# Formalized Proof of Type Safety of Hoare Type Theory

Evgeny Makarov Christian Skalka

University of Vermont
emakarov@gmail.com skalka@cs.uvm.edu

April 23, 2010

**Abstract**

We prove type safety of the Hoare Type Theory (HTT), an extension of Separation Logic and Hoare Logic to higher-order functional programs. Our proof is rather simple and is based on subject reduction, unlike previous work on HTT by Birkedal et al., which uses nontrivial denotational semantics. Further, we formalized our proof in the Coq theorem prover. This formalization can be seen as a basis of a platform for verifying higher-order imperative programs that is alternative to Ynot, another implementation of HTT in Coq.

## 1 Background and Contributions

The use of dependent types as expressive specifications for functional programs is an active topic of research. Unfortunately, research progress in verification of side-effecting programs has been slower despite the importance of this topic. In functional programs, limited use of assignment, for example, is both convenient and common; most dialects of ML include mutable references, and Haskell allows assignment using monads.

On the other hand, Hoare logic has been used for over forty years to specify and prove imperative programs. Hoare logic uses formulas, or *assertions*, of first-order logic (or some other chosen logic) to describe *heaps*, i.e., memory segments. We will write $h \models P$ whenever an assertion $P$ is true on a heap $h$. The derivable judgments of Hoare logic are triples $\{P\}M\{Q\}$ where $M$ is a program and $P$ and $Q$ are assertions, called respectively pre- and postconditions. The triple $\{P\}M\{Q\}$ means that if $h \models P$ for some $h$, then $M$, starting with $h$, works without errors and, if it finishes computation with another heap $h'$, then $h' \models Q$.

We developed an analog of Hoare logic (and its extension, separation logic [6]) for a small functional programming language with assignments. Our goal was not the creation of an effective platform for verifying real-life programs; rather,

we wanted to come up with "standard," educational presentation of the logic, trying to make it resemble the original Hoare logic in clarity, simplicity and, appeal.

Independently, the Ynot group at Harvard [2] have developed two versions of the Hoare Type Theory [3, 1] (let's call them HTT1 and HTT2) with the goal of bringing together Hoare logic and functional programming. Also, on the basis of the Coq proof assistant, the group created the environment Ynot for program verification and reported its use in various verification projects.

We did not judge HTT1 to be a solution to our project because of the complexity of HTT1. Despite not having the strength of the whole Calculus of Constructions (CC), the presentation of HTT1 is quite a bit more involved compared to either Hoare logic or Pure Type Systems (the usual way of presenting CC and similar strong dependently typed systems). The authors admit in [4] that their "inference rules... may look rather unwieldy at first sight, especially when compared to rules in Hoare or Separation Logic... Ynot is really more closely related to predicate transformers than to Hoare Logic."

On the other hand, HTT2, which was developed earlier and independently of our logic, is quite similar to it. We do not claim, therefore, the discovery of the system. However, we present the following contributions that we believe are original.

- The proof of the type safety of HTT2 presented in [5] uses realizability model and is rather complicated. To our knowledge, we give the first traditional proof of type safety based on subject reduction.

- While Ynot is a shallow embedding of HTT in Coq, we used deep embedding and defined operational semantics of the language, which allowed formally proving the type safety of HTT in Coq. The complete Coq code can be downloaded from
  `http://www.cs.uvm.edu/~skalka/skalka-pubs/coq/HTT/HTT.zip`.

- Our formalization adds to Coq only one axiom, Proof Irrelevance (PI), which is universally considered compatible with Coq and even desirable. In contrast, Ynot adds about two dozen axioms.

- While the current Ynot may be considered an implementation of HTT2, whose consistency, as has been said earlier, was proved model-theoretically, the two systems are not exactly the same. For example, HTT2 is impredicative and uses classical logic, while in Coq, impredicativity and classical logic are incompatible. Also, the behavior of ghost variables in the two systems is different. In contrast, one can use our system for verifying programs, and it is precisely this system that is proved sound.

- We discuss some details of inference rules. For example, our change to the typing rule of the reading operator in HTT2 makes it have truly *small footprint* (a desirable property of assertions that describe only the affected portion of the heap). Similarly, we introduce an *existential elimination rule* and discuss the constructive content of the subject reduction theorem.

