved above via progress and preservation assuming soundness of the assertion logic (heap soundness).

Thus in our set-theoretic model, we chose to simply interpret the computation types $\Psi.X.\{P\}x{:}A\{Q\}$ and $\Psi.X.\{P\}x{:}\tau\{Q\}$ as one-element sets, emphasizing that the assertion logic cannot distinguish between different computations. Given this basic decision we are really just left with interpreting a type theory similar to the Extended Calculus of Constructions with a (assertion) logic on top. The type theory has two universes (mono and other types) and is similar to the Extended Calculus of Construction (ECC), except that the mono universe is not impredicative. Hence we can use a simplified version of Luo's model of ECC (Luo, 1990) for the types. Thus our model is really fairly standard and hence we only include a sketch of it here.

As in (Luo, 1990) our model takes place in ZFC set theory with infinite inaccessible cardinals κ_0, κ_1, … (see *loc. cit.* for details). The universe mono is the set of all sets of cardinality smaller than κ_0. The type nat is interpreted as the set of natural numbers, bool as the set of booleans, $\Pi x{:}A.\ B$ and $\Pi x{:}\tau.\ \sigma$ as dependent product in sets, $\Sigma x{:}A.\ B$ and $\Sigma x{:}\tau.\ \sigma$ as dependent sum in sets. Heaps are interpreted as finite partial functions from the set of natural numbers to $\Sigma\alpha{:}\mathsf{mono}.\ \alpha$. Predicates P on a type are interpreted as subsets in the classical way.

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

Theorem 11 (*Heap Soundness*)
The assertion logic of HTT is heap sound.

Proof
Let χ be a value heap. Here we only sketch the argument for case (1) of heap soundness (Definition 9); case (2) is proved analogously.

By assumption $\cdot; \mathsf{mem}; \mathsf{this}([\![\chi]\!]) \Longrightarrow l \hookrightarrow_\tau -$ is derivable. By soundness of the assertion logic we have that $\cdot; \mathsf{mem}; \mathsf{this}([\![\chi]\!]) \Longrightarrow l \hookrightarrow_\tau -$ is true. By definition of the interpretation of $\hookrightarrow_\tau$ this means that $\exists v \in [\![\tau]\!]\ .\ [\![[\![\chi]\!]]\!](l) = v$. 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). $\square$

Note that the denotational model above does not model predicates as *admissible*[1] 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 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

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

model. We are not aware of similar combinations of models and proof methods for models of higher-order store in the literature.

8 Related work

We divide the related work into three groups: (1) Hoare Logics for higher-order effectful programs, (2) dependent type systems for pure and impure functional programming, and (3) languages and tools for extended static checking of effectful programs.

Hoare Logics for higher-order programs. Honda, Berger and Yoshida in (Honda *et al.*, 2005; Berger *et al.*, 2005) present several Hoare Logics for total correctness of PCF with references, where specifications in the form of Hoare triples are taken as propositions. Krishnaswami (2006) proposes a version of Separation Logic for a core monomorphic ML-like language. Similarly to HTT, Krishnaswami bases his logic on a monadic presentation of the underlying programming language. Both proposals do not support polymorphism, strong updates, deallocation or pointer arithmetic. Both are Hoare-like Logics, rather than type theories, and do not integrate expressive specifications into types.

Dependent types for programming. The work on dependently typed systems with stateful features has mostly focused on how to appropriately restrict the effects from appearing in the language of types. If types only depend on pure terms, it becomes possible to use logical reasoning about them. Such systems have mostly employed singleton types to enforce purity. Examples include Dependent ML by Xi and Pfenning (1998; 1999), Applied type systems by Chen and Xi (2005) and Zhu and Xi (2005), and a type system for certified binaries by Shao et al. (2005). HTT differs from these approaches, because types are allowed to depend on monadically encapsulated effectful computations. We also mention the theory of type refinements by Mandelbaum et al. (2003), which reasons about programs with effects, by employing a restricted fragment of linear logic. The restriction on the logic limits the class of properties that can be described, but is undertaken to preserve decidability of typechecking.

