Towards Type-theoretic Semantics for Transactional Concurrency

Aleksandar Nanevski
Microsoft Research, Cambridge
aleksn@microsoft.com

Paul Govereau Greg Morrisett
Harvard University
{govereau,greg}@eecs.harvard.edu

Abstract

We propose a dependent type theory that integrates programming, specifications, and reasoning about higher-order concurrent programs with shared transactional memory. The design builds upon our previous work on Hoare Type Theory (HTT), which we extend with types that correspond to Hoare-style specifications for transactions. The types track shared and local state of the process separately, and enforce that shared state always satisfies a given invariant, except at specific critical sections which appear to execute atomically. Atomic sections may violate the invariant, but must restore it upon exit. HTT follows Separation Logic in providing tight specifications of space requirements.

As a logic, we argue that HTT is sound and compositional. As a programming language, we define its operational semantics and show adequacy with respect to specifications.

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

General Terms Languages, Verification

1. Introduction

Transactional memory is one of the most promising directions in the evolution of concurrent programming languages. It replaces locks, conditional variables, critical regions and other low-level synchronization mechanism with a higher-level linguistic construct of transactions, which delegates synchronization to the run-time system. This frees the programmer from the need to develop potentially complicated and frequently non-modular synchronization protocols that arise in other approaches to concurrency. Transactions make it simpler to write efficient and correct concurrent programs that avoid data races and deadlock. Moreover, transactions are sufficiently well-behaved and compositional to fit naturally into a functional, higher-order language like Haskell [10].

In this paper we are interested not only in programming with transactions, but in developing a formal logic for specification and reasoning about concurrent programs with shared transactional memory. Most program logics for concurrency are based on Hoare Logic [27], and we choose as our starting point the recent work on concurrent Separation Logic [25, 3]. Concurrent Separation Logic has made significant inroads into reasoning about shared memory concurrency by focusing on the idea of spatial separation, whereby each process can be associated with a logical description of exclusive ownership over the state that it requires. This facilitates local reasoning, as the changes that a process makes to its local state do not influence others. Furthermore, Separation Logic leads to a particularly convenient descriptions of transferring ownership of state between processes.

When it comes to accessing shared resources, Separation Logic can specify the invariants that processes must preserve upon the exclusive use of the resource [25, 3]. Alternatively, it can specify upper and lower bounds on how the shared resource may change, in the style of rely-guarantee reasoning [31, 6].

While Separation Logic has significantly simplified the correctness proofs about shared state, it has mostly concerned itself with imperative *first-order* languages and *low-level synchronization primitives* such as locks. However, irrespective of whether one intends to prove his programs correct or not, programming with such low-level primitives remains difficult. In addition, first-order languages, by definition, do not support advanced linguistic features such as higher-order functions, polymorphism, modules, and abstract data types, even though all of these are indispensable for programming in the large as they facilitate code reuse, information hiding and modularity. The higher-order abstractions become all the more important if one wants to support specification and reasoning. Yet, most program logics based on (sequential or concurrent) Hoare or Separation Logic have little or no support for these important modularity features.

In this paper we take the step of combining programming, specification, and reasoning in the style of Separation Logic about *higher-order* programs with transactional concurrency. We build on our previous work on Hoare Type Theory (HTT) [22, 20], which is a dependent type theory with extensive support for programming and reasoning about side-effects related to state. Here, we extend HTT with concurrency and transactional primitives.

The main feature of HTT is the *Hoare type*, which takes the form $\mathsf{ST}\,\{P\}\,x{:}A\{Q\}$ and captures partial correctness within the type system. Hoare types classify programs that can be (sequentially) executed in a state satisfying the predicate P and either diverge, or converge to a result $x{:}A$ and a state satisfying Q. In the course of execution, such programs can perform memory reads, writes, allocations and deallocations. By capturing specifications as types, HTT makes it possible to abstract over and nest the specifications, combine them with the programs and data that they specify, or even build them algorithmically. All of these features significantly improve over the information hiding and code reuse facilities of Hoare Logic.

From the semantic standpoint, the Hoare type $\mathsf{ST}\,\{P\}\,x{:}A\{Q\}$ is a *monad* [18]. In this paper, we introduce yet another monadic family of Hoare types, which serves to encapsulate concurrent be-

TLDI'09, January 24, 2009, Savannah, Georgia, USA.

havior. The new Hoare types take the form $\mathsf{CMD}\,\{I\}\{P\}\,x{:}A\{Q\}$ and classify concurrent programs that execute in a *shared state* satisfying the invariant I, and *local state* satisfying the precondition P. Upon termination, the invariant on the shared state is preserved, but the local state is modified according to the predicate Q. The reader familiar with Haskell's implementation of transactional memory may benefit from the (imprecise) analogy by which Haskell's STM and IO monads correspond to our ST and CMD type families, respectively. For example, as in Haskell, CMD-computations can invoke ST-computations, and fork new CMD threads, but ST-computations are limited to state modifications, in order to facilitate optimistic techniques for implementing transactions.

Similar to Haskell, HTT monads separate the purely functional from the effectful, impure fragment. The pure fragment of HTT includes the Extended Calculus of Constructions (ECC) [13], which is a dependent type theory with support for abstraction over type universes and predicates in higher-order logic, and is the foundation behind the implementation of Coq [15]. For the purposes of this paper, however, we restrict attention to a much smaller fragment, which suffices to illustrate our concurrency extensions.

Our first technical contribution is the formulation of the logical connectives for describing the concurrent behavior. We argue that this logic is sound, and—most importantly—compositional. Just as in any type theory, compositionality is expressed by substitution principles, which guarantee that reasoning about HTT programs can be kept local in the sense that the typechecking and verification of a larger program only requires typechecking and verification of its sub-programs, and not any whole-program reasoning.

Just as any type theory, HTT is not only a program logic, but a programming language at the same time[1]. As the second contribution of the current paper, we endow the stateful and concurrent terms of HTT with operational semantics, and prove that this operational semantics is adequate for the intended interpretation of the Hoare types.

The rest of the paper is structured as follows. In Section 2 we introduce the basic stateful and transactional constructs, and illustrate how programs can be specified using Hoare types. In Section 3 we describe the formal syntax of the language, the connection with some well-known features from Hoare Logic, like ghost variables, and the definitions of the relational connectives that will serve to capture the semantics of state and concurrency. Section 4 presents the type system, and Section 5 describes the basic theorems about it. In Section 6 we introduce the operational semantics, and the proof of its adequacy. Section 7 discusses the related and future work, and Section 8 concludes.

2. Overview of monadic state and transactional memory

There are three conceptual levels in HTT: the purely functional fragment, the ST fragment, and the CMD fragment. As the name suggests, the pure fragment has no computational effects. The ST fragment includes sequential stateful commands $\mathsf{alloc}\,M$ (allocation), $!\,M$ (read), $M_1{:=}M_2$ (write), and $\mathsf{dealloc}\,M$ (deallocation). In addition, the ST fragment contains conditionals, and allows one to construct general recursive computations. The CMD fragment includes commands $\mathsf{atomic}\,E$ (atomically run the ST-computation E), and $x_1 \Leftarrow E_1 \| x_2 \Leftarrow E_2$ (run the two CMD computations E_1 and E_2 in parallel). The CMD fragment also includes a publish primitive which will be explained below, as well as constructors for conditionals and recursion.

[1] Hence exhibiting a variation on the Curry-Howard isomorphism.

The stateful sequential computations are classified by types of the form $\mathsf{ST}\,\{P\}\,x{:}A\{Q\}$, where P and Q are pre- and postconditions on the state. To illustrate these types, and their interaction with lambda abstraction and function types from the pure fragment, consider the function incBy, which takes a pointer l to a nat, a value n:nat, and then increments the contents of l by n. This function can be implemented as follows.

$$\begin{aligned}\mathsf{incBy}\ :\ &\Pi l{:}\mathsf{loc}.\,\Pi n{:}\mathsf{nat}.\\ &\quad [v{:}\mathsf{nat}].\,\mathsf{ST}\,\{l \mapsto_{\mathsf{nat}} v\}\,x{:}1\{l \mapsto_{\mathsf{nat}} v+n\}\\ &= \lambda l.\,\lambda n.\,\mathsf{stdo}\,(t \Leftarrow\, !\,l; l := t+n; \mathsf{return}\,(\,))\end{aligned}$$

The term syntax is chosen to closely resemble Haskell's do-notation, but also to support the meta-theoretic development (i.e., substitution principles). The keyword stdo encapsulates in its scope the stateful part of the code, separating it from the functional abstraction. The stateful code first reads from the location l and binds the obtained value to the (immutable) temporary variable t ($t \Leftarrow !\,l$), then writes back the increased value ($l := t+n$), before returning $(\,)$:1.

The type of incBy is a bit more involved: It specifies that incBy takes two arguments l:loc and n:nat, and returns a block of stateful code with Hoare type $[v{:}\mathsf{nat}].\,\mathsf{ST}\,\{l \mapsto_{\mathsf{nat}} v\}\,x{:}1\{l \mapsto_{\mathsf{nat}} v+n\}$. The notation $[v{:}\mathsf{nat}]$ is a syntactic sugar that will be explained in more detail in Section 3.7. Here, it suffices to say that the variable v serves to relate the value stored in the initial state of the monad, with the value at the ending state. The precondition $l \mapsto_{\mathsf{nat}} v$ requires that at the beginning of the stateful block, the location l *points to* v, and the postcondition ensures that at the end, l points to an incremented value. The variable v scopes over both the precondition and the postcondition, and in accordance with the standard Hoare logic terminology, we call v a *ghost variable*.

Concurrent computations are classified by types of the form $\mathsf{CMD}\,\{I\}\{P\}\,x{:}A\{Q\}$, where P and Q are pre- and post-conditions on the local state of the computation, and I is an invariant on the state that is shared with other processes. The key construct mediating access to shared state is the atomic primitive. It presents the programmer with the abstraction that the enclosed block of code executes sequentially, and in isolation from all the other parallel processes. Of course, our intention is that implementations of this system will not be so naive, and will extract parallelism through advanced run-time techniques such as software transactional memory. In particular, atomic blocks are optimistically run in parallel with the hope that the blocks will not perform conflicting changes to memory. To handle the case where there is a conflict, the run-time system aborts one of the conflicting blocks by rolling back its changes to the store and then re-starting the block.