**Parameters**

addr : Type
heap : Type

**Record**

value := $\{T : \mathsf{Type}, x : T\}$

**Implicit Types**

$T$ : Type
$h$ : heap
$a$ : addr
$v$ : value
$w$ : option value

---

**Parameters**

emp : heap
$*$ : heap → heap → heap
freeAddr : heap → addr
access : heap → addr → option value
update : heap → addr → option value → heap

**Definitions**

$a \in \mathrm{dom}(h) := \exists v.\ h[a] = \mathsf{Some}\, v$
$h_1 \equiv h_2 := \forall a.\ h_1[a] = h_2[a]$
$h_1 \# h_2 := \forall a.\ \neg(a \in \mathrm{dom}(h_1) \wedge a \in \mathrm{dom}(h_2))$

**Notations**

$h[a] := \mathsf{access}\ h\, a$
$h[a := w] := \mathsf{update}\ h\, a\, w$

---

**Axioms**

$\forall h.\ \mathsf{freeAddr}(h) \notin \mathrm{dom}(h)$
$\forall a.\ a \notin \mathrm{dom}(\mathsf{emp})$
$\forall h, a, w.\ (\mathsf{update}\, h\, a\, w)[a] = w$
$\forall h, a, a', w.\ a \neq a' \rightarrow (\mathsf{update}\, h\, a\, w)[a'] = h[a']$
$\forall h_1, h_2, a, v.\ h_1 \# h_2 \rightarrow$
$((h_1 * h_2)[a] = \mathsf{Some}\, v \leftrightarrow h_1[a] = \mathsf{Some}\, v \vee h_2[a] = \mathsf{Some}\, v)$

Figure 1: Heap-related concepts

The rest of the paper is organized as follows. Section 2 describes heaps and assertions. Section 3 discusses choices about the implementation of HTT (deep embedding, ghost variables, and so on) and point out specifics of some inference rules (the ones concerning reading and existential elimination). Section 4 outlines the proof of type safety via subject reduction.

# 2 Implementation of Separation Logic

## 2.1 Heaps

The foundation of our implementation of HTT is an embedding of separation logic into Coq. It is mostly standard but uses Coq modules to separate the specification of heap-related concepts from their implementation. The *module*

*type*, or *signature*, in ML terminology, describing these concepts is shown in Figure 1. Coq keywords Parameter and Axiom indicate abstract entities whose definition is provided in a module that realizes the given module type. The command Implicit Types associates a default type with variables whose name starts with a given string. For example, unless an explicit type is specified, bound variables $a$, $a'$ and $a1$ are assigned type addr.

**Remark 2.1 (Notations in Figures)** To spare the reader from the need to learn too much concrete syntax of Coq, definitions in figures are shown semi-formally, using usual mathematical notation whenever possible. Since we are describing a formalization in Coq, all terms in formulas are typed Coq terms. When convenient, their types are shown in the superscript.

The module type in Figure 1 provides abstract types addr and heap, as well as a function access $h\,a$, written as $h[a]$, that returns the content stored at the address $a$ in the heap $h$. The result has the type option value, where value is a dependent record holding a type $T$ and a term $x : T$, and the dependent type option from the standard library has two constructors: $\mathsf{Some} : \forall T : \mathsf{Type}, x : T.\ \mathsf{option}\,T$ and $\mathsf{None} : \forall T.\ \mathsf{option}\,T$. Thus, a heap $h$ can be viewed as a partial function from addr to value. The domain of $h$ is defined to consist of addresses mapped to $\mathsf{Some}\,w$ for some $w$. We assume that heaps are not everywhere defined (in order to allocate new cells) but are not necessarily finite. The parameter function $\mathsf{freeAddr}\,h$ returns some unallocated address of $h$.

The relation $\equiv$, defined in Figure 1, identifies heaps that are extensionally equal but may be implemented differently. As is standard in separation logic, two heaps $h_1$ and $h_2$ are called disjoint (written $h_1 \# h_2$) if their domains are. The union of two disjoint heaps is denoted $h_1 * h_2$. All parameter operations on heaps are required to respect $\equiv$.

## 2.2 Modularity

The use of modules allows providing different implementations for heaps and operations on them. We constructed one implementation based on the modular Coq library FSets of finite sets and maps, which in turn has several implementations using lists and trees. Besides adding clarity to the proof development, separating specification and implementation allows choosing data structures used by OCaml programs extracted from Coq proofs. For example, one may be interested in the function extracted from the Progress statement, which computes the next state (a pair of a heap and a term). Also, data structures used in the implementation determine the efficiency of *proofs by reflection*, where Coq is used as a programming language to encode decision procedures.