There are also several recent proposals for purely functional languages with dependent types. Examples include Cayenne (Augustsson, 1998), Epigram (McBride & McKinna, 2005), Omega (Sheard, 2004) and Sage (Flanagan, 2006).

Cayenne is an extension of Haskell that allows arbitrary values to occur in types. Like Haskell, Cayenne considers diverging computations to be pure, and admits them into type dependencies. As commented in (Augustsson, 1998), this makes it possible to construct a proof for every proposition that appears in a Cayenne type, by simply building a diverging program whose type equals the required proposition. Hence, Cayenne, as a logic, is unsound. From the computational perspective, this is not necessarily bad, as long as unsoundness is manifested as divergence, rather than a run-time type error. But, it would seem to imply that all program transformations must preserve diverging behavior of the underlying code, which may complicate some optimizations (Gill *et al.*, 1993). Thus, we believe that it is much better if divergence is considered to be an effect, and is appropriately isolated by the type system, as done in HTT.

Epigram is based closely on Luo's UTT (Luo, 1994), and supports a number of practical programming features, like inference of implicit arguments, non-uniform families of types, sophisticated forms of pattern matching, and the use of explicit proofs within programs. Epigram does not currently support imperative features.

Omega (Sheard, 2004) employs singleton types, in a way similar to DML and ATS, to refine the type system of the underlying language. It allows the use of explicit proofs within programs, in order to discharge the various conditions that arise during typechecking.

Sage (Flanagan, 2006) makes typechecking of expressive specifications manageable by combining proving and run-time checks. When the Sage typechecker cannot automatically discharge some verification condition, it inserts an explicit test in the code. If the test fails at run-time, the program is forced into an error state, and the failing test instance is remembered as a falsehood for future use in theorem proving. We believe that a similar functionality can easily be built into HTT, at least for those properties that can actually be computationally verified (this excludes any non-trivial use of quantifiers).

In addition to the purely functional languages described above, we list RSP1 (Westbrook *et al.*, 2005), and Deputy (Condit *et al.*, 2007), as two proposal for using dependent types for specification and reasoning about state in first-order and low-level languages, respectively. HTT differs from these approaches, as we support fully higher-order, as well as polymorphic, imperative programs and stores.

Extended static checking. There are also a number of languages and tools that integrate Hoare-style specifications into the type system and provide a mode of extended static checking of imperative programs. Examples include ESC/Modula-3 (Detlefs *et al.*, 1998), ESC/Java (Leino *et al.*, 2000), JML (Leavens *et al.*, 1999; Burdy *et al.*, 2005), Spec# (Barnett *et al.*, 2004), SPLint (Evans & Larochelle, 2002) and SPARK/Ada (Barnes, 2003). All these systems use the annotations to statically generate the appropriate verification conditions, which are then discharged by an automated theorem prover, or by inserting a run-time check.

For example, SPARK/Ada syntactically limits the programming language, so that the assertions are always decidable. This makes it inapplicable in many situations, but provides a desired guarantee that well-typed programs do not go wrong. Spec# inserts a run-time check whenever it cannot prove a certain condition (with similar restrictions as described above for Sage). In contrast, ESC/Java is designed simply to warn the programmer of a potential error, but does not offer any soundness guarantees.

Finally, we mention two theoretical frameworks for programming with expressive specifications. Abadi and Leino (2004) describe a logic for object-oriented programs where specifications, like in HTT, are treated as types. One of the problems that authors describe concerns the scoping of variables; certain specifications cannot be proved because the inference rule for $\mathsf{let\ val}\ x = E\ \mathsf{in}\ F$ does not allow sufficient interaction between the specifications of E and F. Such problems do not appear in HTT. Birkedal et al. (2005) describe a dependent type system for well-specified programs in idealized Algol extended with heaps. The type system includes a wide collection of higher-order frame rules, which are shown sound by a denotational model. A serious limitation of the type system compared to HTT is that the heap in *loc. cit.* can only contain simple integer values.