Conceptually, the atomic primitive has the following type:

$$\mathsf{atomic} : \mathsf{ST}\,\{I * P\}\,x{:}A\{I * Q\} \to \mathsf{CMD}\,\{I\}\{P\}\,x{:}A\{Q\}$$

We can think of a thread running the command $\mathsf{atomic}\,M$, as acquiring a global lock on the shared state, executing the sequential code M, and then releasing the lock. During the atomic block, the thread is allowed to access both global and local state. Upon entry to the block, the global state is described by the invariant I, and the local state is described by P. Furthermore, we are guaranteed that the local and global state are disjoint through the use of the *separating conjunction* specification $I * P$. Throughout the execution of the atomic block, the thread is allowed to read and modify both the local and global state described by the specification. In particular, it can safely violate the invariant on the global state since no other thread can see the changes during the transaction. Furthermore, the thread is able to freely transfer locations from the local state to the global state and vice versa. Upon termination of the block, the thread must re-establish that the heap can be split into a local portion, now described by Q, and a global

portion once again described by the invariant I, resulting in a post-condition of $I * Q$. In summary, a sequential command with type $\mathsf{ST}\,\{I * P\}\,x{:}A\{I * Q\}$ can be lifted via atomic to a concurrent command with interface $\mathsf{CMD}\,\{I\}\{P\}\,x{:}A\{Q\}$.

As a simple example, consider the following definition:

$$\begin{array}{l}\mathsf{transfer} = \lambda l_1, l_2, n.\,\mathsf{cmdo}\\ \quad (t \Leftarrow \mathsf{atomic}(\\ \quad\quad t_1 \Leftarrow !\, l_1;\\ \quad\quad \mathsf{if}\ t_1 < n\ \mathsf{then\ return\ ff}\\ \quad\quad \mathsf{else}\ (\mathsf{decBy}\ l_1\ n;\ \mathsf{incBy}\ l_2\ n;\ \mathsf{return\ tt}));\\ \quad \mathsf{return}\ t)\end{array}$$

The transfer command attempts to atomically transfer the value n from location l_1 to location l_2, using the auxiliary commands incBy and decBy (not shown here). If l_1 holds a value less than n, then the transfer aborts and returns boolean ff, but if the transfer is successful, the command returns the boolean tt.

We can assign transfer a number of types, depending upon the correctness properties we wish to enforce. For example, in a banking application, we may wish to capture the constraint that the sum of the balances of the accounts must remain constant. That is, money can only be transferred, but not created or destroyed. In such a setting, we can use the following type:

$$\begin{array}{lll}\mathsf{transfer} & : & \Pi l_1{:}\mathsf{loc}.\,\Pi l_2{:}\mathsf{loc}.\,\Pi n{:}\mathsf{nat}.\\ & & \quad \mathsf{CMD}\,\{I(l_1,l_2)\}\{\mathsf{emp}\}\,x{:}\mathsf{bool}\{\mathsf{emp}\}\end{array}$$

where $I(l_1, l_2) = \exists v_1{:}\mathsf{nat}.\,\exists v_2{:}\mathsf{nat}.\,((l_1 \mapsto_{\mathsf{nat}} v_1) * (l_2 \mapsto_{\mathsf{nat}} v_2)) \wedge (v_1 + v_2 = k)$. Here, emp denotes an empty store, and $l \mapsto_\tau v$ denotes a store where location l points to a value v of type τ. Thus, the specification of transfer captures the invariant that the sum of the values in l_1 and l_2 must equal the constant k. Note that during the transfer, the invariant is violated, but is eventually restored. Thus, irrespective of the number of transfers executed between l_1 and l_2, the sum of the values stored into these locations always remains k. Note also that transfer operates only on shared state, and imposes no requirements on the local state. In particular, it can run even if the local state is empty, and any extensions of the local state will not be touched. In HTT, like in Separation Logic, this property is specified by using the predicate emp as a precondition, to tightly describe the local space requirements of the function.

We can now execute a number of transfers between l_1 and l_2 concurrently; the system will take care to preserve the invariant.

$$\begin{array}{l}\mathsf{transfer}_2\ :\ \Pi l_1{:}\mathsf{loc}.\,\Pi l_2{:}\mathsf{loc}.\\ \qquad\qquad \mathsf{CMD}\,\{I(l_1,l_2)\}\{\mathsf{emp}\}\,x{:}\mathsf{bool}\{\mathsf{emp}\}\\ \quad = \mathsf{cmdo}((t_1 \Leftarrow \mathsf{transfer}\ l_1\ l_2\ 10\ \|\\ \qquad\quad t_2 \Leftarrow \mathsf{transfer}\ l_2\ l_1\ 20);\\ \qquad\quad \mathsf{return}(t_1\ \mathsf{and}\ t_2))\end{array}$$

The above function forks two processes to concurrently execute two transfers, one between l_1 and l_2 and the other between l_2 and l_1. The values obtained as a result of each process are collected into variables t_1 and t_2, and the function returns tt if both transfers succeed.

2.1 Guarded commands

As a more interesting example, we next develop a function guard which waits in a busy loop until a provided location contains some required value[2]. The guard definition will be a function of four arguments, so that $\mathsf{guard}\ \alpha\ l\ n\ f$ reads the contents of location l, and loops until this contents equals n. Then it will execute the ST command f atomically, and return the obtained value of type α. For example, $\mathsf{guard}\ 1\ l_1\ 42\ (\mathsf{decBy}\ l_1 35)$ will wait until l_1 contains 42, and then decrement its contents by 35.

$$\begin{array}{l}\mathsf{guard} : \forall\alpha.\,\Pi l{:}\mathsf{loc}.\,\Pi n{:}\mathsf{nat}.\\ \qquad \mathsf{ST}\,\{(l \mapsto_{\mathsf{nat}} n) * J * P\}\,x{:}\alpha\{(l \mapsto_{\mathsf{nat}} -) * J * Q(x)\} \rightarrow\\ \qquad\quad \mathsf{CMD}\,\{(l \mapsto_{\mathsf{nat}} -) * J\}\{P\}\,x{:}\alpha\{Q(x)\}\end{array}$$

The return type of guard is a CMD-monad in order to allow other processes to concurrently set the value of l, while guard is busy waiting. Correspondingly, l should be a shared location, requiring the shared state invariant of the CMD-type to specify that $l \mapsto_{\mathsf{nat}} -$. Whatever the precondition P and postcondition Q on the local state this return type has, the ST-computation that is executed atomically should augment them with the knowledge that l is allocated, and that l contains value n at the beginning of the atomic execution. We further allow that the shared state may include an additional section described by the predicate J. This section can be modified by the ST-computation, as long as the validity of J is preserved. We make guard implicitly polymorphic in the predicates J, P and Q. In this paper we do not formally discuss polymorphism over predicates, but such a feature is inherent in ECC and Coq, and is easy to reconcile with impure extensions [20].

We split the implementation of guard into two parts. The function waitThen carries out the busy loop, but instead of immediately returning the result of the atomic execution, it stores this result into a temporary location r. Using waitThen, guard is implemented as follows.

$$\begin{array}{l}\mathsf{guard} = \Lambda\alpha.\,\lambda l.\,\lambda n.\,\lambda st.\\ \quad \mathsf{cmdo}\ (r \Leftarrow \mathsf{atomic}(t \Leftarrow \mathsf{alloc}\,0; \mathsf{return}\ t);\\ \quad\quad \mathsf{waitThen}\ \alpha\ l\ n\ st\ r;\\ \quad\quad t \Leftarrow \mathsf{atomic}(x \Leftarrow !\, r; \mathsf{dealloc}\ r; \mathsf{return}\ x);\\ \quad\quad \mathsf{return}\ t)\end{array}$$

The code first allocates the temporary location r, then waits on l, expecting the result of waiting to show up in r. Finally, it reads the result from r, and passes it out but only after r is deallocated. Notice that the accesses to store are always within an atomic block.

$$\begin{array}{l}\mathsf{waitThen} = \Lambda\alpha.\,\lambda l.\,\lambda n.\,\lambda st.\,\lambda r.\\ \quad \mathsf{cmdo}(t \Leftarrow \mathsf{fix}(\lambda c.\,\mathsf{cmdo}\\ \quad\quad (ok \Leftarrow \mathsf{atomic}(x \Leftarrow !\, l;\\ \quad\quad\quad \mathsf{if}\ (x = n)\ \mathsf{then}\\ \quad\quad\quad\quad y \Leftarrow st;\\ \quad\quad\quad\quad r := y;\\ \quad\quad\quad\quad \mathsf{return\ ff}\\ \quad\quad\quad \mathsf{else\ return\ tt});\\ \quad\quad \mathsf{if}\ ok\ \mathsf{then}\ x \Leftarrow c; \mathsf{return}\ x\\ \quad\quad \mathsf{else\ return}\ ()));\\ \quad \mathsf{return}\ t)\end{array}$$

Under the fixpoint, waitThen first atomically reads l, and based on the value, either executes st (by the command $y \Leftarrow st$), storing the result y into r, or simply exits the atomic block. Either way, it passes back via the flag ok the information about which branch was taken. If the contents of l was not appropriate, it goes around the loop again, by invoking the fixpoint computation c. Otherwise, r must contain the required value, so the function exits. The type of waitThen is

$$\begin{array}{l}\mathsf{waitThen} : \forall\alpha.\,\Pi l{:}\mathsf{loc}.\,\Pi n{:}\mathsf{nat}.\,\Pi r{:}\mathsf{loc}.\\ \qquad \mathsf{ST}\{(l \mapsto_{\mathsf{nat}} n) * J * P\}x{:}\alpha\{(l \mapsto_{\mathsf{nat}} -) * J * Q(x)\}\\ \qquad\quad \rightarrow \mathsf{CMD}\{(l \mapsto_{\mathsf{nat}} -) * J\}\{P * (r \mapsto_{\mathsf{nat}} 0)\}\,t{:}1\\ \qquad\qquad \{\exists x{:}\alpha.\,Q(x) * (r \mapsto_\alpha x)\}\end{array}$$

The return CMD type requires the existence of the location r in the local state, and guarantees that r contains the result of the atomic execution at the end. The later is ensured by the spatial conjunction with $Q(x)$ in the postcondition. Also, the first two components of the return type represents the loop invariant of the busy loop

[2] Of course, a real implementation will provide something like guard as a blocking primitive instead of encoding it via busy-waiting.

of waitThen. Essentially, throughout the iterations, we know that $(l \mapsto_{\mathsf{nat}} -) * J$ holds for the shared store and $P * (r \mapsto_{\mathsf{nat}} 0)$ holds for the local store.

