HOARE TYPE THEORY: DEPENDENT TYPES FOR STATE

LECTURE AND NOTES BY JAMES T. PERCONTI
MATERIAL TAKEN FROM WORK BY NANEVSKI ET AL.

In this lecture, we'll discuss Hoare Type Theory, a powerful system that lets us write down and enforce precise specifications for stateful, higher-order programs. It integrates ideas from Hoare Logic into a dependent type system. The system for Hoare Type Theory is fairly complex, so let's begin by reviewing separately some of the ideas from dependent types and Hoare logic.

1. Dependent Types

Dependent types let us write down precise specifications for purely functional programs. For example, suppose we want to write a function on natural numbers that always returns a result greater than its argument. In a dependent type system, we can write a type that specifies exactly this:

$$\Pi m\!:\!\mathtt{nat}.\{n\!:\!\mathtt{nat}.n > m\}$$

$\Pi x\!:\!A.B$ is the dependent function type: it describes a function that takes an argument x of type A, and returns a value of type B. x can appear in B: for instance, in a refinement such as $\{y\!:\!A.y = x\}$. A refinement type, or subset type, $\{x\!:\!A.P\}$ is inhabited by terms of type A that also satisfy the proposition P.

We also have a dependent pair type $\Sigma x\!:\!A.B$. The type of the second component of the pair can depend on the value of the first component. That is, the first component has type A, and if it is a value v, then the second component will have type $B[v/x]$. For example, consider this type:

$$\Sigma x\!:\!\mathtt{nat}.\{y\!:\!\mathtt{nat}.y = x + 5\}.$$

This type is inhabited by values like $\langle 3, 8\rangle$ and $\langle 1, 6\rangle$, where the second component is 5 more than the first.

We abbreviate $\Pi x\!:\!A.B$ and $\Sigma x\!:\!A.B$ using the more usual notations $A \to B$ and $A \times B$, respectively, when the dependency is not being used ($x \notin \mathrm{fv}\, B$).

Finally, a dependent type system can treat propositions and types as values and perform computation with them! In the system we will consider today, we will have a simple two-level stratification of types: types that appear in terms are called "monotypes" or "small types" and can't quantify over other types. The "large types" that are not used in computation can quantify over monotypes, giving them type `mono`.

We can do pretty crazy things in this system. For instance, we can have pairs containing a type and a value of that type:

$$\langle \mathtt{nat}, 4\rangle \;:\; \Sigma\alpha\!:\!\mathtt{mono}.\alpha$$

Here is the syntax of the dependent type system we'll work with:

$$\begin{array}{rll}
A, B, C & ::= K \mid \mathtt{nat} \mid \mathtt{bool} \mid \mathtt{prop} \mid \mathtt{1} \mid \mathtt{mono} \mid \Pi x\!:\!A.B \mid \Sigma x\!:\!A.B \mid \{x\!:\!A.P\} & \textit{Types} \\
K, L & ::= x \mid K\ N \mid \mathtt{fst}\, K \mid \mathtt{snd}\, K \mid \mathtt{out}\, K \mid M\!:\!A & \textit{Elim Terms} \\
M, N, P, Q, R, \tau, \sigma & ::= K \mid n \mid \mathtt{true} \mid \mathtt{false} \mid () \mid \lambda x.M \mid \langle M, N\rangle \mid \mathtt{in}\, M & \textit{Intro Terms} \\
& \quad \mid M + N \mid M \times N \mid \mathtt{eq}_{\mathtt{nat}}(M, N) & \\
& \quad \mid \mathtt{xid}_{A,B}(M, N) \mid \top \mid \bot \mid P \wedge Q \mid P \vee Q \mid P \supset Q \mid \neg P \mid \forall x\!:\!A.P \mid \exists x\!:\!A.P & \\
& \quad \mid \mathtt{nat} \mid \mathtt{bool} \mid \mathtt{prop} \mid () \mid \Pi x\!:\!\tau.\sigma \mid \Sigma x\!:\!\tau.\sigma \mid \{x\!:\!\tau.P\} & \\
\Delta & ::= \cdot \mid \Delta, x\!:\!A, \Delta, P & \textit{Contexts}
\end{array}$$

We've touched on some of the constructs here already, but there is still plenty to talk about. First, note that large types are notated with the metavariables A, B, C, while small types are typically written or τ or σ. In the syntax of terms, we have so many metavariables in order to suggest whether we have a small type, a proposition (P, Q, R), or the more usual λ-calculus sort of term (M, N), but there is actually no distinction between them as far as the syntax is concerned. Any of these three kinds of terms can appear as the others. (Obviously, the type system will distinguish them.)

This brings us to the next observation: terms are explicitly split into introduction forms and elimination forms. This allows *bidirectional typechecking*, a technique that makes it explicit when we can and cannot synthesize types. Specifically, we can always generate the type of elim terms, so their typing judgment reflects this:

$$\Delta \vdash K \Rightarrow A$$

Intro terms, on the other hand, don't contain enough information in them to construct the correct type, so we only check them against a given type, as indicated by the shape of their typing judgment:

$$\Delta \vdash M \Leftarrow A$$

To make sure we're all comfortable with this and some of the other concepts of dependent types we've already touched on, let's write down some of the typing rules. Here are the rules for introduction and elimination of dependent functions:

$$\frac{\Delta, x : A \vdash M \Leftarrow B}{\Delta \vdash \lambda x.M \Leftarrow \Pi x : A.B} \qquad \frac{\Delta \vdash K \Rightarrow \Pi x : A.B \qquad \Delta \vdash M \Leftarrow A}{\Delta \vdash K\ M \Rightarrow B[M/x]}$$

Note that the premise of the $\lambda x.M$ rule puts x into the environment for both the function body M and its type B. The application rule handles the dependency by directly substituting the argument supplied into the type of the expression.