## 2.3 Separation Logic

Separation logic is an extension of Hoare logic that is especially suitable for specifying behavior of programs that manipulate pointers to heap, including dangling pointers, aliasing, and so on. It adds a new binary assertion connective

**Parameters**

| | | |
|---|---|---|
| HProp | : | Type |
| $\mathbb{\wedge}, \mathbb{\vee}, *$ | : | HProp → HProp → HProp |
| $\exists$ | : | $\forall T.\ (T \to \mathsf{HProp}) \to \mathsf{HProp}$ |
| Emp | : | HProp |
| $[\cdot]$ | : | Prop → HProp |
| points | : | addr → $\forall T.\ T \to$ HProp |

**Implicit Types**

$P, Q, R$ : HProp

**Notation**

$a \mapsto_T x$ := points $a\, T\, x$

**Parameter**

$\models$ : heap → HProp → Prop

**Definitions**

$P \models Q$ := $\forall h.\ h \models P \to h \models Q$

$P \dashv\models Q$ := $P \models Q \wedge Q \models P$

**Axioms** (Selection)

$\forall h.\ h \models \mathsf{Emp} \leftrightarrow h \equiv \mathsf{emp}$

$\forall h, a, T, x : T.\ h \models a \mapsto_T x \leftrightarrow h \equiv \mathsf{update\ emp}\, a\, (\mathsf{Some}\ \{\!|T, x|\!\})$

$\forall h, A.\ h \models [A] \leftrightarrow A$

$\forall h, P_1, P_2.\ h \models P_1 * P_2 \leftrightarrow$

$\quad \exists h_1, h_2.\ h_1 \# h_2 \wedge h \equiv h_1 * h_2 \wedge h_1 \models P_1 \wedge h_2 \models P_2$

$\forall h_1, h_2, P.\ h_1 \equiv h_2 \to (h_1 \models P \leftrightarrow h_2 \models P)$

Figure 2: Separation logic

$*$, called separation conjunction, with the following semantics: $h \models P_1 * P_2$ means that $h = h_1 * h_2$ for some $h_1 \# h_2$ and $h_i \models P_i$ for $i = 1, 2$.[1]

As a calculus of triples, separation logic adds an important *Frame rule*.

$$\frac{\{P\}M\{Q\}}{\{P * R\}M\{Q * R\}}$$

It means that if $M$ converts a heap satisfying $P$ into one satisfying $Q$, then it will work similarly when run on a larger heap and won't touch the excess part.

Figure 2 shows parameters, axioms and definitions related to separation logic. An abstract type HProp is the type of formulas. Formula semantics is expressed by the parameter relation $\models$ between heaps and formulas. Several axioms defining $\models$ are listed in Figure 2. In particular, Emp is true on a heap with an empty domain, and $a \mapsto_T x$ is true on a heap that maps stores $x : T$ at $a$ and is otherwise empty. (When there is no confusion, we'll suppress the subscript $T$. In the concrete syntax of Coq, there is no need to write $T$, either, since it can be reconstructed as the type of $x$.) The connectives $\mathbb{\wedge}$, $\mathbb{\vee}$ and $\exists$ are just regular connectives $\wedge$, $\vee$ and $\exists$ lifted from Prop to HProp. The notation $[A]$

[1] Separation logic also has an adjoint to $*$, called *separating implication* and denoted $-\!\!*$, but it is not used in this paper.

injects a pure proposition $A$ : Prop into HProp. (Note that, unlike the similar operation in Ynot, $h \models [A]$ does not require that $h$ is empty.)

As other functions on heaps, $\models$ has to respect the relation $\equiv$. This is necessary, for example, to prove $P * \mathsf{Emp} =|\models P$ because the heaps $h$ and $h * \mathsf{emp}$, for example, don't need to have equal representation. Ynot solves this problem by defining heaps as functions from addresses to typed values and by using the functional extensionality axiom. Thus, extensionally equal heaps are related by Leibniz equality and can replace each other in any context. This provides the freedom to define HProp simply as heap $\to$ Prop and enable users to define arbitrary connectives and formulas as predicates on heaps.

In our approach, this is not possible because all connectives must treat heaps respecting their extensional equality. This explains the need for a parameter type HProp. In our implementation using Coq FSets library, HProp is an inductive type; this allows proving the last axiom of Figure 2, expressing extensionality of $\models$. Note that, similar to Ynot, users are free to inject any Coq proposition into HProp using square brackets and to define functions that return values of type HProp.

# 3 Implementation of HTT

The idea of HTT is to separate pure and effectful terms by creating a monad labeled with pre- and postconditions.