2.2 Synchronizing variables

Using guard, we can now implement synchronizing variables (also know as "MVars" in Haskell). A synchronizing variable is a location in memory that either contains a value, or is empty, with two operations: put and take. The put operation will put a new value into an empty variable, and block otherwise. The take operation will block until a variable becomes full, then read the value from a full variable, emptying it.

We implement each synchronizing variable using two locations l and v in the shared heap, l for the empty/full flag, and v for the value. The invariant for the shared heap is $I_{sv}(l, v) = \exists n{:}\mathsf{nat}.\, ((l \mapsto_{\mathsf{nat}} n) \wedge (n = 0 \vee n = 1)) * (v \mapsto_A -)$, requiring that l points to a nat (0 for empty, 1 for full), and that v contains a value of a fixed type. Both put and take are CMD-computations over a shared heap described by I_{sv}. Since these operations only operate on the shared state, the pre- and postconditions on the local state are trivially emp. The implementations call the guard function, instantiated with $J = (v \mapsto_A -)$, and $P = Q = \mathsf{emp}$.

$$\begin{aligned}\mathsf{put} \;:\; &\Pi l{:}\mathsf{loc}.\, \Pi v{:}\mathsf{loc}.\, A \to \mathsf{CMD}\,\{I_{sv}(l,v)\}\{\mathsf{emp}\}\; x{:}1\{\mathsf{emp}\}\\ &= \lambda l.\, \lambda v.\, \lambda x.\, \mathsf{guard}\; 1\; l\; 0\; \mathsf{stdo}\; (l := 1; v := x; \mathsf{return}\; ())\end{aligned}$$

$$\begin{aligned}\mathsf{take} \;:\; &\Pi l{:}\mathsf{loc}.\, \Pi v{:}\mathsf{loc}.\, \mathsf{CMD}\,\{I_{sv}(l,v)\}\{\mathsf{emp}\}\; x{:}A\{\mathsf{emp}\}\\ &= \lambda l.\, \lambda v.\, \mathsf{guard}\; A\; l\; 1\; \mathsf{stdo}\; (l := 0; x \Leftarrow !\, v; \mathsf{return}\; x)\end{aligned}$$

We can test for fullness/emptiness, without blocking, by using the function empty, similar to the one provided in Haskell standard libraries [10].

$$\begin{aligned}\mathsf{empty} \;:\; &\Pi l{:}\mathsf{loc}.\, \Pi v{:}\mathsf{loc}.\, \mathsf{CMD}\,\{I_{sv}(l,v)\}\{\mathsf{emp}\}\; x{:}\mathsf{nat}\{\mathsf{emp}\}\\ &= \lambda l.\, \lambda v.\, \mathsf{cmdo}(t \Leftarrow \mathsf{atomic}(x \Leftarrow !\, l; \mathsf{return}\; x);\\ &\qquad\qquad \mathsf{return}\; t)\end{aligned}$$

2.3 Producer-consumer pattern

HTT includes an additional concurrency primitive publish J, which logically takes a part of the local state described by the predicate J and moves it into the shared state. A computation may need to perform this operation if it wants to spawn some child processes to execute concurrently on the given local state. We illustrate the primitive by building a producer-consumer pattern, whereby we allocate a new synchronizing variable, publish it as shared state, and launch two processes which communicate via the now shared variable. The shared variable becomes a primitive communication channel between the processes.

Suppose that we have a producer function, p, and a consumer function, c. Then, we can easily construct functions which read from and write to a shared variable using p and c.

$$\begin{aligned}\mathsf{produce} = \lambda l.\, \lambda v.\, \mathsf{cmdo}(&t = \mathsf{fix}\; \lambda f.\, \mathsf{cmdo}(x \Leftarrow p; \mathsf{put}\; l\; v\; x;\\ &\qquad\qquad s \Leftarrow f; \mathsf{return}\; s);\\ &q \Leftarrow t; \mathsf{return}\; q)\end{aligned}$$

$$\begin{aligned}\mathsf{consume} = \lambda l.\, \lambda v.\, \mathsf{cmdo}(&t = \mathsf{fix}\; \lambda f.\, \mathsf{cmdo}(x \Leftarrow \mathsf{take}\; l\; v; c\; x;\\ &\qquad\qquad s \Leftarrow f; \mathsf{return}\; s);\\ &q \Leftarrow t; \mathsf{return}\; q)\end{aligned}$$

Here, p and c both obtain a CMD-computations with a shared invariant $I_{sv}(l, v)$. In order to use produce and consume, we must first establish this invariant; this is where publish comes in.

$$\begin{aligned}&\mathsf{cmdo}(l \Leftarrow \mathsf{atomic}(t \Leftarrow \mathsf{alloc}\; 0; \mathsf{return}\; t);\\ &\quad v \Leftarrow \mathsf{atomic}(t \Leftarrow \mathsf{alloc}\; a; \mathsf{return}\; t);\\ &\quad \mathsf{publish}(I_{sv}(l, v));\\ &\quad x_1 \Leftarrow \mathsf{produce}\, l\, v \parallel x_2 \Leftarrow \mathsf{consume}\, l\, v;\\ &\quad \mathsf{return}\; ())\end{aligned}$$

Types	A, B, τ	$::=$	$\alpha \mid \mathsf{bool} \mid \mathsf{nat} \mid 1 \mid \Pi x{:}A.\, B \mid \forall \alpha.\, A \mid$ $\mathsf{ST}\,\{P\}\, x{:}A\{Q\} \mid \mathsf{CMD}\,\{I\}\{P\}\, x{:}A\{Q\}$
Predicates	P, Q, R, I	$::=$	$\mathsf{id}_A(M, N) \mid \mathsf{seleq}_\tau(H, M, N) \mid \top \mid$ $\bot \mid P \wedge Q \mid P \vee Q \mid P \supset Q \mid \neg P \mid$ $\forall x{:}A.\, P \mid \forall \alpha.\, P \mid \forall h{:}\mathsf{heap}.\, P \mid$ $\exists x{:}A.\, P \mid \exists \alpha.\, P \mid \exists h{:}\mathsf{heap}.\, P$
Heaps	H, G	$::=$	$h \mid \mathsf{empty} \mid \mathsf{upd}_\tau(H, M, N)$
Terms	K, L, M, N	$::=$	$x \mid \mathsf{tt} \mid \mathsf{ff} \mid \bar{n} \mid M \oplus N \mid (\,) \mid \lambda x.\, M \mid K\; M \mid$ $\Lambda \alpha.\, M \mid K\; \tau \mid \mathsf{stdo}\; E \mid \mathsf{cmdo}\; E \mid M : A$
Computations	E, F	$::=$	$\mathsf{return}\; M \mid x \Leftarrow K; E \mid x \Leftarrow !_\tau\, M; E \mid$ $M :=_\tau N; E \mid x \Leftarrow \mathsf{alloc}_\tau\, M; E \mid$ $\mathsf{dealloc}\, M; E \mid x \Leftarrow \mathsf{atomic}_{A,P,Q}\, E_1; E \mid$ $(x_1{:}A_1 \Leftarrow E_1{:}P_1 \parallel x_2{:}A_2 \Leftarrow E_2{:}P_2); E \mid$ $\mathsf{publish}\; I; E \mid x \Leftarrow \mathsf{fix}_A\, M; E \mid$ $x \Leftarrow \mathsf{if}_A\, M\; \mathsf{then}\; E_1\; \mathsf{else}\; E_2; E$
Contexts	Δ	$::=$	$\cdot \mid \Delta, x{:}A \mid \Delta, \alpha \mid \Delta, h{:}\mathsf{heap} \mid \Delta, P$

Figure 1. Syntax of HTT.

After allocating l and v, we publish them with the invariant I_{sv}. After the publish, produce and consume can execute in parallel, and each has access to l and v as shared state.

3. Formal syntax and definitions

In this section we present the syntax of HTT (Figure 1), and discuss the constructs in more detail.

3.1 Types

In addition to the already described Hoare types, HTT admits the types of booleans and natural numbers, dependent function types $\Pi x{:}A.\, B$, and polymorphic quantification $\forall \alpha.\, A$. The type variables α in polymorphic quantification ranges over monomorphic types only, as customary in, say, Standard ML (SML). Thus, the current paper only supports predicative polymorphism, although extending HTT with impredicativity is possible [28]. As usual with dependent types, we write $A \to B$ instead of $\Pi x{:}A.\, B$, when the type B does not depend on x. We omit the other useful type constructors from pure type theories, such as Σ-types and inductive types as their interaction with the impure features does not present any theoretical problems. For example, we have studied such extensions in our previous work on sequential HTT in [20, 23].

3.2 Terms

The purely functional fragment consists of the usual term constructors: boolean values, numerals $\bar{n}$ and the basic arithmetic operations (collectively abbreviated as $M \oplus N$), the unit value $(\,)$:1, lambda abstraction and application, and type abstraction and application. We do not annotate lambda abstractions with the domain types, but instead provide a constructor $M{:}A$ to ascribe a type A to the term M. The stdo and cmdo constructors are the introduction forms for the corresponding Hoare types. They are analogous to the monadic-do in Haskell, except that we have separate constructor for each monad, to avoid any confusion.