9 Future work

We briefly sketch several ideas that we plan to carry out in the future: scaling HTT to include quantification over propositions, capturing locality and ownership of state in the type system, investigation of reasoning principles for effectful computations, addition of inductive and recursive types, and stateful types like references and arrays, and other applications of type systems with Hoare types.

Higher-order quantification and local state. The type system presented in this paper does not provide any constructs for quantification over assertions. Yet, this clearly is an important operation. For example, abstracting a proposition at the level of terms and types makes it possible to hide certain details about a function or a module, so that different implementations may share the same signature. This is crucial for modular programming and code reuse. Abstracting a proposition at the level of assertions increases the power of the assertion logic to higher-order, thus providing internal means for defining new predicates (including inductive and coinductive ones) and for reasoning about a large class of types (Church, 1940; Paulson, 2000; SRI International & DSTO, 1991).

In (Nanevski *et al.*, 2007), we have already carried out preliminary investigations into the higher-order extensions, and scaled HTT to include a significant fragment of the Extended Calculus of Constructions (Luo, 1990). It turns out that when combined with Hoare types, abstraction over predicates can represent the local state of modules within the type system. Local state is shared by several functions of the module, but can be hidden from the clients. The precision of the abstracted predicate controls how much information about local state is revealed. By judiciously choosing this predicate, modules may completely protect their local state, or grant a partial or total access to, or even ownership of portions of local state to the clients.

We also believe that higher-order assertion logic is the appropriate framework for studying type and annotation inference for HTT, as higher-order logic should be strong enough to represent any pre- and postcondition that may appear during program verification. This is directly related to the property of Cook completeness (Cook, 1978), which we would also like to prove for HTT.

Reasoning about effectful code. One of the main obstacles in the design of languages for integrated programming and reasoning, as for example outlined in (Burdy *et al.*, 2005), is that effectful code usually cannot appear in specifications. This is problematic from the software engineering point of view, as it leads to code duplication.

Almost every functionality must be implemented once purely, to be used in specification and reasoning, and once impurely, for efficient execution.

In HTT, we admit effectful code in specifications, but that is only the first step towards solving the above problem, as the degree to which such code can be reasoned about is determined by the available equational laws. Currently, HTT admits only the generic monadic laws, which do not capture all the semantic properties of computations. In the future, we plan to investigate which additional equations can be soundly added. This might require developing more refined denotational models than the one described in this paper, and formalizing them in the assertion logic of HTT, so that the equalities between computations can be certified when required.

Incorporating other notions of effects. In this paper, we applied HTT to the problem of reasoning about state with aliasing. But other computational effects and applications seem possible too. For example, we would like HTT to support control effects like exceptions and continuations. Some related work in this direction may be (Tan & Appel, 2006), which extends Hoare Logic with compositional reasoning about programs with labels and goto commands. We also plan for HTT to support concurrency. Separation Logic has been used recently to reason about concurrent programs (O'Hearn *et al.*, 2004; Feng *et al.*, 2007) and we hope that these ideas may be embedded into HTT as well. On the other hand, an approach to concurrency based on software transactional memory, as recently implemented in Haskell (Harris *et al.*, 2005), may be particularly well-suided to the monadic nature of Hoare types. Another interesting extension may involve reasoning through Hoare types about information flow and security (Amtoft *et al.*, 2006).

Stateful types. The types of HTT are currently all pure, in the sense that we abstract away from their actual layout in memory. It is possible to describe the layout of the memory and reason about it *using assertions*, but it is not possible to capture the memory into a type, such as the type of references or arrays, which are inherently stateful.

Yet, such stateful types are important because they can lead to a higher degree of automation of the reasoning principles, as well as to a significant reduction in the size and number of verification conditions. Indeed, in the current paper, we must prove before every location access that the location at the given address exists. On the other hand, if locations were a separate type, rather than natural numbers, we could have a more fine-grained control over them. For example, we could make locations persistent, as in ML and Haskell. Then simple typechecking would guarantee access safety, without needing any additional verification conditions.