**Definition 3.1 (Specification)** Let $T$ : Type, $P$ : HProp and $Q : T \to$ HProp. The expression $\{P\}$-$\{Q\}$ is called a *specification*, and $P$ and $Q$ are called *pre-* and *postcondition*, respectively. The notation $\{P\}$-$\{x : T \mid Q\}$[2] stands for $\{P\}$-$\{\lambda x : T.\, Q\}$. (By default, Coq does not identify $\eta$-long and $\eta$-short term forms.)

One can implement HTT using either shallow or deep embedding. In the shallow embedding (the choice of Ynot), $\{P\}$-$\{Q\}$ is a type of sort Type, and specifications are types of verified programs. One declares or defines operators (return, do, assign, and so on) having specification types, as well as terms that do not have computational meaning but that modify specifications (e.g., Frame rule). To verify that a certain program satisfies a specification $\{P\}$-$\{Q\}$, one starts the proof of a theorem $\{P\}$-$\{Q\}$. The proof term then represents the program (modulo the presence of noncomputational rules). The user may write the proof using the refine tactic, giving it as argument the outline of the program; the rest of the proof would fill in non-computational details.

## 3.1 Ghost Variables and Computational Irrelevance

The choice of embedding is influenced, in particular, by the implementation of *ghost variables*, whose role is relating pre- and postconditions. For example, a

[2] Coq parser is smart enough to accept precisely this notation.

term $M : (\Pi a : \mathsf{addr}\, \forall n : \mathsf{nat}.\, \{a \mapsto n\}\text{-}\{\_ : \mathsf{unit} \mid a \mapsto n+1\})$ increments the content of the argument address $a$. Unlike the real argument $a$, introduced by $\Pi$, $M$ does not receive $n$ at run time.

The ideal solution for ghost variables would be the Implicit Calculus of Constructions (ICC) by A. Miquel and its decidable variant ICC*, currently developed by B. Barras (See the discussion on computational irrelevance in [1, Section 2.1.1].) In addition to regular CC constructs: dependent product $\Pi x : T.\, U$, abstraction $\lambda x : T.\, M$ and application $MN$, ICC* has their implicit counterparts: $\Pi[x : T].\, U$, $\lambda[x : T].\, M$ and $M[N]$. Implicit arguments (in brackets) are not supposed to drive the computation but only to assure its correctness. Correspondingly, the judgment $\Gamma \vdash \lambda[x : T].\, M : \Pi[x : T].\, U$ can be derived from $\Gamma, x : T \vdash M : U$ only when $x$ is not free in $M$ *after all implicit arguments are removed* from $M$.

In regular Coq, there is no fully satisfactory way to simulate this behavior. The inference rule of CiC that introduces $\lambda$-abstraction has the form

$$\frac{\Gamma \vdash \Pi x : T.\, U : s \qquad \Gamma, x : T \vdash M : U}{\Gamma \vdash (\lambda x : T.\, M) : (\Pi x : T.\, U)}$$

(The first premise $\Gamma \vdash \Pi x : T.\, U : s$ ensures that forming a function space with domain $T$ and codomain $U$ is allowed.) The rule does not make a distinction between free occurrences of $x$ in $M$ and in $U$; both are closed in the conclusion. Thus, implementing ghost variables with deep embedding, when the program is a part of the proof-term, presents a problem.

As a "duct-tape" solution, Ynot uses the distinction between sorts $\mathsf{Prop}$ and $\mathsf{Set}$. Types of sort $\mathsf{Prop}$ are theorems, while types in $\mathsf{Set}$ are computational data-structures. The Coq principle of PI states that terms that are proofs of theorems cannot be used for driving programs but only to justify that certain program steps are valid. One of the manifestations of PI is that inductive types of sort $\mathsf{Prop}$ cannot be taken apart using the built-in $\mathsf{match}$ construct if the type of the overall term is in $\mathsf{Set}$ or $\mathsf{Type}$.

Using the inductive type $\mathsf{inhabited} : \mathsf{Type} \to \mathsf{Prop}$, which has one constructor $\mathsf{inhabits} : \forall T,\, T \to \mathsf{inhabited}\, T$, any term of sort $\mathsf{Type}$ can be injected into $\mathsf{Prop}$. The opposite transformation can only occur in the context of building a proof. Since Ynot specifications, which are types of verified programs, have sort $\mathsf{Set}$, terms injected into $\mathsf{Prop}$ in this way essentially cannot be used by programs and may only serve in supporting proofs.