3.3 Computations

The scope of stdo and cmdo is a computation, which is a semicolon separated list of commands, terminating with a return value. We have already described the intuition behind most of the constructors in Section 2. However, some of these require explicit annotations with types, pre/postconditions and invariants, which were omitted before, so we now revisit them with the additional details.

For example, HTT supports strong updates by which a location pointing to a value of type τ_1 may be updated with a value of some other type τ_2. Correspondingly, the ST primitives for reading,

writing and allocation must be annotated with the type of the manipulated value.

CMD-computations are annotated as follows. (1) $(x_1{:}A_1 \Leftarrow E_1{:}P_1 \parallel x_2{:}A_2 \Leftarrow E_2{:}P_2)$ forks two parallel child processes E_1 and E_2. Upon their termination, the processes are joined, that is, their return results are bound to x_1 and x_2, respectively, their private space is returned to the parent process, and the execution proceeds to the subsequent command of the parent process. The $\parallel$ command explicitly requires the return types A_1 and A_2 and the preconditions P_1 and P_2 on the parallel processes. The preconditions indicate which part of the local state of the parent process is delegated to each of E_1 and E_2. The split of the local state must be disjoint. If the two processes are supposed to share state, then that state must be declared as shared in the invariant of the CMD type. (2) $x \Leftarrow \mathsf{atomic}_{A,P,Q}\, E_1$ explicitly requires the return type A of E_1 as well as the precondition P and postcondition Q on the local state that E_1 manipulates. This local state will be joined with the shared state of the parent process, before E_1 executes atomically. (3) publish I does not require additional annotation beyond the predicate I. However, we mention here that I must be *precise*, in the sense that it uniquely defines the heap fragment that should be published. For example, the predicate $(x \mapsto_{\mathsf{nat}} - * y \mapsto_{\mathsf{nat}} -)$ is precise, and would correspond to publishing the locations x and y. On the other hand, the predicate $\top$ is not precise, as it holds of every heap. Precision is customarily required of shared state invariants in Separation Logic [3, 25].

Finally, the conditional and the fix constructs are present in both monads. The conditional is annotated with the expected types of its branches. Fix is annotated with the type A, and computes the least fixed point of the function $M{:}A{\to}A$. Here A must be a Hoare type, in order to guarantee that all uses of recursion (and hence, potential occasions for non-termination) appear under the guard of stdo or cmdo. Thus, non-termination in HTT is considered an effect, and is encapsulated under the monad, in order to preserve the logical properties of the underlying pure calculus. In particular, the encapsulation prevents the recursion from unrolling during normalization of terms.

3.4 Heaps

In HTT we model heap locations by natural numbers, although in the examples we write loc instead of nat to emphasize when a natural number is used as a pointer. Heaps are modeled as functions mapping a location N to a pair (τ, M) where $M{:}\tau$. In this case, we say that N *points to* M, or that M is the *contents* of N. The type τ is required to be monomorphic, in order to preserve the predicativity of the type theory. However, τ can be a dependent function type, as well as a Hoare type. Thus, heaps in HTT are *higher-order*, albeit predicative.

Syntactically, we build heaps out of the following primitives: (1) empty stands for the empty heap, that is, a nowhere defined function. (2) $\mathsf{upd}_\tau(H, M, N)$ is a function which returns $N{:}\tau$ at argument M, but equals H at other arguments. It models the heap obtained from H by writing $N{:}\tau$ into the address M.

3.5 Predicate logic and heap semantics

For the treatment of our concurrency primitives, it suffices to focus on a first-order, polymorphic, predicate logic over heaps. Aside from the usual connectives of first-order logic, we provide two primitives: (1) $\mathsf{id}_A(M, N)$ is the equality at type A. We will frequently abbreviate it as $M =_A N$ or simply $M = N$. (2) $\mathsf{seleq}_\tau(H, M, N)$ reflects the semantics of heaps into the assertion logic of the Hoare types. It holds iff the heap H contains $N{:}\tau$ at location M. The following axioms relate seleq and update.

$$\neg\mathsf{seleq}_\tau(\mathsf{empty}, M, N)$$
$$\mathsf{seleq}_\tau(\mathsf{upd}_\tau(H, M, N), M, N)$$
$$M \neq M_2 \wedge \mathsf{seleq}_\tau(\mathsf{upd}_\sigma(H, M_1, N_1), M_2, N_2) \supset \mathsf{seleq}_\tau(H, M_2, N_2)$$
$$\mathsf{seleq}_\tau(H, M, N_1) \wedge \mathsf{seleq}_\tau(H, M, N_2) \supset N_1 =_\tau N_2$$

The first axiom states that an empty heap does not contain any assignments. The second and the third are the well-known McCarthy axioms for functional arrays [17]. The fourth axiom asserts a version of heap functionality: a heap may assign at most one value to a location, for each given type. The fourth axiom is slightly weaker than expected, as we would like to state that a heap assigns at most one type and value to a location. This is easily expressible in the extension of HTT with higher-order logic and an equality predicate on types [20].

3.6 Separation logic

Given the heaps as above, we can now define predicates expressing heap equality, disjointness, and disjoint union of heaps [22].

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

In English, $\supset$ is logical implication, $\subset\supset$ is logical equivalence, $H_1 = H_2$ is heap equality, $M \in H$ iff the heap H assigns to the location M, share states that H_1 and H_2 agree on the location M, and splits states that H can be split into disjoint heaps H_1 and H_2.

We next formally define the assertions familiar from Separation Logic [24]. All of these are relative to the free variable m, which denotes the current heap fragment of reference. We will call predicates with one free heap variable m *unary predicates*, and use letters P, R, S and I to range over them. Given a unary predicate P, we will customarily use the syntax for functional application, and write $P\ H$ as an abbreviation for $[H/\mathsf{m}]P$.

$$\begin{aligned}
\mathsf{emp} &= \mathsf{m} = \mathsf{empty} \\
M \mapsto_\tau N &= \mathsf{m} = \mathsf{upd}_\tau(\mathsf{empty}, M, N)) \\
M \hookrightarrow_\tau N &= \mathsf{seleq}_\tau(\mathsf{m}, M, N) \\
P * S &= \exists h_1{:}\mathsf{heap}.\exists h_2{:}\mathsf{heap}.\ \mathsf{splits}(\mathsf{m}, h_1, h_2) \wedge P\ h_1 \wedge S\ h_2 \\
P \mathrel{-\!\!*} S &= \forall h_1{:}\mathsf{heap}.\forall h_2{:}\mathsf{heap}.\ \mathsf{splits}(h_2, h_1, \mathsf{m}) \supset P\ h_1 \supset S\ h_2 \\
\mathsf{this}\, H &= \mathsf{m} = H \\
\mathsf{precise}\ P &= \forall h_1, h_1', h_2, h_2'{:}\mathsf{heap}.\ \mathsf{splits}(\mathsf{m}, h_1, h_1') \supset \mathsf{splits}(\mathsf{m}, h_2, h_2') \supset P\ h_1 \supset P\ h_2 \supset h_1 = h_2
\end{aligned}$$

We have already given the informal descriptions of emp, $M \mapsto_\tau N$ and $P * S$ in Section 2. $M \hookrightarrow_\tau N$ iff current heap contains *at least* the location M pointing to $N{:}\tau$. $P \mathrel{-\!\!*} S$ holds iff any extension of the current heap by a heap satisfying P, produces a heap satisfying S. this (H) iff the current heap equals H. Concerning the last predicate, precise P holds iff for any given heap m, there is *at most one* subheap h such that $P\ h$.

With these definitions, it should now be apparent that the preconditions, postconditions and invariants in our Hoare types are predicates over heaps, and that they implicitly depend on the heap variable m. For example, the type $\mathsf{ST}\,\{\mathsf{emp}\}\,x{:}A\{\mathsf{emp}\}$ really equals $\mathsf{ST}\,\{\mathsf{m} = \mathsf{empty}\}\,x{:}A\{\mathsf{m} = \mathsf{empty}\}$, where m in the precondition denotes the initial heap, and m in the postcondition de-

notes the ending heap of a stateful computation. Thus, the two m variables are really different. If we wanted to make the scope of m explicit, we would write the type $\mathsf{ST}\,\{P\}\,x{:}A\{Q\}$ explicitly as

$$\mathsf{ST}\,\{\mathsf{m}.\,P\}\,x{:}A\{\mathsf{m}.\,Q\}$$

However, in order to reduce clutter, we leave the bindings of m implicit. We adopt a similar strategy for $\mathsf{CMD}\,\{I\}\{P\}\,x{:}A\{Q\}$, where I also depends on an implicitly bound variable m.

3.7 Ghost variables and binary postconditions

The programs from Section 2 already exhibit that the use of Hoare types frequently requires a set of ghost variables that scope over the precondition and the postcondition in order to relate the two. For example, the program incBy with the type

$$\mathsf{incBy} : \Pi l{:}\mathsf{loc}.\,\Pi n{:}\mathsf{nat}.\,[v{:}\mathsf{nat}].\,\mathsf{ST}\,\{l \mapsto_{\mathsf{nat}} v\}\,x{:}1\{l \mapsto_{\mathsf{nat}} v + n\}$$

needs the ghost variable v to name the value initially stored into l. One may entertain the possibility of treating v as an ordinary, Π-bound variable and re-type incBy as:

$$\begin{array}{rcl}\mathsf{incBy}' & : & \Pi l{:}\mathsf{loc}.\,\Pi n{:}\mathsf{nat}.\,\Pi v{:}\mathsf{nat}.\\ & & \quad \mathsf{ST}\,\{l \mapsto_{\mathsf{nat}} v\}\,x{:}1\{l \mapsto_{\mathsf{nat}} v + n\}\end{array}$$

This is not a good solution, however, as Π-abstraction will require every caller of incBy' to instantiate v at run-time. The ghost variable v, which should serve only the purpose of logically connecting the precondition and the postcondition of the Hoare type, suddenly acquires a computational significance; it has to be explicitly supplied by the caller, and the value, when instantiated, has to produce a precondition that is true at the given program point. More concretely, in order to increment the contents of l by executing incBy', the caller must already know what the value stored in l is. This, of course, makes the usefulness of incBy' quite dubious. If the caller already knows the stored value, why not simply write its increment back into l directly?