One of the challenges in adding stateful types is deciding which axioms they should satisfy. For example, we can always consider references to simply be natural numbers (as currently in HTT), but references are often expected to exhibit a certain recursive behavior, and thus may need to be defined as some sort of recursive type (please see below).

The extension with stateful types may be related to the recent substructural type systems for state, such as the work of Ahmed, Fluet and Morrisett (2005; 2005; 2006), Collinson and Pym (2006), and the logic of Reus and Schwinghammer (2006).

Recursive types. Inductive types have been extensively studied in dependent type theories (Dybjer, 1994; Dybjer & Setzer, 2006), including pattern-matching against them (McBride & McKinna, 2005). However, general recursive types have not received as much attention, possibly because recursive types introduce divergence into the term language, and thus destroy the logical properties of the theory.

In HTT, it should be possible to add recursive types safely, as long as divergence is monadically encapsulated within the Hoare types. This may be achieved by pushing the elimination construct for the recursive type into the effectful fragment. Typing rules for the introduction and elimination of recursive types may be roughly as follows.

$$\frac{\Delta \vdash M \Leftarrow [\mu\alpha.A/\alpha]^a(A)}{\Delta \vdash \mathsf{fold}_{\mu\alpha.A}\, M \Rightarrow \mu\alpha.A}$$

$$\frac{\Delta \vdash K \Rightarrow \mu\alpha.A \qquad \Delta, x{:}[\mu\alpha.A/\alpha]^a(A); P \wedge \mathsf{id}_{\mu\alpha.A}(\mathsf{fold}\, x, K) \vdash E \Rightarrow y{:}B.\ Q}{\Delta; P \vdash x = \mathsf{unfold}\, K; E \Rightarrow y{:}B.\ (\exists x{:}[\mu\alpha.A/\alpha]^a(A).Q)}$$

We believe that recursive types are also the correct way of controlling the recursive behavior that arises in the presence of higher-order store and the type of references (known in the literature as “tying the Landin’s knot”). The store in HTT is higher-order, as locations may point to arbitrary higher-order functions and stateful computations, but HTT does not admit this kind of recursion through the store. Our typing rules require that each computation specifies all the locations that it may touch. Thus, such locations must be created *before* the computation and its type. It is not possible to create a location of a Hoare type that depends on the very same location, thus potentially creating a loop. If a loop in the store is required, it should be specified either using assertions in the pre- and postconditions of computations (already available in HTT), or using recursive types.

We consider this to be an advantage of HTT over other stateful languages. It seems to us as a good design to delegate all the concerns about recursion to explicit fixpoint constructs, rather than allow implicit recursive behavior to arise through interaction of otherwise unrelated programming features.

10 Conclusions

We present Hoare Type Theory (HTT), which is a novel framework for extending a dependently typed functional language with imperative programming features like state and non-termination. The supported operations on state involve allocation, deallocation, lookup and strong update, so that a location may be updated with values of varying types. The store in HTT is higher-order, in the sense that locations may point to arbitrary higher-order functions and effectful computations. We also admit pointer arithmetic.

The main idea that allows for safe combination of dependent types and effects follows the familiar "specifications-as-types" principle of type theory (Howard, 1980). We introduce the distinguished Hoare type $\{P\}x{:}A\{Q\}$, which classifies effectful programs with a precondition P, postcondition Q and return value $x{:}A$, thus internalizing into the type system the well-known specification methodology from Hoare Logic (Hoare, 1969). The addition of Hoare types makes it possible to statically track the use of stateful operations, and make sure they are performed only if the required preconditions are satisfied. Hoare types also generalize the notion of monads, used extensively in modern simply-typed functional programming (Peyton Jones, 2003).