This solution has an additional advantage of imposing no restrictions on the location of implicit universal quantifiers. There are also disadvantages. One is a small notational overhead: if $x : \mathsf{inhabited}\, T$ is a ghost variable and $P : T \to \mathsf{HProp}$ is a predicate, then one must write $x \,\tilde{}\tilde{}\, P$, which is a notation for $\lambda h.\, \exists y, x = \mathsf{inhabits}\, y \wedge Pyh$, instead of simply $Px$. A more serious issue is that this approach requires an axiom $\forall T : \mathsf{Set}\, \forall x, y : T.\ \mathsf{inhabits}\, x = \mathsf{inhabits}\, y \to x = y$. However, this axiom contradicts PI because both terms $\mathsf{inhabits}\, x$ and $\mathsf{inhabits}\, y$ have type $\mathsf{inhabited}\, T : \mathsf{Prop}$, and PI implies $\mathsf{inhabits}\, x = \mathsf{inhabits}\, y$. Thus, in the presence of PI, all terms of the same type, such as $\mathsf{true}$ and $\mathsf{false}$, are provably equal.

$$
\begin{array}{rcl}
M, N \ : \ \mathsf{Term} & ::= & \mathsf{return}\, x \mid \mathsf{do}\ x \leftarrow M;\ {}^{T}M \mid \mathsf{new}\, x \mid \mathsf{read}\, a \mid \mathsf{assign}\, a\, x \mid \\
& & \mathsf{free}\, a \mid \mathsf{if}\ x^{\mathsf{bool}}\ \mathsf{then}\ M\ \mathsf{else}\ M \mid f \mid \mathsf{fix}\, f.\ {}^{T}M \mid \mathsf{app}\, M\, x \\
S \ : \ \mathsf{Spec} & ::= & \{P\}\text{-}\{{}^{T}Q\} \mid \forall\ {}^{T}S \mid \Pi^{\mathsf{fix}}\ {}^{T}S
\end{array}
$$

Figure 3: Term syntax

Even though PI is not derivable in Coq by itself and is by no means a typical part of Coq developments, it is implied by classical logic. As a result, the use of classical logic in Ynot is problematic.

## 3.2 Ghost Variables and Deep Embedding

Our implementation of ghost variables is different. Instead of making $\{P\}$-$\{Q\}$ a type, we prove theorems of the form $\mathsf{Derives}\, M\ \{P\}$-$\{Q\}$, denoted by $\vdash M : \{P\}$-$\{Q\}$. Here $M$, which represent the program under verification, has an inductive type $\mathsf{Term}$, and $\{P\}$-$\{Q\}$ has type $\mathsf{Spec}$. These inductive types are shown in Figure 3. The inductive predicate $\mathsf{Derives}$ comprising inference rules of HTT is shown in Figure 4.

**Remark 3.2** In Figure 3, we combine type information with the traditional BNF form. For example, $\mathsf{Term}$ is an inductive type, and the default placeholder for $\mathsf{Term}$ is $M$. When the definition requires a function of some type $T \to \mathsf{Term}$ rather than an object of type $\mathsf{Term}$, we write $T$ as the left superscript: ${}^{T}M$. The type of the term itself, if necessary, is shown as the right superscript, as in $x^{\mathsf{bool}}$.

In creating the datatype $\mathsf{Term}$, which is characteristic of deep embedding, we tried to use as much as possible the functionality already available in Coq, such as substitutions and unification. We had to implement manually only effectful operations and recursion (see the discussion below). That's why, for example, the reduction rule

$$(h, \mathsf{do}\ x \leftarrow \mathsf{return}\, x;\ M) \triangleright (h, Mx)$$

from Figure 5 uses regular application instead of manually written substitution function.

The inductive type $\mathsf{Spec}$ in Figure 3 includes a constructor $\forall : (\Pi T.\, (T \to \mathsf{Spec}) \to \mathsf{Spec})$, which is the universal quantifier for ghost variables. For example, consider a function $S \stackrel{\text{def}}{=} \lambda n : \mathsf{nat}.\, \{a \mapsto n\}$-$\{a \mapsto n+1\} : \mathsf{nat} \to \mathsf{Spec}$ and a term $M \stackrel{\text{def}}{=} \mathsf{do}\ x \leftarrow \mathsf{read}\, a;\ \mathsf{assign}\, a\, (x+1) : \mathsf{Term}$. One can prove $\vdash M : \forall S$. The last inference rule would be

$$\frac{\Pi n : \mathsf{nat}.\, (\vdash M : Sn)}{\vdash M : \forall S}\ (\forall I)$$

Thus, separating the verified program $M$ from the proof-term of the theorem $\vdash M : \forall S$, we are able to require that the ghost variable $n$ does not occur in $M$.