A better alternative, and the one that we adopt here, is to allow that postconditions not only depend on the variable m denoting the current heap at the end of the computation, but also on the variable i that denotes the initial heap. That is, if we made the scopes explicit, then the type $\mathsf{ST}\,\{P\}\,x{:}A\{Q\}$ would be written as $\mathsf{ST}\,\{\mathsf{m}.\,P\}\,x{:}A\{\mathsf{i}.\,\mathsf{m}.\,Q\}$. The second heap variable in the postcondition can be used to relate the values stored in the initial heap, to the values stored in the ending heap. The type of incBy may be written as

$$\begin{array}{l}\mathsf{incBy}'' : \Pi l{:}\mathsf{loc}.\,\Pi n{:}\mathsf{nat}.\,\mathsf{ST}\{\exists v.\,l \mapsto_{\mathsf{nat}} v\}\,r : 1\\ \qquad\qquad\qquad\qquad \{\forall v.\,(l \mapsto_{\mathsf{nat}} v)\,\mathsf{i} \supset (l \mapsto_{\mathsf{nat}} v)\,\mathsf{m}\}\end{array}$$

Under this binding convention, the syntax of Hoare types with ghost variables becomes just a syntactic sugar. The Hoare type $[\Delta].\,\mathsf{ST}\,\{P_1\}\,x{:}A\{P_2\}$, where Δ is a variable context, and P_1,P_2 are unary predicates over m, can be desugared into

$$\mathsf{ST}\,\{\exists\Delta.\,P_1\}\,x{:}A\{\forall\Delta.\,P_1\,\mathsf{i} \supset P_2\,\mathsf{m}\}$$

Similarly, the Hoare type $[\Delta].\,\mathsf{CMD}\,\{I\}\{P_1\}\,x{:}A\{P_2\}$ is desugared into $\mathsf{CMD}\,\{I\}\{\exists\Delta.\,P_1\}\,x{:}A\{\forall\Delta.\,P_1\,\mathsf{i} \supset P_2\,\mathsf{m}\}$. In the rest of the paper, we will use the described convention on ghost variables in order to abbreviate the Hoare types that appear in our examples. However, in the development of the meta theory of HTT, we will assume that postconditions in Hoare types depend on two heap variables: i which denotes the initial heap, and m which denotes the ending heap of the computation.

We call predicates that depend on both i and m *binary predicates*, and use Q and T to range over them. We use X to range over either unary or binary predicates. We will again use the syntax of functional application and write $Q\;H_1\;H_2$ as an abbreviation for $[H_1/\mathsf{i}, H_2/\mathsf{m}]Q$.

We next define several operators on binary predicates that will have a prominent role in the semantics of HTT.

$$\begin{array}{rcl}\delta P & = & P \wedge \mathsf{i} = \mathsf{m}\\ \nabla P & = & P \wedge \mathsf{i} = \mathsf{i}\\ \Box P & = & P\,\mathsf{i} \wedge P\,\mathsf{m}\\ X \circ Q & = & \exists h{:}\mathsf{heap}.\,[h/\mathsf{m}]X \wedge Q\;h\;\mathsf{m}\\ Q_1 \mathbin{*\!\!*} Q_2 & = & \exists i_1, i_2, m_1, m_2{:}\mathsf{heap}.\\ & & \quad \mathsf{splits}(\mathsf{i}, i_1, i_2) \wedge \mathsf{splits}(\mathsf{m}, m_1, m_2) \wedge\\ & & \quad Q_1\;i_1\;m_1 \wedge Q_2\;i_2\;m_2\\ P \multimap Q & = & \forall i_0, h{:}\mathsf{heap}.\,\mathsf{splits}(\mathsf{i}, i_0, h) \supset P\;i_0 \supset\\ & & \quad \exists m_0.\,\mathsf{splits}(\mathsf{m}, m_0, h) \wedge Q\;i_0\;m_0\\ P_1\;P_2 \multimap\!\!\circ\, Q_1\;Q_2 & = & \forall i_0, h{:}\mathsf{heap}.\,\mathsf{splits}(\mathsf{i}, i_0, h) \supset\\ & & \quad \forall i_1, i_2{:}\mathsf{heap}.\,\mathsf{splits}(i_0, i_1, i_2) \supset\\ & & \quad\quad P_1\;i_1 \supset P_2\;i_2 \supset\\ & & \quad\quad\quad \exists m_0, m_1, m_2.\,\mathsf{splits}(m, m_0, h) \wedge\\ & & \quad\quad\quad\quad \mathsf{splits}(m_0, m_1, m_2) \wedge\\ & & \quad\quad\quad\quad\quad Q_1\;i_1\;m_1 \wedge Q_2\;i_2\;m_2\\ M\,?\,Q_1\;Q_2 & = & (M = \mathsf{tt} \supset Q_1) \wedge (M = \mathsf{ff} \supset Q_2)\end{array}$$

In English, δP extends the unary predicate P to binary, diagonal one. ∇P is also a binary predicate, albeit one that holds for *any* domain heap (it ignores the variable i). $\Box P$ requires that P holds for both the domain and the range heaps, but unlike δ, does not require that the two heaps are actually equal. $X \circ Q$ is a relational composition. The predicate $Q_1 \mathbin{*\!\!*} Q_2$ is the generalization of separating conjunction to the binary case. It holds if both domain and range heaps can be split in two, so that Q_1 relates the first halfs and Q_2 relates the second halfs. $P \multimap Q$ is a binary predicate relating the heap i with m only if m can be obtained by replacing any subheap of i satisfying P with a subheap related by Q. $P_1\;P_2 \multimap\!\!\circ\, Q_1\;Q_2$ is the generalization of $P \multimap Q$. It pairwise replaces P_1 according to Q_1 and P_2 according to Q_2 to obtain the heap m starting from i. $M\,?\,Q_1\;Q_2$ is the relational version of a conditional.

Example. The binary relation $(l \mapsto_{\mathsf{nat}} v) \multimap \nabla(l \mapsto_{\mathsf{nat}} v + 1)$ holds between two heaps i and m if and only if m can be obtained from i by replacing *all* parts of i satisfying $l \mapsto_{\mathsf{nat}} v$ (and there can be at most one such part), with a part satisfying $l \mapsto_{\mathsf{nat}} v + 1$. Such a relation therefore directly captures the semantics of an ST computation that increments the contents of l.

4. Type system

The easiest way to support reasoning about effectful computations is to translate them into some mathematical entity that is already supported by the underlying type theory. In this paper, we have chosen to translate effectful computations into *binary relations on heaps*, so that a computation may be viewed as relating its initial to its ending heap. Choosing relations for the modeling of Hoare types has the additional benefit that we can then also represent partial and non-deterministic computations; that is, computations with no result, or computations with more than one result, respectively. The translation of computations into relations is performed by the typing rules, which formalize the well-known idea of calculating strongest postcondition [5].

We will have four typing judgments for computations, two for ST-computations and two for CMD-computations. The ST judgments are $\Delta; P \vdash E \Rightarrow x{:}A.\,Q$ and $\Delta; P \vdash E \Leftarrow x{:}A.\,Q$. The first judgment takes a unary predicate P and a computation E, and generates the binary predicate Q that most tightly captures the semantics of E (i.e., Q is the strongest postcondition). In the process, the rule also verifies that the return result of E has type A, where A is supplied as input to the judgment. The second judgment checks that Q is a postcondition, not necessarily the strongest one for E with respect to P.

The CMD judgments are, similarly, $\Delta; I; P \vdash E \Rightarrow x{:}A.\,Q$ and $\Delta; I; P \vdash E \Leftarrow x{:}A.\,Q$, except that here P and Q are a pre- and post-condition on the local state of E, while the unary predicate

I keeps the invariant on the state that E shares with other processes. By formation, I is required to be precise.

We will make use of further several judgments: (1) $\Delta \vdash K \Rightarrow A$ takes a pure term K and generates its type if it can; (2) $\Delta \vdash M \Leftarrow A$ checks that M has type A. These two judgments implement bidirectional typechecking for the pure fragment. (3) $\Delta \vdash P$ checks that the predicate P is true. It is a completely standard natural deduction for polymorphic first-order logic with equality, except that it also formalizes heaps, via the four axioms listed in Section 3. (4) $\Delta \vdash A \Leftarrow \mathsf{type}$ and $\Delta \vdash P \Leftarrow \mathsf{prop}$ are type and predicate formation judgments, and (5) $\Delta \vdash \tau \Leftarrow \mathsf{mono}$ checks that τ is a monomorphic type. The last three judgments are fairly obvious, so we omit them here.

4.1 Typechecking ST-computations

We start with a structural rule which relates the synthesis and checking of postconditions: if Q' is a strongest postcondition, and from knowing Q' we can derive Q, then Q is a postcondition.

$$\frac{\Delta; P \vdash E \Rightarrow x{:}A.\, Q' \qquad \Delta, x{:}A, \mathsf{i}, \mathsf{m{:}heap}, \delta P \circ Q' \vdash Q}{\Delta; P \vdash E \Leftarrow x{:}A.\, Q}$$

Rather than simply taking Q' as a hypothesis when trying to derive Q, the rule takes $\delta P \circ Q'$. Unrolling the definitions of $\circ$ and δ, this basically injects the knowledge that the initial heap of Q also satisfies P, which should be expected as P is a precondition that the checking starts with. This rule essentially implements the law of consequence well-known in Hoare Logic.