Here are the rules for introduction and elimination of dependent pairs:

$$\frac{\Delta \vdash M \Leftarrow A \qquad \Delta \vdash N \Leftarrow B[M/x]}{\Delta \vdash \langle M, N \rangle \Leftarrow \Sigma x : A.B} \qquad \frac{\Delta \vdash K \Rightarrow \Sigma x : A.B}{\Delta \vdash \mathtt{fst}\, K \Rightarrow A} \qquad \frac{\Delta \vdash K \Rightarrow \Sigma x : A.B}{\Delta \vdash \mathtt{snd}\, K \Rightarrow B[\mathtt{fst}\, K/x]}$$

The dependency appears in both the premise of the intro rule and the result type of the `snd` rule.

Next, let's look at the rules for refinements. We use the intro form $\mathtt{in}\, M$ and the elim form $\mathtt{out}\, M$ as syntactic markers for refinements.

$$\frac{\Delta \vdash M \Leftarrow A \qquad \Delta \vDash P[M/x]}{\Delta \vdash \mathtt{in}\, M \Leftarrow \{x : A.P\}} \qquad \frac{\Delta \vdash K \Rightarrow \{x : A.P\}}{\Delta \vdash \mathtt{out}\, K \Rightarrow A} \qquad \frac{\Delta \vdash K \Rightarrow \{x : A.P\}}{\Delta \vDash P[\mathtt{out}\, K/x]}$$

The introduction rule here requires us to prove that the proposition P actually applies to the term M. Whenever we have a proof obligation, or *verification condition*, like this, we can pass it to an arbitrary theorem prover (or even ignore it altogether!). If we don't prove all the verification conditions, the type system is decidable, and still gives us a type safety guarantee similar to what a simple type system would give us. To ensure the precise specifications of our dependent type system are met, we do need to prove them (which is undecidable in general).

This separation between the decidable and undecidable components of our type system is also important in other places. For instance, we need to be able to reason about equality of terms. Our language of propositions includes a heterogeneous equality predicate $\mathtt{xid}_{A,B}(M, N)$, which is true when A and B are the same type and M and N are the same term and have that type. This is our propositional, undecidable notion of equality.

We also need a decidable notion of equality (we directly need it to compare types, as we'll see momentarily, but terms can appear in types, so we need to compare terms as well). To get this, we reduce terms to normal form and then compare them syntactically. This works because our language is terminating, but setting it up correctly requires a lot of machinery, threaded through the entire

type system, that we will be leaving out. To give an idea of what is involved, here is the real type rule for applications:

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

The bracketed term at the end of each judgment is the normalized form of the term that was input. The apply metafunction normalizes the application itself. The substitution performed by apply and the substitution in the synthesized type $B[M'/x]^a_A$ are *hereditary substitutions*. A hereditary substitution must renormalize any redexes generated by the substitution — which could in turn require more hereditary substitutions. It's a lot of work just to write down the definition of this operation, let alone to prove that it terminates! We won't worry about the details of normalization from here on, but it's important to note that it is going on behind the scenes.

The decidable equality we've been discussing, known as *definitional equality*, is used when we want to change the direction of typechecking. When an elim term appears in an intro position, we need to compare the type it synthesizes to the type we are checking. On the other hand, since intro terms can only appear in elim positions with explicit type annotations, we don't have to worry about equality in that rule:

$$\frac{\Delta \vdash K \Rightarrow A \qquad A = B}{\Delta \vdash K \Leftarrow B} \qquad\qquad \frac{\Delta \vdash A\ \mathtt{type} \qquad \Delta \vdash M \Leftarrow A}{\Delta \vdash M\!:\!A \Rightarrow A}$$

OK, that's enough about the details of our dependent type system. By now at least one reason that adding effects to this language is problematic should be clear: effects destroy our ability to decide equality by normalizing terms, most obviously by giving us terms that diverge. To solve this, we'll encapsulate our effects in a monad based on another model for program specifications. That model is Hoare logic, a reasoning system designed for imperative programs.

2. Hoare Logic

Our goal is to examine the core idea of Hoare logic and integrate it into our dependent type system, so we won't go into much detail about Hoare logic as an independent entity. Essentially, we specify the behavior of a computation, or effectful program fragment, by writing a *Hoare triple*

$$\{P\}\ E\ \{Q\}.$$

Here E is the computation for which we are giving the specification. P is a precondition, an assertion giving requirements for the state of the program before E is run. Q is a postcondition, an assertion that describes the state of the program after E completes, assuming P was satisfied before E began to be computed.

For a simple example, let's write a program fragment that increments the value at some location ℓ (assumed to be in the heap already):

$$u = !\ell;\ \ell := u + 1$$

We can give it the following specification:

$$n\!:\!\mathtt{nat}.\{\ell \mapsto n\}\ u = !\ell;\ \ell := u + 1\ \{\ell \mapsto n + 1\}$$

The precondition says that ℓ maps to some natural number n in the heap, and in the postcondition ℓ will map to $n + 1$. The variable n that appears in both the precondition and postcondition, but not in the computation itself, is called a *logic variable* or *ghost variable*. Individual Hoare triples are often easier to read using logic variables, but it is easier to work with the metatheory without them. Fortunately, we will see in a moment that the binding structure implied by logic variables can be encoded in a system where the precondition and postcondition are self-contained.