$$\Gamma \vdash \mathsf{new}\, x : \{\mathsf{emp}\}\text{-}\{a : \mathsf{addr} \mid a \mapsto x\}\ (\textsc{New})$$

$$\Gamma \vdash \mathsf{read}\, a : \{a \mapsto_T x\}\text{-}\{y : T \mid [y = x] \mathbin{\mathbb{\wedge}} a \mapsto_T x\}\ (\textsc{Read})$$

$$\Gamma \vdash \mathsf{assign}\, a\, y : \{a \mapsto x\}\text{-}\{z : \mathsf{unit} \mid a \mapsto y\}\ (\textsc{Assign})$$

$$\Gamma \vdash \mathsf{free}\, a : \{a \mapsto x\}\text{-}\{z : \mathsf{unit} \mid \mathsf{emp}\}\ (\textsc{Free})$$

$$\Gamma \vdash \mathsf{return}\, x^T : \{\mathsf{emp}\}\text{-}\{x : T \mid [x = M] \mathbin{\mathbb{\wedge}} \mathsf{emp}\}\ (\textsc{Return})$$

$$\frac{\Gamma \vdash M : \{P\}\text{-}\{x : T \mid Qx\} \qquad \forall x : T\, (\Gamma \vdash Nx : \{Qx\}\text{-}\{y : U \mid Ry\})}{\Gamma \vdash \mathsf{do}\ x \leftarrow M;\ N : \{P\}\text{-}\{y : U \mid Ry\}}\ (\textsc{Do})$$

$$\frac{\begin{array}{c}\Gamma \vdash M_1 : \{[x = \mathsf{true}] \mathbin{\mathbb{\wedge}} P\}\text{-}\{y : T \mid Qy\} \\ \Gamma \vdash M_2 : \{[x = \mathsf{false}] \mathbin{\mathbb{\wedge}} P\}\text{-}\{y : T \mid Qy\}\end{array}}{\Gamma \vdash \mathsf{if}\ x\ \mathsf{then}\ M_1\ \mathsf{else}\ M_2 : \{P\}\text{-}\{y : T \mid Qy\}}\ (\textsc{If})$$

$$\frac{\Gamma(f) = \Pi x : T.\, Sx}{\Gamma \vdash f : \Pi x : T.\, Sx}\ (\textsc{FVar}) \qquad \frac{\forall x : T\, (\Gamma, f : \Pi y : T.\, Sy) \vdash Mx : Sx}{\Gamma \vdash \mathsf{fix}\, f.\, M : \Pi x : T.\, Sx}\ (\textsc{FAbs})$$

$$\frac{\Gamma \vdash M : \Pi x : T.\, Sx \qquad y : T}{\Gamma \vdash \mathsf{app}\, M\, y : Sy}\ (\textsc{FApp})$$

$$\frac{\forall x : T\, (\Gamma \vdash M : Sx)}{\Gamma \vdash M : \forall x : T.\, Sx}\ (\forall I) \qquad \frac{\Gamma \vdash M : \forall x : T.\, Sx \qquad y : T}{\Gamma \vdash M : Sy}\ (\forall E)$$

$$\frac{\Gamma \vdash M : \{P\}\text{-}\{x : T \mid Q\} \qquad P' \models P \qquad \forall x : T\, (Qx \models Q'x)}{\Gamma \vdash M : \{P'\}\text{-}\{x : T \mid Q'\}}\ (\textsc{Cons})$$

$$\frac{\Gamma \vdash M : \{P\}\text{-}\{x : T \mid Qx\}}{\Gamma \vdash M : \{P * R\}\text{-}\{x : T \mid (Qx) * R\}}\ (\textsc{Frame})$$

$$\frac{\forall x : T\, (\Gamma \vdash M : \{Px\}\text{-}\{y : U \mid Qy\})}{\Gamma \vdash M : \{\exists\, x : T.\, Px\}\text{-}\{y : U \mid Qy\}}\ (\exists E)$$

Figure 4: Typing rules

An obvious drawback of this implementation of ghost variables is that they can span only one specification. The same restriction occurs in the type system from [5], the theoretical model of Ynot. In any case, establishing a link between a pre- and postcondition of a single specification is arguably the most important role of ghost variables.