Embedding specifications into types has significant advantages over the original Hoare Logic. In particular, if specifications are treated as types, then it becomes possible to combine the specifications with the source code, allowing for nesting, combination with other types, abstraction and packaging of a specification with the data whose properties it describes. All of these operations are essential from the point of view of modularity, information hiding and data abstraction, but are not available in Hoare Logic. For example, we are able to seamlessly incorporate Hoare types with other higher-order features like higher-order functions and polymorphism, which were traditionally a difficult extension in Hoare Logic (Cartwright & Oppen, 1978; O'Donnell, 1982).

We also show that in the presence of polymorphism, we can define the propositional connectives from Separation Logic, which has been recently proposed as an extension to Hoare Logic, particularly suited for reasoning about pointer aliasing. In addition, HTT specifications can quantify over heap variables to name the particular heap fragments of interest. This ability is not directly available in Separation Logic, but in HTT we can use it to freely combine the Separation Logic specifications in the so-called "small-footprint" style, with the classical "large-footprint" approach. We believe that the former is preferable when specifying the lack of aliasing or freshness of newly-generated locations, but the later may be more compact in the cases when aliasing is allowed. Also, explicit naming comes handy when specifying the invariance of a subheap which the computation may read from during execution. Such a subheap is in the footprint of the computation, so that its invariance cannot be inferred using the Frame Rule of Separation Logic, but must instead be explicitly stated in the specification.

We further establish that HTT is sound, in the usual sense of Preservation and Progress theorems. That is, evaluation of HTT terms preserves the type information, and well-typed terms never get stuck. Furthermore, we establish that HTT satisfies the customary substitution principles of dependent type theories, and also show that the usual Hoare-style rule for sequential composition is admissible. It may be interesting that the later admissible rule also takes the form of a substitution principle, associated with the operation of monadic substitution of effectful programs.

The substitution principles formally establish the modular nature of HTT, and guarantee that separate verifications of program modules suffice to verify the composite program. Crucial in proving the substitution principles was the *syntax-directed* nature of HTT typechecking, ensuring that the verification conditions for any given computation depend on the computation alone, rather than on an unpredictable global context in which the computation may execute. This property is achieved by organizing the typing rules so as to compute the strongest postconditions for effectful code, following the general idea of *predicate transformers* (Dijkstra, 1975). Thus, in addition to the "specifications-as-types" principle which we adopted by introducing Hoare types, we believe that HTT also provides a form of "proofs-as-programs" principle for effectful code (Howard, 1980).

References

Abadi, Martin, & Leino, K. Rustan M. (2004). A logic of object-oriented programs. *Pages 11–41 of: Verification: Theory and practice*. Springer-Verlag.

Ahmed, Amal, Fluet, Matthew, & Morrisett, Greg. 2005 (September). A step-indexed model of substructural state. *Pages 78–91 of: International Conference on Functional Programming, ICFP'05*.

Altenkirch, Thorsten, Dybjer, Peter, Hofmann, Martin, & Scott, Phil. (2001). Normalization by evaluation for typed lambda calculus with coproducts. *Pages 303–310 of: Symposium on Logic in Computer Science, LICS'01*.

Amtoft, Torben, Bandhakavi, Sruthi, & Banerjee, Anindya. (2006). A logic for information flow in object-oriented programs. *Pages 91–102 of: Symposium on Principles of Programming Languages, POPL'06*.

Augustsson, Lennart. (1998). Cayenne – a language with dependent types. *Pages 239–250 of: International Conference on Functional Programming, ICFP'98*.

Barnes, John. (2003). *High integrity software: The SPARK approach to safety and security*. Addison-Wesley.

Barnett, Mike, Leino, K. Rustan M., & Schulte, Wolfram. (2004). The Spec# programming system: An overview. *CASSIS 2004*. Lecture Notes in Computer Science. Springer.

Berger, Martin, Honda, Kohei, & Yoshida, Nobuko. 2005 (September). A logical analysis of aliasing in imperative higher-order functions. *Pages 280–293 of:* Danvy, Olivier, & Pierce, Benjamin C. (eds), *International Conference on Functional Programming, ICFP'05*.

