A gentle introduction to Hoare Type Theory

Hyeyoung Shin
hyeyoung@iastate.edu

Iowa State University

6 July 2016

Hoare Type Theory, Polymorphism and Separation (2007)

Nanevski, Morrisett and Birkedal

Problem

Errors not caught by today's type systems

- array-index-out-of-bounds
- division-by-zero

and high-level correctness issues

- invariants
- protocols on mutable data structures

Alternative Solutions

1. dependent types

2. program logics
   - Hoare Logic
   - Separation Logic

Hoare Logic

Hoare Triple: $\{P\}C\{Q\}$

A reasoning system that can carry out *program verification* well suited to imperative programs

- a natural way of writing down specifications of programs
- a *compositional proof technique*

Separation Logic

An extension of Hoare's logic for specifying and verifying properties of dynamically-allocated linked data structure

- much simpler by-hand specification and program proofs

Specifications are "small": it concentrates on the resources relavant to its correct operations (its "footprint")

More specifically, during its execution, c may access only memory locations whose existence is asserted in the precondition or that have been allocated by c itself

- *local reasoning*

$P * Q$: Separating conjunction

P and Q hold for separate portions of memory and program-proof rules that exploit separation to provide modular reasoning about programs

- *local reasoning*

In addition to the standard Hoare logic rules, Separation logic supports the frame rule:

$$\frac{\{P\}C\{Q\}}{\{P * R\}C\{Q * R\}} \ \mathrm{mod}(C) \cap \mathrm{free}(R) = \emptyset$$

“none of the variables modified by C occur free in R”

Limitation of Previous work

1. dependent types
   - do not work well with side effects (state updates, non-termination)
2. program logics
   - do not integrate *into* the type system
   - make it difficult to scale the logics to support the abstraction mechanisms (higher order functions, polymorphism, modules)

Solution

Hoare Type Theory: Types with Hoare-style specifications

The Hoare Type

The key mechanism: Hoare Type constructor $\Psi.X.\{P\}x : A\{Q\}$

- Simultaneously isolates and describes the effects of imperative commands
- keeps track of the effectful computation in the dependent setting
  - refinement of the concept of *monad*
  - "type marker" for computations with side effects
- can be nested, combined with other types, and abstracted within terms, types and predicates
- provides a unified system for programming, specification and reasoning about programs

Contents of the talk

1. Overview
2. Syntax
   - Terms
   - Computations
   - Types
   - Heaps and locations
   - Assertions
3. Type System
   - Equational reasoning
   - Typing rules
4. Hereditary substitutions
5. Properties
6. Operational semantics
   - Preservation
   - Progress
7. Heap soundness

1. Overview of HTT programs

- the pure fragment
  - higher-order functions
  - constructs for type polymorphism
- the impure fragment
  - allocation
  - lookup
  - strong update
  - deallocation of memory
  - conditional
  - loops (recursion)

the Hoare type $\{P\}x : A\{Q\}$ classifies the impure programs

Example: Small and large footprints

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

- y must be fresh because $\{\text{emp}\}$ prohibits alloc form working with existing locations

Example: Small and large footprints

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

- z and v denote the assumed existing location and its contents
- alloc$'$ allows execution only in heaps with *at least* one boolean location
- The specification insists that the contents of z is not changed by the execution
- $*$ specifies that z and v belong to disjoint heap portions, hence y is fresh

Example: Small and large footprints

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

- $\mathsf{this}(h) = \mathsf{hid}(\mathsf{mem}, H)$ "heap equality"
- We can name the heap encountered before allocation using a ghost variable h

Example: Small and large footprints

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

- classical style (large footprint)

Example: Small and large footprints

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

- classical style (large footprint)
- By explicitly naming various heap fragments with ghost variables, HTT can freely switch from the small and large footprint specifications

- small specificaiton: convenient for programming
- large footprint: convenient in case aliasing is allowed
- naming of heap fragment: easy to connect to the assertion logic in HTT

Example: Diverging computation

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

- non-termination is an effect (impure computation)
- do E: intro term for the Hoare type which encapsulates the effectful computation E and suspends its evaluation
- so **diverge**, when forced, sets up a recursive function f(y:1) = do (eval f(y)).