The typing rule for monadic unit in a sense corresponds to a rule for assignment to the variable x found in the classical formulations of Hoare Logic:

$$\frac{\Delta \vdash M \Leftarrow A}{\Delta; P \vdash \mathsf{return}\ M \Rightarrow x{:}A.\, \delta(x = M)}$$

The postcondition $\delta(x = M)$ simply states that after executing $\mathsf{return}\ M$ the return value $x = M$ and the initial and the ending heap are equal since no state has been modified.

$$\frac{\Delta \vdash \tau \Leftarrow \mathsf{mono} \qquad \Delta \vdash M \Leftarrow \mathsf{nat} \qquad \Delta, \mathsf{m{:}heap}, P \vdash M \hookrightarrow_\tau - \qquad \Delta, x{:}\tau; P \circ \delta(M \hookrightarrow_\tau x) \vdash E \Rightarrow y{:}B.\, Q}{\Delta; P \vdash x \Leftarrow !_\tau\, M; E \Rightarrow y{:}B.\, (\exists x{:}\tau.\, \delta(M \hookrightarrow_\tau x) \circ Q)}$$

The rule for memory read must check that τ is a well-formed monomorphic type, as only values of monomorphic types can be stored into heaps. Further, the location M must be a natural number, and M must point to a value of type τ. The later is ensured by the entailment $P \vdash M \hookrightarrow_\tau -$, which may be seen as a *verification condition* that needs to be discharged in order to guarantee the correctness of the program. The continuation E is then checked in a context extended with variable $x{:}\tau$, and the precondition for checking E must appropriately reflect the knowledge that x binds the value read from M. This is achieved by composing P with $\delta(M \hookrightarrow_\tau x)$. Alternatively, we could have used the equivalent $P \wedge (M \hookrightarrow_\tau x)$, which is the standard postcondition for memory lookup, but we choose the current formulation in order to emphasize the compositional nature of typechecking. For example, after the relation Q corresponding to E is obtained, we need to lift it to include the semantics of the lookup, before we return it as a postcondition generated for the original computation. We do so by composing $\delta(M \hookrightarrow_\tau x) \circ Q$. One can now see the important intuition that, in general, the strongest postcondition generated for some computation E always has a form of an ordered sequence of compositions of smaller relations, each of which precisely captures the primitive effectful commands of E, in the order in which they appear in E. This substantiates our claim that typechecking simply translates the E into a relation (equivalently, predicate). In fact, the translation is almost literal, as the structure of the obtained predicate completely mirrors E.

Memory writes follow a similar strategy.

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

To write into the location M, we first ensure that it is allocated (verification condition $P \vdash M \hookrightarrow -$). Then the continuation E is checked with respect to a predicate $P \circ (M \mapsto - \multimap \nabla(M \mapsto_\tau N))$. Intuitively, following the definition of the connective $\multimap$, this predicate "replaces" a portion of the heap satisfying $M \mapsto -$ by a heap satisfying $M \mapsto_\tau N$, while preserving the rest of the structure described by P. Thus, the predicate correctly models the semantics of memory lookup.

The idea behind the typechecking of stateful commands should now be obvious, so we simply display the rules for allocation and deallocation without further comment.

$$\frac{\Delta \vdash \tau \Leftarrow \mathsf{mono} \qquad \Delta \vdash M \Leftarrow \tau \qquad \Delta, x{:}\tau; P \circ (\mathsf{emp} \multimap \nabla(x \mapsto_\tau M)) \vdash E \Rightarrow y{:}B.\, Q}{\Delta; P \vdash x \Leftarrow \mathsf{alloc}_\tau\, M; E \Rightarrow y{:}B.\, (\exists x{:}\tau.\, (\mathsf{emp} \multimap \nabla(x \mapsto_\tau M)) \circ Q)}$$

$$\frac{\Delta \vdash M \Leftarrow \mathsf{nat} \qquad \Delta, \mathsf{m{:}heap}, P \vdash M \hookrightarrow - \qquad \Delta; P \circ (M \mapsto - \multimap \nabla \mathsf{emp}) \vdash E \Rightarrow x{:}A.\, Q}{\Delta; P \vdash \mathsf{dealloc}\, M; E \Rightarrow x{:}A.\, (M \mapsto - \multimap \nabla \mathsf{emp}) \circ Q}$$

4.2 Typechecking CMD-computations

The CMD-judgments have similar rules for consequence and unit as the one presented in the ST case. We omit these here, and focus instead on the primitives for concurrency.

$$\frac{\Delta; \mathsf{m{:}heap}, P \vdash P_1 * P_2 * \top \qquad \Delta; I; P_1 \vdash E_1 \Rightarrow x_1{:}A_1.\, Q_1 \qquad \Delta; I; P_2 \vdash E_2 \Rightarrow x_2{:}A_2.\, Q_2 \qquad \Delta, x_1{:}A_1, x_2{:}A_2; I; P \circ (P_1\, P_2 \multimap\!\!\circ\, Q_1\, Q_2) \vdash E \Rightarrow x{:}A.\, Q}{\Delta; I; P \vdash (x_1{:}A_1 \Leftarrow E_1{:}P_1 \parallel x_2{:}A_2 \Leftarrow E_2{:}P_2); E \Rightarrow x{:}A.\, (\exists x_1{:}A_1, x_2{:}A_2.\, (P_1\, P_2 \multimap\!\!\circ\, Q_1\, Q_2) \circ Q)}$$

The command $\parallel$ for fork-join parallelism checks the processes E_1 and E_2 with respect to the ascribed preconditions on local state P_1 and P_2 to obtain strongest postconditions Q_1 and Q_2. A verification condition is issued to check that P_1 and P_2 indeed split the local state of the parent process into disjoint sections (the entailment $P \vdash P_1 * P_2 * \top$). Then the common continuation E is checked with respect to a new description of the state $P \circ (P_1\, P_2 \multimap\!\!\circ\, Q_1\, Q_2)$, which captures the semantics that the local heap described by P is changed so that the P_i fragment are independently updated according to Q_i. Thus, the predicate correctly captures the semantics of concurrent execution of E_1 and E_2. Since E_1 and E_2 are checked using the same shared invariant I, they can modify the shared state, but only in ways which preserve the truth value of I.

Instead of the postcondition $P_1\, P_2 \multimap\!\!\circ\, Q_1\, Q_2$, one may consider here an alternative $(P_1 \multimap Q_1) * (P_2 \multimap Q_2)$. The later would be more in line with the corresponding rule from Concurrent Separation Logic (CSL) which takes $Q_1 * Q_2$ as a postcondition for parallel composition. In the presence of binary postconditions, however, this is not a strong enough specification, and will prevent us from proving the progress and preservation theorems for our system in Section 6. Indeed, this postcondition only states that the ending heap can be split into subheaps in one particular way, but it loses the information that such a splitting is obtained from the splitting of the initial heap, or that different splitting of the initial heap will entail different splittings of the ending heap.

$$\frac{\Delta, \mathsf{m{:}heap}, P \vdash P_1 * \top \qquad \Delta; I * P_1 \vdash E_1 \Leftarrow x_1{:}A_1.\, \Box I * Q_1 \qquad \Delta, x_1{:}A_1; I; P \circ (P_1 \multimap Q_1) \vdash E \Rightarrow x{:}A.\, Q}{\Delta; I; P \vdash x_1 \Leftarrow \mathsf{atomic}_{A_1,P_1,Q_1} E_1; E \Rightarrow x{:}A.\, (\exists x_1{:}A_1.\, (P_1 \multimap Q_1) \circ Q)}$$

Semantically, atomic first acquires exclusive access to the shared state, and then executes E_1 using both the shared and the designated chunk of the local state. Thus, E_1 must be checked against a precondition $I * P_1$, where I is the descriptor of the shared state, and P_1 is the descriptor of the designated chunk of the local state. It is ensured that P_1 describes local state by emitting a verification condition $P \vdash P_1 * \top$. We emphasize that E_1 is an ST-computation, and thus it makes no semantic distinction between local and shared state. Upon exit, E_1 releases what used to be shared state, so it must make sure that its invariant is preserved. Thus, E_1 is checked against a postcondition $\Box I \mathbin{*\!\!*} Q$, which requires that E_1 changes the I portion of the its initial heap in such a way that the changed subheap satisfies I again. This portion is what will be released as an updated version of the shared state. The rest of the heap is what used to be the local state of the parent process, and is changed according to some pre-specified Q. The continuation E is then simply checked against a local heap that updates a P_1 part of P according to the binary relation Q. The update is expressed using the relation $P \circ (P_1 \multimap Q_1)$.

In order for the semantics to make sense, we must make sure that there is only one portion in the combined shared/local heap that satisfies I, else E_1 may not know how much space must be returned to shared status. That is why I is required to be precise, enforcing that it always determines a unique subheap. Precision is a standard requirement on invariants of shared resources in Separation Logic [3, 25, 31].

$$\frac{\Delta \vdash \mathsf{precise}\ J \qquad \Delta, \mathsf{m{:}heap}, P \vdash J * \top \qquad \Delta; I * J; P \circ (J \multimap \nabla \mathsf{emp}) \vdash E \Rightarrow x{:}A.\, Q}{\Delta; I; P \vdash \mathsf{publish}\ J; E \Rightarrow x{:}A.\, (J \multimap \nabla \mathsf{emp}) \circ Q}$$

Publish takes a predicate J and promotes the chunk of the local heap satisfying J into shared status. Thus, J must hold of a unique part of the local heap. Existence of such a part is ensured by the verification condition $P \vdash J * \top$, and uniqueness is ensured by the requirement that J is precise. The published state is shared throughout the scope of the continuation E, which must be checked against an extended invariant on the shared state $(I * J)$ and a description of a shrunken local state $P \circ (J \multimap \nabla \mathsf{emp})$. The later predicate simply states that the unique J part of P is replaced by an empty heap, thus subtracting J from P. In this paper we have taken a simplifying assumption that once published state cannot be reclaimed as private anymore. A more general publishing command would have a postcondition $(J \multimap \nabla \mathsf{emp}) \circ Q \circ (\mathsf{emp} \multimap \nabla J)$, thus signifying that J is returned into the private state. However, the more general rule requires a much more involved development of operational semantics, and hence we leave it for future work.