Biering, B., Birkedal, L., & Torp-Smith, N. 2005 (July). *BI hyperdoctrines, Higher-Order Separation Logic, and Abstraction*. Tech. rept. ITU-TR-2005-69. IT University of Copenhagen, Copenhagen, Denmark.

Birkedal, Lars, Torp-Smith, Noah, & Reynolds, John C. (2004). Local reasoning about a copying garbage collector. *Pages 220–231 of: Symposium on Principles of Programming Languages, POPL'04*.

Birkedal, Lars, Torp-Smith, Noah, & Yang, Hongseok. 2005 (June). Semantics of separation-logic typing and higher-order frame rules. *Pages 260–269 of: Symposium on Logic in Computer Science, LICS'05*.

Burdy, Lilian, Cheon, Yoonsik, Cok, David, Ernst, Michael, Kiniry, Joe, Leavens, Gary T., Leino, K. Rustan M., & Poll, Erik. (2005). An overview of JML tools and applications. *International journal on software tools for technology transfer*, **7**(3), 212–232.

Cartwright, Robert, & Oppen, Derek C. (1978). Unrestricted procedure calls in Hoare's logic. *Pages 131–140 of: Symposium on Principles of Programming Languages, POPL'78*.

Chen, Chiyan, & Xi, Hongwei. 2005 (September). Combining programming with theorem proving. *Pages 66–77 of: International Conference on Functional Programming, ICFP'05*.

Church, Alonzo. (1940). A formulation of the simple theory of types. *The journal of symbolic logic*, **5**(2), 56–68.

Collinson, Matthew, & Pym, David J. (2006). Bunching for regions and locations. *Electronic Notes in Theoretical Computer Science*, **158**, 171–197.

Condit, Jeremy, Harren, Matthew, Anderson, Zachary, Gay, David, & Necula, George. 2007 (March). Dependent types for low-level programming. *Pages 520–535 of: European Symposium on Programming, ESOP'07.* Lecture Notes in Computer Science, vol. 4421.

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

Detlefs, David L., Leino, K. Rustan M., Nelson, Greg, & Saxe, James B. 1998 (December). *Extended static checking.* Compaq Systems Research Center, Research Report 159.

Dijkstra, Edsger W. (1975). Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, **18**(8), 453–457.

Dybjer, Peter. (1994). Inductive families. *Formal aspects of computing*, **6**(4), 440–465.

Dybjer, Peter, & Setzer, Anton. (2006). Indexed induction-recursion. *Journal of logic and algebraic programming*, **66**(1), 1–49.

Evans, David, & Larochelle, David. (2002). Improving security using extensible lightweight static analysis. *IEEE Software*, **19**(1), 42–51.

Feng, Xinyu, Ferreira, Rodrigo, & Shao, Zhong. (2007). On the relationship between concurrent separation logic and assume-guarantee reasoning. *Pages 173–188 of: European Symposium on Programming, ESOP'07.* Lecture Notes in Computer Science, vol. 4421.

Flanagan, Cormac. (2006). Hybrid type checking. *Pages 245–256 of: Symposium on Principles of Programming Languages, POPL'06.*

Fluet, Matthew, Morrisett, Greg, & Ahmed, Amal. 2006 (March). Linear regions are all you need. *Pages 7–21 of: European Symposium on Programming, ESOP'06.*

Ghani, N. (1995). Beta-eta equality for coproducts. *Pages 171–185 of: International Conference on Typed Lambda Calculus and Applications, TLCA'95.* Lecture Notes in Computer Science, no. 902. Springer.

Gill, Andrew, Launchbury, John, & Peyton Jones, Simon L. 1993 (June). A short cut to deforestation. *Pages 223–232 of: Conference on functional programming languages and computer architecture.*

Girard, Jean-Yves, Lafont, Yves, & Taylor, Paul. (1989). *Proofs and types.* Cambridge University Press.