2. Syntax

Terms: the purely functional fragment of HTT

The separation into **intro** and **elim** terms facilitate *bidirectional typing checking* (Pierce & Turner. 2000)

Computations: the effectful fragment of HTT

Semicolon-separated lists of commands, terminated with a return type
The commands are executed in the order of the list and usually bind their result to a variable
Variables in HTT are immutable unlike those of Hoare logic

Commands

Types

- the primitive types: booleans and nats
- unit type: 1
- the Hoare type: $\Psi.X.\{P\}x : A\{Q\}$
- dependent function type: $\Pi_{x:A}B(x)$
- polymorphic type: $\forall\alpha.A$

The Hoare Type: $\Psi.X.\{P\}x{:}A\{Q\}$

- specifies an effectful computation with precondition P and postcondition Q returning a result of type A
- x: the name of the return type
- Q may depend on x
- Ψ lists the variables
- X lists the heap variables (ghost variables: only appear in the assertions, not in the program)

Dependent function type: $\Pi x{:}A.B$

a.k.a forall type or product type

$$f : A \to \bigcup_{a:A} B(a) \quad \text{and} \quad \forall a \in A. f(a) \in B(a)$$

This is equivalent to the more concise notation

$$f : \prod_{a:A} B(a)$$

For example, if $A = \{0, 1, 2\}$, then $\Pi x{:}A.B(x) = B(0) \times B(1) \times B(2)$.

Polymorphic type: $\forall\alpha.A$

polymorphically quantifies over the monotype variable α

- *monotype* in HTT: any type that does not contain polymorphic quantification, except in assertions $\Rightarrow$ *predicative* polymorphism (range of type variable is restricted to monotypes)

Extending HTT to support *impredicative* polymorphism is left for future work since it significantly complicates the termination argument for normalization (e.g. X in T $= \forall X.X \to X$ ranges over all types, including T itself)

Heaps and locations

Memory locations as natural number to support pointer arithmetic
Heaps as finite functions

$N \longrightarrow (\tau, M)$

"N points to M" or "M is the contents of location N"

- empty
- upd_τ (H, M, N)

Assertions: from classical first order logic

- $id_A(M, N)$
- seleq$_\tau(H, M, N)$: the heap H at address M contains a term N:τ
- $P \supset\subset Q$: $P \supset Q \wedge Q \supset P$
- hid(H_1, H_2) : the heap equality
- $M \in H$: M assigns the location M
- $M \notin H$
- share(H_1, H_2, M) : H_1 and H_2 agree on the location M
- splits(H, H_1, H_2) : H can be split into disjoint heaps H_1 and H_2

Assertions: from Separation logic

Free variable **mem** denotes the current heap fragment

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

3.Type System

Type checking judgements require testing if two terms are definitionally equal and testing definitional equality involves normalization.

- Two terms are equal only if the canonical forms are syntactically the same, modulo α-conversion
- Unusual in HTT: normalization is undertaken *simultaneously* with type checking

3.1 Equational reasoning

Beta and eta equalities are most important in HTT definitional equality
What's Unusual in HTT? Normalization is undertaken *simultaneously* with type checking

Equations for

- $\Pi x{:}A.B$

$$(\lambda x.M : \Pi x{:}A.B)N \longrightarrow_\beta [N{:}A/x]M$$
$$K \longrightarrow_\eta \lambda x.Kx \quad x \notin FV(K)$$

- $\forall \alpha.A$

$$(\Lambda \alpha.M : \forall \alpha.A)\tau \longrightarrow_\beta [\tau/\alpha]M$$
$$K \longrightarrow_\eta \Lambda \alpha.K\alpha \quad \alpha \notin FTV(K)$$

- 1

$$K \longrightarrow_\eta ()$$

Equations for the Hoard type

requires an auxiliary operation of monadic substitution

$$\langle E/x{:}A\rangle F,$$

which subsequently composes the computations E and F, " E followed by F", and free variable $x{:}A$ in F is bound to the value of E.

Monadic substitution is defined by induction on the structure of E.

$$x \longleftarrow (\mathrm{do}E : \{P\}x{:}A\{Q\}); F \longrightarrow_{\beta} \langle E/x{:}A\rangle F$$

$$K \longrightarrow_{\eta} \mathrm{do}(x \longleftarrow K; return x)$$

Normalization algorithm

uses specific expand (*expand*$_A$) and reduction (*hereditary substitution*) strategy.

- expand$_A$ iterates over the type A and expands the given argument (elim or intro term) according to each encountered type constructor
- hereditary substitution operates on canonical terms only (e.g. whenever an ordinary capture avoiding substitution creates a redex like $(\lambda x.M)N$, hereditary substitution continues by immediately substituting N for x in M)

Normalization algorithm

uses specific expand (*expand*$_A$) and reduction (*hereditary substitution*) strategy.