4.3 Typechecking generic computational primitives

The typing rule for conditional is unsurprising; it obtains the postconditions for the branches, and then checks the continuation with what amounts to a disjunction of these postconditions. We present only the CMD rule, as the ST rule is analogous.

$$\frac{\begin{array}{c}\Delta \vdash A \Leftarrow \mathsf{type} \\ \Delta \vdash M \Leftarrow \mathsf{bool} \qquad \Delta; I; P \circ \delta(M = \mathsf{tt}) \vdash E_1 \Rightarrow x{:}A.\, Q_1 \\ \Delta; I; P \circ \delta(M = \mathsf{ff}) \vdash E_2 \Rightarrow x{:}A.\, Q_2 \\ \Delta, x{:}A; I; P \circ (M\,?\,Q_1\,Q_2) \vdash E \Rightarrow y{:}B.\, Q\end{array}}{\begin{array}{l}\Delta; I; P \vdash x \Leftarrow \mathsf{if}_A\ M\ \mathsf{then}\ E_1\ \mathsf{else}\ E_2; E \\ \qquad \Rightarrow y{:}B.\, (\exists x{:}A.\, (M\,?\,Q_1\,Q_2) \circ Q)\end{array}}$$

The fragment of HTT described so far may easily be presented in a more customary form with Hoare triples for partial correctness, because the constructs have been essentially first-order. We now describe the two effectful constructs which are higher-order in an essential way.

The first is monadic bind, whose first-order analogue is the Hoare rule for sequential composition.

$$\frac{\begin{array}{c}\Delta \vdash K \Rightarrow \mathsf{CMD}\,\{I_1\}\{P_1\}\,x_1{:}A_1\{Q_1\} \\ \Delta, \mathsf{m{:}heap}, I \vdash I_1 * (I_1 \mathbin{-\!\!*} I) \qquad \Delta, \mathsf{m{:}heap}, P \vdash P_1 * \top \\ \Delta, x_1{:}A_1; I; P \circ (P_1 \multimap Q_1) \vdash E \Rightarrow x{:}A.\, Q\end{array}}{\Delta; I; P \vdash x_1 \Leftarrow K; E \Rightarrow x{:}A.\, (\exists x_1{:}A_1.\, (P_1 \multimap Q_1) \circ Q)}$$

The difference is that monadic bind allows the first composed computation to be obtained by *evaluating* K, whereas in Hoare Logic, the composed processes must be supplied explicitly. HTT, as well as other monadic calculi, treats computations as first-class values which can be supplied as function arguments, obtained as function results, and abstracted. These features are the essential ingredients for our lifting of Hoare Logic to higher-order.

Returning to the specifics of bind, we notice that the code encapsulated by K is executable in a local heap satisfying P (ensured by the verification condition $P \vdash P_1 * \top$), and a shared heap satisfying I (ensured by the verification condition $I \vdash I_1 * (I_1 \mathbin{-\!\!*} I)$). The later condition implies that I_1 is a *restriction* of I to the subheap determined by I_1, thus ensuring that K, by preserving the invariant I_1, also preserves the larger invariant I. Finally, according to the Hoare type of K, its execution changes the P_1 chunk of the local heap as described by the postcondition Q_1. Thus, the continuation E is appropriately checked against a precondition $P \circ (P_1 \multimap Q_1)$.

The second higher-order connective is the fixpoint. Due to the presence of higher-order functions, HTT can replace the looping constructs of Hoare logic by a general fixpoint combinator over a Hoare type. The fixpoint computation is supposed to be immediately executed, so the typing rule combines the usual typing of fixpoint combinators with monadic bind.

$$\frac{\begin{array}{c}B = \mathsf{CMD}\,\{I_1\}\{P_1\}\,x_1{:}A_1\{Q_1\} \qquad \Delta \vdash B \Leftarrow \mathsf{type} \\ \Delta, \mathsf{m{:}heap}, I \vdash I_1 * (I_1 \mathbin{-\!\!*} I) \qquad \Delta, \mathsf{m{:}heap}, P \vdash P_1 * \top \\ \Delta \vdash f \Leftarrow B \to B \qquad \Delta, x_1{:}A_1; I; P \circ (P_1 \multimap Q_1) \vdash E \Rightarrow x{:}A.\, Q\end{array}}{\Delta; I; P \vdash x_1 \Leftarrow \mathsf{fix}_B f; E \Rightarrow x{:}A.\, (\exists x_1{:}A_1.\, (P_1 \multimap Q_1) \circ Q)}$$

The ST-judgment contains similar rules for monadic bind and fixpoint. They are strictly simpler than the above, as they do not have to account for the the shared state invariant I.

Finally, we integrate the monadic judgments into the pure fragment, using the two do coercions.

$$\frac{\Delta; P \vdash E \Leftarrow x{:}A.\, Q}{\Delta \vdash \mathsf{stdo}\ E \Leftarrow \mathsf{ST}\,\{P\}\,x{:}A\{Q\}}$$

$$\frac{\Delta; I; P \vdash E \Leftarrow x{:}A.\, Q}{\Delta \vdash \mathsf{cmdo}\ E \Leftarrow \mathsf{CMD}\,\{I\}\{P\}\,x{:}A\{Q\}}$$

Because stdo and cmdo hygienically isolate the effectful computations from the pure ones, the pure fragment may be formulated in a standard way, drawing on the formalism of ECC.

$$\frac{}{\Delta, x{:}A, \Delta_1 \vdash x \Rightarrow A} \qquad \frac{\Delta, x{:}A \vdash M \Leftarrow B}{\Delta \vdash \lambda x.\, M \Leftarrow \Pi x{:}A.\, B}$$

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

$$\frac{\Delta \vdash K \Rightarrow \forall\alpha.\, A \qquad \Delta \vdash \tau \Leftarrow \mathsf{mono}}{\Delta \vdash K\ \tau \Rightarrow [\tau/\alpha]A} \qquad \frac{\Delta \vdash K \Rightarrow B \qquad A = B}{\Delta \vdash K \Leftarrow A}$$

$$\frac{\Delta \vdash M \Leftarrow A}{\Delta \vdash M{:}A \Rightarrow A}$$

In HTT, we adopt a formulation with bidirectional typechecking, inspired by the work on Concurrent LF [32], where elimination terms synthesize their types, and introduction terms must be supplied a type to check against. We can always check an elimination term K by first synthesizing its type and confirming that it is equal with the supplied type. We can always switch the direction from

```
waitThen = Λα. λl. λn. λst. λr.
  cmdo(-- I1:{(l ↦nat −) ∗ J}
       -- P1:{P ∗ r ↦nat 0}
       t ⇐ fix_A(λc. cmdo
         (-- I2:{(l ↦nat −) ∗ J}
          -- P2:{P ∗ r ↦nat 0}
          ok ⇐ atomic bool, P ∗ (r ↦nat 0),
                        t1. (t1 ? (P ∗ r ↦nat 0)
                                  (∃y:α. Q(y) ∗ r ↦α y))
                 (-- P3:{l ↦nat − ∗ J ∗ P2}
                  x ⇐ !nat l;
                  -- P4:{l ↦nat x ∗ J ∗ P2}
                  t1 ⇐ if (x = n) then
                         -- P5:{l ↦nat n ∗ J ∗ P2}
                         y ⇐ st;
                         -- P6:{l ↦nat − ∗ J ∗ Q(y) ∗ r ↦nat 0}
                         r :=α y;
                         -- P7:{l ↦nat − ∗ J ∗ Q(y) ∗ r ↦α y}
                         return ff
                         else return tt;
                  -- P8:{(t1 = ff ⊃ ∃y:α. P7) ∧ (t1 = tt ⊃ P4)}
                  return t1);
          -- I9:{(l ↦nat −) ∗ J}
          -- P9:{ok ? (P ∗ r ↦nat 0)
                      (∃y:α. Q(y) ∗ r ↦α y)}
          if ok then
            -- I10:{(l ↦nat −) ∗ J}
            -- P10:{P ∗ r ↦nat 0}
            t3 ⇐ c;
            -- I11:{(l ↦nat −) ∗ J}
            -- P11:{∃y:α. Q(y) ∗ r ↦α y}
            return t3
          else return ()));
       -- I12:{(l ↦nat −) ∗ J}
       -- P12:{∃y:α. Q(y) ∗ r ↦α y}
       return t)
```

Figure 2. Annotated example.

checking to synthesis by using the constructor $M{:}A$ to supply the type A to be synthesized.

4.4 Annotated example

The code in Figure 2 presents the typing derivation of the function waitThen from Section 2.1. We check the function against the type:

$$\forall\alpha.\, \Pi l{:}\mathsf{loc}.\, \Pi n{:}\mathsf{nat}.\, \Pi r{:}\mathsf{loc}.\\ \mathsf{ST}\{(l \mapsto_{\mathsf{nat}} n) * J * P\}x{:}\alpha\{(l \mapsto_{\mathsf{nat}} -) * J * Q(x)\} \to A$$

where $A = \mathsf{CMD}\{(l \mapsto_{\mathsf{nat}} -) * J\}\{P * (r \mapsto_{\mathsf{nat}} 0)\}t{:}1\{\exists x{:}\alpha.\, Q(x) * (r \mapsto_\alpha x)\}$. The code is annotated with predicates to illustrate the properties of the state at the various program points. We use I for predicates about the shared state, and P for predicates about local state. The typechecker will require that several verification conditions be proved, before the program is deemed correct. These are: (1) $P_2 \vdash P * (r \mapsto_{\mathsf{nat}} 0) * \top$, to enter atomic; (2) $P_3 \vdash l \hookrightarrow_{\mathsf{nat}} -$, to read from l; (3) $P_5 \vdash l \mapsto_{\mathsf{nat}} n * J * P * \top$, to execute st; (4) $P_6 \vdash r \hookrightarrow -$, to write into r; (5) $P_8 \vdash \Box I_2 \mathbin{⅋} (t_1\,?\,(P * r \mapsto_{\mathsf{nat}} 0)\ (\exists y{:}\alpha.\, Q(y) * r \mapsto_\alpha y))$, to prove that the postcondition supplied as an index to atomic is satisfied, and thus atomic exits correctly; (6) $I_{10} \vdash ((l \mapsto_{\mathsf{nat}} -) * J) * (((l \mapsto_{\mathsf{nat}} -) * J) \mathrel{-⅋} I_{10})$ and $P_{10} \vdash P * (r \mapsto_{\mathsf{nat}} 0) * \top$, to execute the recursive call to c, and (7) $P_{12} \vdash \exists x{:}\alpha.\, Q(x) * r \mapsto_\alpha x$, to prove that the postcondition of waitThen holds. All of these conditions are fairly easy to discharge.