We're making our way toward integrating Hoare triples into a type system, so let's separate the term from its specification and replace it with the type of value returned by the computation. Thus

we get a *Hoare type*, which has the form

$$\{P\}\ x\!:\!A\ \{Q\}.$$

To talk about the state of the program before the coputation is run, the precondition needs to be given the initial heap. Since it must relate the state before the computation to the state after, the postcondition needs the initial and final heaps. Thus P and Q should be propositions of type $\mathsf{heap} \to \mathtt{prop}$ and $\mathsf{heap} \to \mathsf{heap} \to \mathtt{prop}$ respectively. The postcondition can also depend on the return value x of the computation.

The Hoare type for our example program is

$$u = !\ell;\ \ell := u + 1\ :\ n\!:\!\mathtt{nat}.\{\ell \mapsto_{\mathtt{nat}} n\}\ r\!:\!\mathtt{1}\ \{\ell \mapsto_{\mathtt{nat}} n + 1\}.$$

Now let's define our encoding of ghost variables in Hoare types:

$$x\!:\!A.\{P\}\ y\!:\!B\ \{Q\} \stackrel{\text{def}}{=} \{\lambda i.\exists x\!:\!A.P\ i\}\ y\!:\!B\ \{\lambda i.\lambda m.\forall x\!:\!A.(P\ i \supset Q\ m)\}$$

In the precondition, x is simply quantified as an existential, while in the postcondition, any value of type A that satisfied P under the initial heap should satisfy Q under the final heap. Thus the expanded type for our example is

$$\{\lambda i.\exists n\!:\!\mathtt{nat}.(\ell \mapsto_{\mathtt{nat}} n)\ i\}\ r\!:\!\mathtt{1}\ \{\lambda i.\lambda m.\forall n\!:\!\mathtt{nat}.(\ell \mapsto_{\mathtt{nat}} n)\ i \supset (\ell \mapsto_{\mathtt{nat}} n + 1)\ m\}.$$

Note that the final precondition and postcondition have the correct types, assuming that the operation $x \mapsto_\tau v$ (which we will define later) has type $\mathsf{heap} \to \mathtt{prop}$.

3. Hoare Type Theory

We will now add effectful computations to our dependently typed language. A computation E is encapsulated by the `do` construct, which suspends its evaluation until another computation forces it to run. Computations can dereference and assign to memory locations, run other computations, branch, return a pure value, and run recursive (possibly divergent) functions. To classify these new terms, we also add the Hoare type, as developed in the previous section. A computation of type $\{P\}\ x\!:\!A\ \{Q\}$ can be run in a heap i such that $P\ i$ holds, and will either diverge or reach a heap m and return a value $v\!:\!A$ such that $Q[v/x]\ i\ m$ holds.

Here is the full set of additions to our language. Note that the constructs $\mathtt{return}\,M$ and $x = c;\ E$ correspond to monadic unit and bind operations, respectively.

$$\begin{array}{rll}
A, B, C & ::= \cdots \mid \{P\}\ x\!:\!A\ \{Q\} & \textit{Types} \\
M, N, P, Q, R, \tau, \sigma & ::= \cdots \mid \mathtt{do}\,E \mid \{P\}\ \sigma\!:\!\tau\ \{Q\} & \textit{Intro Terms} \\
c & ::= !_\tau\,M \mid M :=_\tau N \mid \mathtt{if}_A\,M\,\mathtt{then}\,E_1\,\mathtt{else}\,E_2 & \textit{Commands} \\
 & \quad \mid \mathtt{fix}\,x(y\!:\!A)\!:\!B = \mathtt{do}\,E\,\mathtt{in}\,\mathtt{eval}\,x\ M & \\
E, F & ::= \mathtt{return}\,M \mid \mathtt{run}\ x = K\,\mathtt{in}\,E \mid x = c;\ E & \textit{Computations}
\end{array}$$

The type of our example program is already correctly formed, but for completeness, note that the program itself should be adjusted slightly to match this syntax:

$$\mathtt{do}(u = !_{\mathtt{nat}}\ \ell;\ z = (\ell :=_{\mathtt{nat}} u + 1);\ \mathtt{return}\,z).$$

With these additions to the language, we now have a complete system for Hoare type theory. It combines the strengths of dependent types and Hoare logic. Everything we need to write many useful programs and specifications can be encoded from the small set of primitives we have. For instance, let's develop an encoding for the type heap itself.

We model a heap as a set of locations paired with the values they map to. We can encode a set of terms of type A as an $A \to \mathtt{prop}$ that returns a true proposition when its argument is in the set. To know what type of value we have at a location, we will just add the type itself as a component of the tuple. Finally, we use a refinement to require that the heap is finite and only maps each location at most once. Thus the full definition of the type of a heap is

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

Properties like Finite and Functional are straightforward to define, though they in turn require some more definitions. The definitions of $\leq$, $\in$, Injective, and Surjective are left as exercises to the reader (or you can go look in the paper).