$$
\begin{array}{rcll}
(h, \mathsf{new}\, x) & \triangleright & (h * [a \mapsto x], a) & a = \mathsf{freeAddr}(h) \\
(h, \mathsf{read}\, a) & \triangleright & (h, x) & h(a) = \mathsf{Some}\, x \\
(h, \mathsf{assign}\, a\, x) & \triangleright & (h[a := \mathsf{Some}\, x], ()) & a \in \mathrm{dom}(h) \\
(h, \mathsf{free}\, a) & \triangleright & (h[a := \mathsf{None}], ()) & a \in \mathrm{dom}(h) \\
(h, \mathsf{do}\ x \leftarrow \mathsf{return}\, x;\ M) & \triangleright & (h, Mx) & \\
(h, \mathsf{if\ true\ then}\ M_1\ \mathsf{else}\ M_2) & \triangleright & (h, M_1) & \\
(h, \mathsf{if\ false\ then}\ M_1\ \mathsf{else}\ M_2) & \triangleright & (h, M_2) & \\
(h, \mathsf{app}(\mathsf{fix}\, f.\, M)\, x) & \triangleright & (h, (Mx)[\mathsf{fix}\, f.\, M/f]) &
\end{array}
$$

$$
\frac{(h_1, M_1) \triangleright (h_2, M_2)}{(h_1, \mathsf{do}\ x \leftarrow M_1;\ N) \triangleright (h_2, \mathsf{do}\ x \leftarrow M_2;\ N)}
$$

$$
\frac{(h_1, M_1) \triangleright (h_2, M_2)}{(h_1, \mathsf{app}\, M_1\, x) \triangleright (h_2, \mathsf{app}\, M_2\, x)}
$$

Figure 5: Reductions

### 3.3 General Recursion

Coq is strongly normalizing system, which is important for its consistency. To add general recursion to our language, we had to manually program it. We added a constructor $\Pi^{\mathsf{fix}} : T \to \mathsf{Spec}$ denoting a dependent type of a recursive function. Variables, substitution and application are implemented in a standard way using de Bruijn indices.

As a result, the general form of the judgment is $\Gamma \vdash M : S$ where $\Gamma : \mathsf{list\ Spec}$, $M : \mathsf{Term}$ and $S : \mathsf{Spec}$.

### 3.4 Inference Rules

Inference rules of HTT shown in Figure 4 are similar to those in [5] and [1]. We added $(\exists E)$ rule, which is analogous to the left existential rule of sequent calculus. It is very convenient because one often encounters functions with existentially quantified preconditions. The variable $x$ in $(\exists E)$ is similar to ghost variables because it is not allowed to occur in the program.

Using this rule allowed us formulating rules for heap operations in a truly *small footprint* style. For example, the rule for read in [1] looks as follows (after slightly changing the syntax):

$$
\mathsf{read}\, p : \{\exists v.\, p \mapsto v * P(v)\}\text{-}\{v : T \mid p \mapsto v * P(v)\}
$$

Here the presence of an arbitrary predicate $P(v)$ violates small footprint approach and is not needed in our system.

# 4 Type Safety

In this section, we collect the main statements formally proved in Coq about HTT.

**Definition 4.1 (Nonterminal State)** A *state* is a pair of a heap and a term. A state $(h, M)$ is called *nonterminal* if there exists another state $(h', M')$ and $(h, M) \triangleright (h', M')$ according to the rules of Figure 5.

**Theorem 4.2** If $(h, M)$ is nonterminal and $h \# h'$, then $(h * h', M)$ is also nonterminal.

**Definition 4.3 (Nondeterministic Step)** Let the relation $\triangleright'$ be like $\triangleright$ where the line for $\mathsf{new}$ is replaced by the following one.

$$(h, \mathsf{new}\, x) \triangleright' (h * [a \mapsto x], a) \quad \text{for any } a \notin \mathrm{dom}(h)$$

Thus, the relation $\triangleright$ is deterministic, where new cells are allocated according to the function $\mathsf{freeAddr}$, while $\triangleright'$ may allocate any free address. It is obvious that $\triangleright \subseteq \triangleright'$.

**Theorem 4.4 (Frame Property)** Suppose $h_1 \# h$, $(h_1, M_1)$ is nonterminal, and $(h_1 * h, M_1) \triangleright' (h_2', M_2)$ for some $h_2', M_2$. Then there exists a heap $h_2$ such that $(h_1, M_1) \triangleright' (h_2, M_2)$, $h_2' = h_2 * h$ and $h_2 \# h$.

Frame Property asserts that a computational set in the big heap $h_1 * h$ can be simulated by a set in the small heap $h_1$, so that the excess portion $h$ is not touched. It has been known since the original days of separation logic (e.g., [7]) that memory allocation must be nondeterministic for Frame Property to hold. In turn, Frame Property is the basis on which the soundness of the (Frame) inference rule rests.