Greif, I., & Meyer, A. (1979). Specifying programming language semantics: a tutorial and critique of a paper by Hoare and Lauer. *Pages 180–189 of: Symposium on Principles of Programming Languages, POPL'79.*

Harris, Tim, Marlow, Simon, Peyton Jones, Simon, & Herlihy, Maurice. (2005). Composable memory transactions. *Pages 48–60 of: Symposium on Principles and Practice of Parallel Programming, PPoPP'05.*

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

Hofmann, Martin. 1995 (July). *Extensional concepts in intensional type theory.* Ph.D. thesis, Department of Computer Science, University of Edinburgh. Avaliable as Technical Report ECS-LFCS-95-327.

Honda, Kohei, Yoshida, Nobuko, & Berger, Martin. 2005 (June). An observationally complete program logic for imperative higher-order functions. *Pages 270–279 of: Symposium on Logic in Computer Science, LICS'05.*

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

Jim, Trevor, Morrisett, Greg, Grossman, Dan, Hicks, Michael, Cheney, James, & Wang, Yanling. 2002 (June). Cyclone: A safe dialect of C. *Pages 275–288 of: USENIX Annual Technical Conference.*

Krishnaswami, Neelakantan. (2006). Separation logic for a higher-order typed language. *Pages 73–82 of: Workshop on Semantics, Program Analysis and Computing Environments for Memory Management, SPACE'06.*

Leavens, Gary T., Baker, Albert L., & Ruby, Clyde. (1999). JML: A notation for detailed design. *Pages 175–188 of:* Kilov, Haim, Rumpe, Bernhard, & Simmonds, Ian (eds), *Behavioral specifications of businesses and systems.* Kluwer Academic Publishers.

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

Leino, K. Rustan M., Nelson, Greg, & Saxe, James B. 2000 (October). *ESC/Java user's manual.* Compaq Systems Research Center. Technical Note 2000-002.

Luo, Zhaohui. (1990). *An extended calculus of constructions.* Ph.D. thesis, University of Edinburgh.

Luo, Zhaohui. (1994). *Computation and reasoning: A type theory for computer science.* Oxford University Press.

Mandelbaum, Yitzhak, Walker, David, & Harper, Robert. 2003 (September). An effective theory of type refinements. *Pages 213–226 of: International Conference on Functional Programming, ICFP'03.*

Martin-Löf, Per. (1996). On the meanings of the logical constants and the justifications of the logical laws. *Nordic journal of philosophical logic*, **1**(1), 11–60.

McBride, Conor. (1999). *Dependently typed functional programs and their proofs.* Ph.D. thesis, University of Edinburgh.

McBride, Conor, & McKinna, James. (2005). The view from the left. *Journal of functional programming*, **14**(1), 69–111.

McCarthy, John L. (1962). Towards a mathematical science of computation. *Pages 21–28 of: IFIP Congress.*

Moggi, Eugenio. (1989). Computational lambda-calculus and monads. *Pages 14–23 of: Symposium on Logic in Computer Science, LICS'89.*

Moggi, Eugenio. (1991). Notions of computation and monads. *Information and computation*, **93**(1), 55–92.

Morrisett, Greg, Ahmed, Amal, & Fluet, Matthew. 2005 (April). L3: A linear language with locations. *Pages 293–307 of: International Conference on Typed Lambda Calculus and Applications, TLCA'05.* Lecture Notes in Computer Science, vol. 3461.

Nanevski, Aleksandar, Ahmed, Amal, Morrisett, Greg, & Birkedal, Lars. (2007). Abstract Predicates and Mutable ADTs in Hoare Type Theory. *Pages 189–204 of: European Symposium on Programming, ESOP'07.* Lecture Notes in Computer Science, vol. 4421.

Necula, George C. 1997 (January). Proof-carrying code. *Pages 106–119 of: Symposium on Principles of Programming Languages, POPL'97.*

O'Donnell, Michael J. (1982). A Critique of the Foundations of Hoare Style Programming Logics. *Communications of the ACM*, **25**(12), 927–935.