$$\begin{aligned}\mathsf{Finite}_A &: (A \to \mathtt{prop}) \to \mathtt{prop} \\ &= \lambda x.\exists n:\mathtt{nat}.\exists f:\{y:\mathtt{nat}.y \leq n\} \to \{z:A.x\ z\}.\ \mathsf{Injective}(f) \wedge \mathsf{Surjective}(f) \\ \mathsf{Functional}_{A,B} &: (A \times B \to \mathtt{prop}) \to \mathtt{prop} \\ &= \lambda R.\forall x:A.\forall y_1,y_2:B.(\langle x,y_1\rangle \in R \wedge \langle x,y_2\rangle \in R) \supset \mathtt{xid}_{B,B}(y_1,y_2)\end{aligned}$$

Now that we have our representation for heaps, we'll want some operations in order to work with them. Here are just a couple:

$$\begin{aligned}\mathsf{empty} &: \mathsf{heap} \\ &= \mathtt{in}\,(\lambda h.\bot) \\ \mathsf{upd} &: \Pi\alpha:\mathtt{mono}.\mathsf{heap} \to \mathtt{nat} \to \alpha \to \mathsf{heap} \\ &= \lambda\alpha.\lambda h.\lambda n.\lambda x.\ \mathtt{in}\,(\lambda t.(n = \mathtt{fst}\,t \supset \mathtt{snd}\,t = \langle\alpha,x\rangle) \wedge (n \neq \mathtt{fst}\,t \supset (\mathtt{out}\,h)\ t)) \\ \mathsf{seleq} &: \Pi\alpha:\mathtt{mono}.\mathsf{heap} \to \mathtt{nat} \to \alpha \to \mathtt{prop} \\ &= \lambda\alpha.\lambda h.\lambda n.\lambda x.(\mathtt{out}\,h)\ \langle n,\langle\alpha,x\rangle\rangle \\ \mapsto &: \Pi\alpha:\mathtt{mono}.\mathtt{nat} \to \alpha \to \mathsf{heap} \to \mathtt{prop} \\ &= \lambda\alpha.\lambda n.\lambda x.\lambda h.\ \mathtt{xid}_{\mathsf{heap},\mathsf{heap}}(h, \mathsf{upd}\,\alpha\,\mathsf{empty}\,n\,x)\end{aligned}$$

empty is of course the empty heap, and upd is used to extend or update a heap with a new value. seleq takes a heap, a location, a value, and that value's type, and returns the proposition that that value is in the heap at that location. $\mapsto$ is similar in that it takes a heap entry (location, value, and type) and a heap and returns a proposition about that entry's presence in the heap, but it matches the given heap to one containing only the supplied entry.

Does the definition of $\mapsto$ match our intuition from its use in the example? What if we want to run that program in a heap that contains entries besides ℓ? Actually, we don't want to have to worry about other things in the heap, since our program doesn't touch them or care about them. We don't have to worry about them because the precondition and postcondition of the Hoare type are implicitly *small-footprint* specifcations. This means they only need to describe the fragment of the heap that is actually affected by the computation being described. This fragment is assumed to be combined with any *frame*, or "rest of the heap," that is disjoint from it. Thus, a precondition $\{x \mapsto v\}$ accepts heaps that also contain locations other than x, but only as long as none of them is aliased to x, since the frame must be disjoint from the part of the heap described by the specification.

4. Local State

We will now develop another example to show how Hoare type theory can hide the internal state invariants of a program. Let's start with a fairly simple computation that builds a counter:

$$\mathsf{make\_counter} = \mathtt{do}\,(\mathtt{run}\ x = (\mathsf{alloc}\ \mathtt{nat}\ 0)\ \mathtt{in}\ \mathtt{do}\,(z =!_{\mathtt{nat}} x; x :=_{\mathtt{nat}} z+1; \mathtt{return}\,z+1)).$$

Here, alloc is not something we've defined yet, but we can encode an allocator without too much trouble. Assume we have an allocator that relies on some invariant $I:\mathsf{heap} \to \mathtt{prop}$, and that the allocation function has the following type:

$$\mathsf{alloc} : \Pi\alpha:\mathtt{mono}.\Pi x:\alpha.\{I\}\ r:\mathtt{nat}\ \{\lambda i.(I * r \mapsto_\alpha x)\}$$

In the postcondition, the $*$ operation is a disjoint union of heap fragments.

So: make_counter allocates a new reference and returns a computation that will increment that reference and return its new value whenever it is called. Let's try to write down the type of this

program:

$$\begin{aligned}&\mathsf{make\_counter} : \{I\}\\ &\qquad y : (v : \mathtt{nat}.\{x \mapsto_{\mathtt{nat}} v\}\ r : \mathtt{nat}\ \{\lambda m.(x \mapsto_{\mathtt{nat}} v+1)\ m \wedge r = v+1\})\\ &\qquad \{\lambda i.(I * x \mapsto_{\mathtt{nat}} 0)\}\end{aligned}$$

This type isn't quite right, because we've left x unbound. But ignoring that for the moment, note that the outer computation just calls alloc and returns a value, so its precondition and postcondition essentiallty match alloc's, while the value returned is a computation that increments and returns the value of x.

How should x be bound in this type? Ideally, we don't want it to show up at all, since we only have access to the allocated location inside the counter. As long as they are correct, different implementations of a counter should be indistinguishable. For example, what if we replace $\mathsf{make\_counter}$ with

$$\begin{aligned}&\mathsf{do}\,(\mathtt{run}\ x = \mathsf{alloc}\ \mathtt{nat}\ 0\ \mathtt{in}\\ &\quad \mathtt{run}\ y = \mathsf{alloc}\ \mathtt{nat}\ 0\ \mathtt{in}\\ &\quad \mathsf{do}\,(z =!_{\mathtt{nat}} x; w =!_{\mathtt{nat}} y; x :=_{\mathtt{nat}} z+1; y :=_{\mathtt{nat}} w+1; \mathtt{return}(z+w)/2+1))?\end{aligned}$$

This version of a counter is silly, but clearly equivalent except in the way it locally uses state. We should be able to give different implementations the same type, even if they use local state differently.