- expand$_A$ iterates over the type A and expands the given argument (elim or intro term) according to each encountered type constructor
- hereditary substitution operates on canonical terms only (e.g. whenever an ordinary capture avoiding substitution creates a redex like $(\lambda x.M)N$, hereditary substitution continues by immediately substituting N for x in M)

3.2 Typing rules

Undecidable

1. basic type-checking and verification-condition generation
   - completely automatic
2. validate the generated verification-conditions
   - can be fed into an automated theorem prover
   - can be discharged by hand
   - can be ignored

HTT vs. dependent types

- When HTT was invented, some 10 years ago, dependent types were generally considered as being limited to the purely-functional and terminating programming model.
- So it was a bit of a surprise that we could combine dependent types with dynamic state (i.e., pointers), and general recursion. (Amazingly, nobody had tried the idea before, in the dependently-typed setting.)
- Even these days, when people do Hoare logic in a proof assistant such as Coq, most of the time they use deep embedding, which, while certainly expressive, results in proofs that are consider to be torture.

HTT vs. Hoare logic (or Separation logic)

- At the time HTT was invented, Separation logic was a first-order theory, incapable of reasoning about function calls, let alone abstract types, modules, objects, callbacks, etc.
- All of these come for free in HTT, as they are all instances of the Π and Σ types. (see ICFP'08 paper on the implementation of HTT, for a number of examples that are higher-order in this sense, and use the combination of dependent types with pre- and post-conditions, in an essential way)
- Moreover, stack variables in Hoare logic are quite a complication, which is avoided in HTT, by means of the monadic formulation.
- Thus, monadic formulation made proofs much easier, as it eliminated pesky sidevconditions related to stack variables.
- Since then, higher-order versions of separation logic have been formulated, not by using dependent types, but still using monads.
- They are basically on the spectrum between ordinary separation logic and HTT, i.e., HTT is the logical limit of the idea.

HTT!

It's usefulness lies in being able to structure your proofs in an easy and natural way, rather than in what can be technically done in it, compared to other systems.

6. Operational semantics

call-by-value, left-to-right operational semantics

- if $\Delta; P \vdash E \Leftarrow x{:}A.Q$ is derivable in the type system, then it is the case that evaluating E in a heap in which P holds produces a heap in which Q holds (if E terminates).
- only defined for well-typed terms

Syntax

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

Syntax

$$
\begin{array}{lll}
\textit{Values} & v, l & ::= \ (\,) \mid \lambda x.\ M \mid \Lambda\alpha.\ M \mid \mathsf{do}\ E \mid \mathsf{true} \mid \mathsf{false} \mid \mathsf{z} \mid \mathsf{succ}\ v \\
\textit{Value heaps} & \chi & ::= \ \cdot \mid \chi, l \mapsto_\tau v
\end{array}
$$

- **values**: we use l to range over nats when they are used as pointers
- **value heaps**: assignments from nats to values, where each assignment is indexed by a type
  - run-time concept
  - used in the evaluation judgements to describe the state in which programs execute
  - note that heaps mentioned earlier are different and used for reasoning in the assertion logic
  - two notions of heaps correspond to each other $\Rightarrow$ *heap soundness*

Syntax

$$
\begin{array}{llll}
\textit{Continuations} & \kappa & ::= & \cdot \mid x{:}A.\ E; \kappa \\
\textit{Control expressions} & \rho & ::= & \kappa \triangleright E
\end{array}
$$

- **continuation**: sequence of computations of the form $x{:}A.E$, where E may depend on the bound variable $x{:}A$
- **control expression**: pairs up a computation E and a continuation k so that E provides the initial value with which the execution k can start
  - self-contained computation
  - makes the call-by-value semantics of the computation $x \leftarrow E; F$ explicit
  - evaluation of this computation is carried out by creating the control expression $x.F \triangleright E$;
    "push $x.F$ on to the continuation and proceed to evaluate E"

Syntax

$$\textit{Abstract machines} \quad \mu \quad ::= \quad \chi, \kappa \triangleright E$$

- **abstract machine**: a pair of a value heap X and a control expression $k \triangleright E$
  - evaluated against the heap, to eventually produce a result and possibly change the heap

Evaluation

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

Evaluation of abstract machines

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

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

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

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

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

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

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

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

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

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

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

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

Soundness

via Preservation and Progress theorems

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. By induction on the evaluation judgement, using inversion on the typing derivation, and substituting principles

Theorem 10 (*Progress*)
Suppose that the assertion logic of HTT is heap sound. Then the following hold.

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

Proof. By straightforward case analysis on the involved expressions, employing inversion on the typing derivations, and the proof of the third statement requires heap soundness

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.

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

Proof. refer to the paper