**Theorem 4.5 (Progress)** Suppose that $\vdash M : \{P\}\text{-}\{Q\}$ and that $h \models P$. Then either $(h, M)$ is nonterminal, or $M$ is $\mathsf{return}\, x$ for some term $x$.

**Theorem 4.6 (Subject Reduction)** Suppose that $\vdash M_1 : \{P_1\}\text{-}\{Q\}$, $h_1 \models P_1$ and $(h_1, M_1) \triangleright' (h_2, M_2)$. Then there exists an assertion $P_2$ such that $\vdash M_2 : \{P_2\}\text{-}\{Q\}$ and $h_2 \models P_2$.

**Lemma 4.7** Suppose that $x : T$ and $Q : T \to \mathsf{HProp}$. If $\vdash \mathsf{return}\, x : \{P\}\text{-}\{Q\}$, then $P \models Qx$.

**Corollary 4.8 (Partial Correctness)** If $\vdash M_1 : \{P_1\}\text{-}\{Q\}$, $h_1 \models P_1$ and $(h_1, M_1) \triangleright^* (h_2, \mathsf{return}\, x)$, where $\triangleright^*$ denotes the reflexive-transitive closure of $\triangleright$, then $h_2 \models Qx$.

It is interesting that Theorem 4.2 is only used in the proof of Progress, while Frame Property is used in Subject Reduction. Thus, Progress holds for the

deterministic allocator. Further, Subject Reduction quantifies over any nondeterministic step, including the one done using the freeAddr function. Therefore, Partial Correctness also holds for the deterministic allocator.

We conclude with a remark about the constructive character of the Progress and Subject Reduction theorems. The type Term has sort Type because via return $x$ a member of Term may contain any Coq object. On the other hand, the statement about derivability $\vdash M : \{P\}\text{-}\{Q\}$ has sort Prop. Therefore, its proof, which contains the HTT derivation, cannot be used in Coq to compute things of sort Type, including terms of our programming language.

This is made on purpose to demonstrate precisely where information contained in a typing derivation is used. It turns out that the proof of Progress proceeds by induction on the term, not on its typing derivation, and the information from the assumptions $\vdash M : \{P\}\text{-}\{Q\}$ and $h \models P$ is used only to prove that certain terms are well-typed. When an OCaml program is extracted from the proof of Progress, these assumptions disappear and the resulting function converts one program term into another.

In contrast, Subject Reduction theorem can well be nonconstructive, for example, when classical logic is used. The theorem takes and returns objects of sort Prop, and the proof proceeds by induction on the HTT derivation. For example, one assumption may be $h_1 \models P_1' * P_1''$, and the assertion $P_2$, which is built by the proof of the theorem, may depend on how $h_1$ is split to satisfy $P_1'$ and $P_1''$. There may or may not be a way to find how to split $h_1$ because this depends on the axioms used in a particular verification session. Nevertheless, even if a derivation includes the rule $(\exists E)$ and the existential precondition is proved nonconstructively, the program term from the next computational step will be found by the algorithm contained in the proof of the Progress statement.

# References


[1] Adam Chlipala, Gregory Malecha, Greg Morrisett, Avraham Shinnar, and Ryan Wisnesky. Effective interactive proofs for higher-order imperative programs. In *ICFP '09: Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*, August–September 2009.

[2] Adam Chlipala et al. Ynot project. http://ynot.cs.harvard.edu/.

[3] Aleksandar Nanevski, Greg Morrisett, and Lars Birkedal. Hoare type theory, polymorphism and separation. *Journal of Functional Programming*, 18(5–6):865–911, 2008.

[4] Aleksandar Nanevski, Greg Morrisett, Avraham Shinnar, Paul Govereau, and Lars Birkedal. Ynot: Reasoning with the awkward squad. In *ICFP '08: Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*, September 2008.

[5] Rasmus Lerchedahl Petersen, Lars Birkedal, Aleksandar Nanevski, and Greg Morrisett. A realizability model for impredicative hoare type theory. In *17th*

*European Symposium on Programming, ESOP 2008*, volume 4960 of *Lecture Notes in Computer Science*, pages 337–352. Springer, 2008.

[6] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of the Seventeenth Annual IEEE Symposium on Logic in Computer Science*, pages 55–74. IEEE, 2002.

[7] Hongseok Yang and Peter O'Hearn. A semantic basis for local reasoning. In Mogens Nielsen and Uffe Engberg, editors, *Proceedings of the 5th International Conference on Foundations of Software Science and Computation Structures (FOSSACS 2002)*, volume 2303 of *Lecture Notes in Computer Science*, pages 281–335, Grenoble, France, 2002. Springer.