To make our local state invariants abstract, we can modify our counter implementations to return the invariant itself. Note that there is no way to inspect a proposition at runtime, so passing the invariant cannot change the computational behavior of the program, but only allows the type to abstract over it.

$$\begin{aligned}\mathsf{counter}' = \mathsf{do}\,&(\mathtt{run}\ x = (\mathsf{alloc}\ \mathtt{nat}\ 0)\ \mathtt{in}\\ &\langle(\lambda v.x \mapsto_{\mathtt{nat}} v), \mathsf{do}\,(z =!_{\mathtt{nat}} x; x :=_{\mathtt{nat}} z+1; \mathtt{return}\ z+1)\rangle)\\ \mathsf{counter2}' = \mathsf{do}\,&(\mathtt{run}\ x = \mathsf{alloc}\ \mathtt{nat}\ 0\ \mathtt{in}\\ &\mathtt{run}\ y = \mathsf{alloc}\ \mathtt{nat}\ 0\ \mathtt{in}\\ &\langle\lambda v.(x \mapsto_{\mathtt{nat}} v) * (y \mapsto_{\mathtt{nat}} v),\\ &\quad \mathsf{do}\,(z =!_{\mathtt{nat}} x; w =!_{\mathtt{nat}} y; x :=_{\mathtt{nat}} z+1; y :=_{\mathtt{nat}} w+1; \mathtt{return}(z+w)/2+1)\rangle)\end{aligned}$$

With this change, x is now well-scoped, as the invariant will get instantiated with the correct location once the counter is initialized. We can write the shared type of the two counters without referring to x (or y), as follows:

$$\begin{aligned}&\mathsf{counter}', \mathsf{counter2}' : \{I\}\\ &\qquad t : \Sigma inv : \mathtt{nat} \to \mathsf{heap} \to \mathtt{prop}.\\ &\qquad\quad v : \mathtt{nat}.\{inv\ v\}\ r : \mathtt{nat}\ \{\lambda h.inv\ (v+1)\ h \wedge r = v+1\}\\ &\qquad \{\lambda i.(I * \mathtt{fst}\ t\ 0)\}\end{aligned}$$

Though our example of a counter is contrived, we can apply the same idea to get abstract handling of local state in many other situations.

5. Typing Rules for HTT

We've gotten a taste of the kinds of specifications Hoare type theory lets us encode, so now let's see how the system manages to do all this. We add a pair of judgments for computations to the

dependent type system we had earlier:

$$\Delta; P \vdash E \Leftarrow x : A.Q$$
$$\Delta; P \vdash E \Rightarrow x : A.Q$$

The first judgment checks if a computation E, under environment Δ and precondition P, will return a value of type A and leave the heap in a state that satisfies the postcondition Q. The second judgment still takes Δ, P, E, and A as inputs, but it synthesizes the postcondition. In particular, it synthesizes the strongest postcondition Q that could result from running E in a heap that satisfies P. the checking judgment has one inference rule, so let's look at it:

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

To check a computation against a given postcondition Q, we synthesize its strongest postcondition S using the $\Rightarrow$ judgment, and then we have a proof obligation that S implies Q under an appropriate environment.

To see how the other judgment works, let's start with the simplest case: the computation that just returns a pure value.

$$\frac{\Delta \vdash M \Leftarrow A}{\Delta; P \vdash \mathtt{return}\, M \Rightarrow x : A.(\lambda i.\lambda m.(P\ i\ m) \wedge \mathtt{xid}_{A,A}(x, M))}$$

The postcondition we generate is the precondition, but with an additional requirement that specifies the value returned. The only premise needed is that the value we return typechecks.

One odd thing you might notice in the `return` rule is that P is given two arguments. Actually, preconditions in these judgments, unlike those in the Hoare type for a complete computation must always take two heaps, since as we will see in the remaining rules, the postcondition of one piece of a computation can become the precondition of the next piece. Think of these arguments as the heap at the beginning of the whole program and the heap right before the current piece of the computation. At the beginning of a computation, these will be the same. Let's look at the rule that gets us started:

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

To typecheck a complete, `do`-encapsulated computation against its Hoare type, we build a two-argument precondition out of the given precondition R, by requiring the two arguments to be the same and then applying R to one of them. We also frame the fragment of the heap that R applies to with an arbitrary heap fragment (one that satisfies $\lambda m'.\top$). This is the interface between the small-footprint specifications for Hoare types we discussed earlier and the specifications in the judgments, which must describe the whole heap. The postcondition we check for E in the premise uses an operator $\gg$, whose formal definition we will omit here, but which essentially builds a large-footprint postcondition out of Q by requiring that if the initial heap i consists of a fragment described by R and a frame, then the final heap must leave the latter unchanged, while the former is described by $(Q\ i)$.

We'll omit the typing rules for conditionals and recursive functions, but let's look at the rules for dereferencing, assignment, and running a computation:

Here is the rule for dereference:

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

A computation beginning with a dereference must be dereferencing a well-typed expression of type `nat` (a location), and then we check the rest of the computation in a context that adds x, the variable we bound, under a precondition that extends the earlier precondition by requiring that the location

we dereferenced maps to x in the heap (the $\hookrightarrow$ is just a shorthand for seleq with the arguments rearranged; unlike $\mapsto$, it doesn't specify a singleton heap, which we need here because we can't do a disjoint union: P will already have something to say about location M). We also add a verification condition that in a heap satisfying P, M must point to something of the right type τ.