O'Hearn, Peter, Reynolds, John, & Yang, Hongseok. (2001). Local reasoning about programs that alter data structures. *Pages 1–19 of: International Workshop on Computer Science Logic, CSL'01.* Lecture Notes in Computer Science, vol. 2142. Springer.

O'Hearn, Peter W., Yang, Hongseok, & Reynolds, John C. (2004). Separation and information hiding. *Pages 268–280 of: Symposium on Principles of Programming Languages, POPL'04.*

Paulson, Lawrence C. (2000). A formulation of the simple theory of types (for Isabelle). *Pages 246–274 of: International Conference in Computer Logic, COLOG'88.* Lecture Notes in Computer Science, vol. 417. Springer.

Peyton Jones, Simon (ed). (2003). *Haskell 98 Language and Libraries: The Revised Report.* Cambridge University Press.

Peyton Jones, Simon L, & Wadler, Philip. (1993). Imperative functional programming. *Pages 71–84 of: Symposium on Principles of Programming Languages, POPL'93.*

Pfenning, Frank, & Davies, Rowan. (2001). A judgmental reconstruction of modal logic. *Mathematical structures in computer science*, **11**(4), 511–540.

Pierce, Benjamin C., & Turner, David N. (2000). Local type inference. *ACM Transactions on Programming Languages and Systems*, **22**(1), 1–44.

Reus, Bernhard, & Schwinghammer, Jan. (2006). Separation logic for higher-order store. *International Workshop on Computer Science Logic, CSL'06.*

Reynolds, John C. (2002). Separation logic: A logic for shared mutable data structures. *Pages 55–74 of: Symposium on Logic in Computer Science, LICS'02.*

Shao, Zhong, Trifonov, Valery, Saha, Bratin, & Papaspyrou, Nikolaos. (2005). A type system for certified binaries. *ACM Transactions on Programming Languages and Systems*, **27**(1), 1–45.

Sheard, Tim. 2004 (October). Languages of the future. *Pages 116–119 of: Object-oriented programming, systems, languages, and applications, OOPSLA'04.*

Smith, Frederick, Walker, David, & Morrisett, Greg. (2000). Alias types. *Pages 366–381 of:* Smolka, Gert (ed), *European Symposium on Programming, ESOP'00.* Lecture Notes in Computer Science, vol. 1782.

SRI International, & DSTO. 1991 (July). *The HOL system: Description.* University of Cambridge Computer Laboratory.

Tan, Gang, & Appel, Andrew W. (2006). A compositional logic for control flow. *Pages 80–94 of: International Conference on Verification, Model Checking and Abstract Interpretation, VMCAI'06.* Lecture Notes in Computer Science, vol. 3855.

Wadler, Philip. (1998). The marriage of effects and monads. *Pages 63–74 of: International Conference on Functional Programming, ICFP'98.*

Watkins, Kevin, Cervesato, Iliano, Pfenning, Frank, & Walker, David. (2004). A concurrent logical framework: The propositional fragment. *Pages 355–377 of:* Berardi, S., Coppo, M., & Damiani, F. (eds), *Types for proofs and programs.* Lecture Notes in Computer Science, vol. 3085. Springer.

Westbrook, Edwin, Stump, Aaron, & Wehrman, Ian. (2005). A Language-based Approach to Functionally Correct Imperative Programming. *Pages 268–279 of: International Conference on Functional Programming, ICFP'05.*

Xi, Hongwei, & Pfenning, Frank. 1998 (June). Eliminating array bound checking through dependent types. *Pages 249–257 of: Conference on Programming Language Design and Implementation, PLDI'98.*

Xi, Hongwei, & Pfenning, Frank. 1999 (January). Dependent types in practical programming. *Pages 214–227 of: Symposium on Principles of Programming Languages, POPL'99.*

Zhu, Dengping, & Xi, Hongwei. (2005). Safe programming with pointers through stateful views. *Pages 83–97 of: Practical Aspects of Declarative Languages, PADL'05.* Lecture Notes in Computer Science, vol. 3350. Long Beach, California: Springer.