Next, we have the rule for assignment:

$$\frac{\Delta \vdash \tau \Leftarrow \texttt{mono} \qquad \Delta \vdash M \Leftarrow \texttt{nat} \qquad \Delta \vdash N \Leftarrow \tau \\ \Delta, i : \mathsf{heap}, m : \mathsf{heap}, (P\ i\ m) \vDash (M \hookrightarrow -)\ m \qquad \Delta; P \circ ((M \mapsto -) \multimap (M \mapsto_\tau N)) \vdash E \Rightarrow y : B.Q}{\Delta; P \vdash M :=_\tau N; E \Rightarrow y : B.Q}$$

First, we typecheck the pure terms that make up the location to assign to and the value to assign. As in the dereference rule, we have a proof obligation that the precondition implies that the location M is mapped in the heap, and a premise that the rest of the computation typechecks under an appropriate precondition. In this case, the condition says to take the part of the original precondition P that describes M (that is, the part that looks like $M \mapsto -$) and replace it with $M \mapsto_\tau N$. (Formal definitions of $\circ$, $\multimap$, and $\gg$ are in the appendix, but the details are not worth giving here.)

Finally, here is the rule for running a computation:

$$\frac{\Delta \vdash K \Rightarrow \{R\}\ x : A\ \{S\} \\ \Delta, i : \mathsf{heap}, m : \mathsf{heap}, (P\ i\ m) \vDash (R * \lambda m'.\top)\ m \qquad \Delta, x : A; P \circ (R \gg S) \vdash E \Rightarrow y : B.Q}{\Delta; P \vdash \texttt{run}\ x = K\ \texttt{in}\ E \Rightarrow y : B.\lambda i.\lambda m.\exists x : A.(Q\ i\ m)}$$

First, we must synthesize the type of the computation we want to run. Recall that an intro term such as $\texttt{do}\, E$ appearing in an elim position must be annotated with a type, so we will often actually be checking K against a Hoare type that was given. Next, we check the rest of the current computation E, under a precondition that composes P with the large-footprint version of the postcondition of K, that is, $R \gg S$ (this is the same thing we did in the do rule).

None of the rules in the computation judgments required us to perform decidable checks for equality of specifications. Thus we don't need to normalize computations, and we can soundly handle their effects in our dependent type system.

To recap, we've seen how Hoare type theory merges Hoare logic-style specifications for effectful computations into a dependent type system, and taken a bit of a look at what kinds of precise specifications the system lets us encode. We've also had a look at the inner workings of the type system.

6. Bibliography

Polymorphism and Separation in Hoare Type Theory; Aleksandar Nanevski, Greg Morrisett, Lars Birkedal; ICFP 2006

This earlier version of HTT is similar to the version we developed here in many ways, but it is only first-order; that is, we can't abstract over propositions in the same way as we saw, and thus we can't hide local state invariants.

Abstract Predicates and Mutable ADTs in Hoare Type Theory; Aleksandar Nanevski, Amal Ahmed, Greg Morrisett, Lars Birkedal; ESOP 2007.

This paper gives the version of Hoare Type Theory that was discussed in these notes. If you are looking for more detail, the technical report for this paper is a good place to go. It explicitly lays out pretty much everything going on in the system, including all the details needed for normalization that we alluded to, and all the other definitions we elided.

Ynot: Dependent Types for Imperative Programs; Aleksandar Nanevski, Greg Morrisett, Avi Shinnar, Paul Govereau, Lars Birkedal; ICFP 2008

In this paper, the authors show how HTT can be implemented as an axiomatic extention to Coq, and discuss a bunch of the features of their implementation.

Appendix: Complete Typing Rules

These rules are still simplified: they don't perform normalization. For the *really* complete typing rules, see the tech report version of Nanevski et al. 2007.

Checking contexts. $\boxed{\vdash \Delta\ \mathtt{ctx}}$

$$\frac{}{\vdash \cdot\ \mathtt{ctx}} \qquad \frac{\vdash \Delta\ \mathtt{ctx} \qquad \Delta \vdash A\ \mathtt{type}}{\vdash \Delta, x{:}A\ \mathtt{ctx}} \qquad \frac{\vdash \Delta\ \mathtt{ctx} \qquad \Delta \vdash P \Leftarrow \mathtt{prop}}{\vdash \Delta, P\ \mathtt{ctx}}$$

Checking types. $\boxed{\Delta \vdash A\ \mathtt{type}}$

$$\frac{\Delta \vdash K \Rightarrow \mathtt{mono}}{\Delta \vdash K\ \mathtt{type}} \qquad \frac{}{\Delta \vdash \mathtt{nat}\ \mathtt{type}} \qquad \frac{}{\Delta \vdash \mathtt{bool}\ \mathtt{type}} \qquad \frac{}{\Delta \vdash \mathtt{prop}\ \mathtt{type}} \qquad \frac{}{\Delta \vdash \mathtt{mono}\ \mathtt{type}}$$

$$\frac{}{\Delta \vdash \mathtt{1}\ \mathtt{type}} \qquad \frac{\Delta \vdash A\ \mathtt{type} \qquad \Delta, x{:}A \vdash B\ \mathtt{type}}{\Delta \vdash \Pi x{:}A.B\ \mathtt{type}} \qquad \frac{\Delta \vdash A\ \mathtt{type} \qquad \Delta, x{:}A \vdash B\ \mathtt{type}}{\Delta \vdash \Sigma x{:}A.B\ \mathtt{type}}$$

$$\frac{\Delta \vdash A\ \mathtt{type} \qquad \Delta, x{:}A \vdash P \Leftarrow \mathtt{prop}}{\Delta \vdash \{x{:}A.P\}\ \mathtt{type}}$$

$$\frac{\Delta \vdash P \Leftarrow \mathtt{heap} \to \mathtt{prop} \qquad \Delta \vdash A\ \mathtt{type} \qquad \Delta, x{:}A \vdash Q \Leftarrow \mathtt{heap} \to \mathtt{heap} \to \mathtt{prop}}{\Delta \vdash \{P\}\ x{:}A\ \{Q\}\ \mathtt{type}}$$

Checking intro terms. $\boxed{\Delta \vdash M \Leftarrow A}$

$$\frac{\Delta \vdash K \Rightarrow A \qquad A = B}{\Delta \vdash K \Leftarrow B} \qquad \frac{}{\Delta \vdash \mathtt{true} \Leftarrow \mathtt{bool}} \qquad \frac{}{\Delta \vdash \mathtt{false} \Leftarrow \mathtt{bool}} \qquad \frac{}{\Delta \vdash n \Leftarrow \mathtt{nat}}$$

$$\frac{}{\Delta \vdash () \Leftarrow \mathtt{1}} \qquad \frac{\Delta \vdash M \Leftarrow \mathtt{nat} \qquad \Delta \vdash N \Leftarrow \mathtt{nat}}{\Delta \vdash M + N \Leftarrow \mathtt{nat}} \qquad \frac{\Delta \vdash M \Leftarrow \mathtt{nat} \qquad \Delta \vdash N \Leftarrow \mathtt{nat}}{\Delta \vdash M \times N \Leftarrow \mathtt{nat}}$$

$$\frac{\Delta \vdash M \Leftarrow \mathtt{nat} \qquad \Delta \vdash N \Leftarrow \mathtt{nat}}{\Delta \vdash \mathtt{eq}_{\mathtt{nat}}(M, N) \Leftarrow \mathtt{nat}} \qquad \frac{\Delta, x{:}A \vdash M \Leftarrow B}{\Delta \vdash \lambda x.M \Leftarrow \Pi x{:}A.B} \qquad \frac{\Delta \vdash M \Leftarrow A \qquad \Delta \vdash N \Leftarrow B[M/x]}{\Delta \vdash \langle M, N\rangle \Leftarrow \Sigma x{:}A.B}$$

$$\frac{\Delta \vdash M \Leftarrow A \qquad \Delta \vDash P[M/x]}{\Delta \vdash \mathtt{in}\, M \Leftarrow \{x{:}A.P\}} \qquad \frac{\Delta; \lambda i.\lambda m.i = m \wedge (R * \lambda m'.\top)\ m \vdash E \Leftarrow x{:}A.R \gg Q}{\Delta \vdash \mathtt{do}\, E \Leftarrow \{R\}\ x{:}A\ \{Q\}}$$

$$\frac{\Delta \vdash A\ \mathtt{type} \qquad \Delta \vdash B\ \mathtt{type} \qquad \Delta \vdash M \Leftarrow A \qquad \Delta \vdash M \Leftarrow B}{\Delta \vdash \mathtt{xid}_{A,B}(M, N) \Leftarrow \mathtt{prop}} \qquad \frac{}{\Delta \vdash \top \Leftarrow \mathtt{prop}}$$

$$\frac{}{\Delta \vdash \bot \Leftarrow \mathtt{prop}} \qquad \frac{\Delta \vdash M \Leftarrow \mathtt{prop} \qquad \Delta \vdash N \Leftarrow \mathtt{prop}}{\Delta \vdash M \wedge N \Leftarrow \mathtt{prop}} \qquad \frac{\Delta \vdash M \Leftarrow \mathtt{prop} \qquad \Delta \vdash N \Leftarrow \mathtt{prop}}{\Delta \vdash M \vee N \Leftarrow \mathtt{prop}}$$

$$\frac{\Delta \vdash M \Leftarrow \mathtt{prop} \qquad \Delta \vdash N \Leftarrow \mathtt{prop}}{\Delta \vdash M \supset N \Leftarrow \mathtt{prop}} \qquad \frac{\Delta \vdash M \Leftarrow \mathtt{prop}}{\Delta \vdash \neg M \Leftarrow \mathtt{prop}}$$

$$\frac{\Delta \vdash A\ \mathtt{type} \qquad \Delta, x{:}A \vdash M \Leftarrow \mathtt{prop}}{\Delta \vdash \forall x{:}A.M \Leftarrow \mathtt{prop}} \qquad \frac{\Delta \vdash A\ \mathtt{type} \qquad \Delta, x{:}A \vdash M \Leftarrow \mathtt{prop}}{\Delta \vdash \exists x{:}A.M \Leftarrow \mathtt{prop}}$$

$$\frac{}{\Delta \vdash \mathtt{nat} \Leftarrow \mathtt{mono}} \qquad \frac{}{\Delta \vdash \mathtt{bool} \Leftarrow \mathtt{mono}} \qquad \frac{}{\Delta \vdash \mathtt{prop} \Leftarrow \mathtt{mono}} \qquad \frac{}{\Delta \vdash \mathtt{1} \Leftarrow \mathtt{mono}}$$

$$\frac{\Delta \vdash \tau \Leftarrow \mathtt{mono} \qquad \Delta, x : \tau \vdash \sigma \Leftarrow \mathtt{mono}}{\Delta \vdash \Pi x : \tau . \sigma \Leftarrow \mathtt{mono}} \qquad \frac{\Delta \vdash \tau \Leftarrow \mathtt{mono} \qquad \Delta, x : \tau \vdash \sigma \Leftarrow \mathtt{mono}}{\Delta \vdash \Sigma x : \tau . \sigma \Leftarrow \mathtt{mono}}$$

$$\frac{\Delta \vdash \tau \Leftarrow \mathtt{mono} \qquad \Delta, x : \tau \vdash P \Leftarrow \mathtt{prop}}{\Delta \vdash \{x : \tau . P\} \Leftarrow \mathtt{mono}}$$

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

Checking elim terms. $\boxed{\Delta \vdash K \Rightarrow A}$

$$\frac{}{\Delta, x : A, \Delta' \vdash x \Rightarrow A} \qquad \frac{\Delta \vdash K \Rightarrow \Pi x : A . B \qquad \Delta \vdash M \Leftarrow A}{\Delta \vdash K\ M \Rightarrow B[M/x]} \qquad \frac{\Delta \vdash K \Rightarrow \Sigma x : A . B}{\Delta \vdash \mathtt{fst}\, K \Rightarrow A}$$

$$\frac{\Delta \vdash K \Rightarrow \Sigma x : A . B}{\Delta \vdash \mathtt{snd}\, K \Rightarrow B[\mathtt{fst}\, K/x]} \qquad \frac{\Delta \vdash K \Rightarrow \{x : A . P\}}{\Delta \vdash \mathtt{out}\, K \Rightarrow A} \qquad \frac{\Delta \vdash A\ \mathtt{type} \qquad \Delta \vdash M \Leftarrow A}{\Delta \vdash M : A \Rightarrow A}$$

Inference rules. $\boxed{\Delta \vDash P}$

$$\frac{}{\Delta, P, \Delta' \vDash P} \qquad \frac{\Delta, P \vDash Q}{\Delta \vDash P \supset Q} \qquad \frac{\Delta \vDash P \supset Q \qquad \Delta \vDash P}{\Delta \vDash Q} \qquad \frac{\Delta, x : A \vDash P}{\Delta \vDash \forall x : A . P}$$

$$\frac{\Delta \vDash \forall x : A . P \qquad \Delta \vdash M \Leftarrow A}{\Delta \vDash P[M/x]} \qquad \frac{\Delta \vdash K \Rightarrow \{x : A . P\}}{\Delta \vDash P[K/x]}$$

There are also a lot of axioms giving the properties of primitives.

Checking computations. $\boxed{\Delta; P \vdash E \Rightarrow x : A . Q}$ $\boxed{\Delta; P \vdash E \Leftarrow x : A . Q}$

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

$$\frac{\Delta; P \vdash E \Rightarrow x\!:\!A.S \qquad \Delta, x\!:\!A, i\!:\!\mathsf{heap}, m\!:\!\mathsf{heap}, (S\ i\ m) \vDash (Q\ i\ m)}{\Delta; P \vdash E \Leftarrow x\!:\!A.Q}$$

$$\frac{\Delta \vdash M \Leftarrow A}{\Delta; P \vdash \mathtt{return}\, M \Rightarrow x\!:\!A.(\lambda i.\lambda m.(P\ i\ m) \wedge \mathtt{xid}_{A,A}(x, M))}$$

$$\frac{\begin{array}{c}\Delta \vdash K \Rightarrow \{R\}\ x\!:\!A\ \{S\} \\ \Delta, i\!:\!\mathsf{heap}, m\!:\!\mathsf{heap}, (P\ i\ m) \vDash (R * \lambda m'.\top)\ m \qquad \Delta, x\!:\!A; P \circ (R \gg S) \vdash E \Rightarrow y\!:\!B.Q\end{array}}{\Delta; P \vdash \mathtt{run}\ x = K\ \mathtt{in}\, E \Rightarrow y\!:\!B.\lambda i.\lambda m.\exists x\!:\!A.(Q\ i\ m)}$$

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

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

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

$$\frac{\begin{array}{c}\Delta \vdash A\ \mathtt{type} \qquad \Delta, x\!:\!A \vdash T\ \mathtt{type} \\ T = \{R\}\ y\!:\!B\ \{S\} \qquad \Delta \vdash M \Leftarrow A \qquad \Delta, i\!:\!\mathsf{heap}, m\!:\!\mathsf{heap}, (P\ i\ m) \vDash (R[M/x] * \lambda h.\top)\ m \\ \Delta, x\!:\!A, f\!:\!\Pi x\!:\!A.T; \lambda i.\lambda m.i = m \wedge (R * \lambda h.\top)\ m \vdash E \Leftarrow y\!:\!B.R \gg S \\ \Delta, y\!:\!B[M/x]; P \circ (R \gg S)[M/x] \vdash F \Rightarrow z\!:\!C.Q\end{array}}{\Delta; P \vdash y = \mathtt{fix}\, f(x\!:\!A) : T = \mathtt{do}\, E\, \mathtt{in}\, \mathtt{eval}\, f\ M; F \Rightarrow z\!:\!C.(\lambda i.\lambda m.\exists y\!:\!B[M/x].Q\ i\ m)}$$