5. Properties

5.1 Equational reasoning and logical soundness

Like every other type theory, HTT has to define a judgment $A = B$ for checking if two types A and B are equal. This judgment was used, for example, in the typing rule of the pure fragment where the bidirectional typechecker switches from synthesis to checking mode. Because types are dependent, and thus may contain terms, the judgment is non-trivial, as it has to account for term equality as well. In addition, it has to be decidable.

Thus, the equality judgment of HTT, as of other (intensional) type theories, like ECC or Coq, is quite restricted and includes only equations that lead to decidable theories. In Coq, for example, the equality judgment only admits beta reduction. Other equations of interest may, of course, be added as axioms. Properties that rely on such axioms cannot be proved automatically by the typechecker, but must be discharged by the user via explicit proofs.

The development of the monadic fragment of HTT does not depend on the organization and the equations of the pure fragment. So for example, if we chose Coq as an environment type theory of HTT, we could freely use beta reduction in the equational reasoning.

In the previous work [22], we have allowed equational reasoning that relies on beta and eta equalities for Π-types, and the generic monadic laws (unit and associativity [18]) for Hoare types. We have shown that such an equational theory is decidable, using the technically involved, but conceptually quite simple and elegant idea of *hereditary substitutions* [32]. The current paper uses literally the same pure fragment, so the same proof of decidability applies.

THEOREM 1 (Relative decidability of typechecking). *Given an oracle for deciding the verification conditions (that is, deciding the judgment $\Delta \vdash P$, where P is a proposition), all the typing judgments of HTT are decidable.*

In the actual implementation of HTT, the oracle from the above theorem can be replaced by explicit proofs, to be obtained by some form of automatic or interactive theorem proving. The later has been the approach that we adopted in the implementation of HTT in Coq, where a modicum of automation can be recovered by employing Coq tactics and tacticals. Theorem 1 can then be viewed as a guarantee that verification condition generation and typechecking are terminating processes. This would not be a particularly deep property in any first-order language, but because HTT is higher-order, deciding equality requires *normalization*. This is a non-trivial algorithm, but, by Theorem 1, it terminates.

THEOREM 2 (Soundness of the HTT logic). *The judgment $\Delta \vdash P$ cannot derive falsehood, and hence is sound.*

Theorem 2 is established by exhibiting a model of HTT. In [22], we have described a set-theoretic model which takes place in ZFC with infinite inaccessible cardinals $\kappa_0, \kappa_1, \ldots$ All the types are given their obvious set-theoretic interpretation, the universe mono of monomorphic types is the set of all sets of cardinality smaller than κ_0, heaps are interpreted as finite functions from nat to $\Sigma\alpha{:}\mathsf{mono}.\, \alpha$, and predicates on a type are interpreted as subsets of the type. Crucially, Hoare types are *interpreted as singleton sets*, and therefore all the computations in HTT are given the same logical interpretation. This is sufficient to argue logical soundness, because HTT currently contains no axioms declaring equations or other kinds of relations on effectful computations (except the monadic laws, which are very simplistic, and are handled by the typechecker). Not surprisingly, the same model suffices to argue the logical soundness of the extension from the current paper.

The above theorems concerns HTT when viewed as a logic. But HTT is at the same time a programming language, and we need to also prove that it is sound when viewed as a programming language. In particular, we need to show that if $I; P \vdash E \Leftarrow x{:}A.\, Q$, then indeed, if E is executed in a shared heap satisfying invariant I and the local heap satisfying the predicate P, and E terminates, then the ending heap satisfies the predicate Q. This

theorem would justify our typing rules and show them *adequate* with respect to the intuitive operational interpretation of E.

We will prove this *adequacy theorem* for the current extension of HTT in Section 6, after we have formally defined the operational semantics.

5.2 Framing and compositionality

HTT computations satisfy the following standard properties.

LEMMA 3. *Suppose that* $\Delta; I; P \vdash E \Leftarrow x{:}A.\, Q$. *Then:*

1. *Weakening consequent. If* $\Delta, x{:}A, i{:}\mathsf{heap}, m{:}\mathsf{heap}, Q \vdash Q'$, *then* $\Delta; I; P \vdash E \Leftarrow x{:}A.\, Q'$.
2. *Strengthening precedent. If* $\Delta, m{:}\mathsf{heap}, P' \vdash P$, *then* $\Delta; I; P' \vdash E \Leftarrow x{:}A.\, \delta P' \circ Q$.
3. *Local frame.* $\Delta; I; P * \top \vdash E \Leftarrow x{:}A.\, P \multimap Q$.
4. *Shared frame. If* J *is precise, then* $\Delta; I * J; P \vdash E \Leftarrow x{:}A.\, Q$.

Similar properties hold for ST*-computations as well.*

The proofs of these properties are somewhat involved, and the interested reader is referred to the associated technical report [21]. However, we do comment here on the Local frame property, which may be seen as somewhat unusual, compared to the other works on Separation Logic. A more recognizable form of the local frame property may be

$$\Delta; I; P * R \vdash E \Leftarrow x{:}A.\, Q \mathbin{**} \delta R$$

which directly states that E may be executed in an initial heap extended with an arbitrary subheap satisfying R, as long as the ending heap is extended with the same subheap, also satisfying R. Intuitively, this property holds since the initial typing of E prevents it from touching any disjoint state, and thus R must be preserved across the execution.

We note that the later form of the frame principle is easily derivable from Lemma 3. Indeed if $\Delta; I; P * \top \vdash E \Leftarrow x{:}A.\, P \multimap Q$, then by strengthening precedent we first get $\Delta; I; P * R \vdash E \Leftarrow x{:}A.\, \delta(P * R) \circ (P \multimap Q)$ and then because $\delta(P * R) \circ (P \multimap Q) \vdash Q \mathbin{**} \delta R$ we can weaken the consequent into $\Delta; I; P * R \vdash E \Leftarrow x{:}A.\, Q \mathbin{**} \delta R$.

We further show that HTT is compositional in the sense that typechecking of a program (which amounts to verification) requires only that the individual sub-programs are typechecked separately. There is no need for whole-program reasoning, as the types are strong enough to isolate the program components and serve as their interfaces.

As in any other type theory, HTT's compositionality theorem takes the form of a substitution principle, and we present some selected statements. Here we assume the operation $\langle E_1/x{:}A\rangle E_2$ on computations E_1 and E_2 that *prepends* E_1 onto E_2. More formally, if $E_1 = (C; \mathsf{return}\ M)$, where C is a list of commands, then $\langle E_1/x{:}A\rangle E_2$ is defined to be $C; [M{:}A/x]E_2$.

LEMMA 4 (Substitution principle). *Suppose that* $\Delta \vdash M \Leftarrow A$, *and abbreviate* $[M{:}A/x]T$ *with* T', *for arbitrary* T. *Then the following holds:*

1. *If* $\Delta, x{:}A, \Delta_1 \vdash N \Leftarrow B$ *then* $\Delta, \Delta_1' \vdash N' \Leftarrow B'$.
2. *If* $\Delta, x{:}A, \Delta_1; I; P \vdash E \Leftarrow y{:}B.\, Q$ *and* $y \notin \mathsf{FV}(M)$, *then* $\Delta, \Delta_1'; I'; P' \vdash E' \Leftarrow y{:}B'.\, Q'$.
3. *If* $\Delta; I; P \vdash E_1 \Leftarrow x{:}A.\, Q$ *and* $\Delta, x{:}A; I; P \circ Q \vdash E_2 \Leftarrow y{:}B.\, T$, *and* $x \notin \mathsf{FV}(B)$ *then* $\Delta; I; P \vdash \langle E_1/x{:}A\rangle E_2 \Leftarrow y{:}B.\, (\exists x{:}A.\, Q \circ T)$.

Notice that the last statement of the substitution principle is essentially and adaptation to binary postconditions of the Hoare-style inference rule for sequential composition. This is an additional aspect of the connection between monadic bind and sequential composition that we mentioned in Section 4. The proofs of these lemmas can be found in the associated technical report [21].

6. Operational semantics

In this section we focus on the operational semantics of the monadic fragment of HTT, and prove theorems about ST and CMD-computations. The purely functional fragment is quite standard. Since the functional fragment is a sub-language of ECC, we know that it is *strongly normalizing*. Therefore, we can give the functional fragment a number of reduction strategies, including call-by-name and call-by-value. Alternatively, we can *normalize* all of the pure subterms before applying the evaluation rules for the monadic terms. Thus, we omit the treatment of the pure fragment, and refer the interested reader to [22].

The operational semantics of monadic computations requires the following syntactic categories.

$$\begin{array}{lrcl}
\textit{Run-time heaps} & \chi & ::= & \cdot \mid \chi, l \mapsto_\tau M \\
\textit{Abstract machines} & \mu & ::= & (\chi, E) \mid (\chi, E_1 \mid x{:}A.\, E_2) \\
\textit{Stacks} & \kappa\,[P, E] & ::= & (x_1{:}A_1 \Leftarrow E_1{:}P_1 \parallel x_1{:}A_2 \Leftarrow E{:}P); E_3 \mid \\
 & & & (x_1{:}A_1 \Leftarrow E{:}P \parallel x_2{:}A_2 \Leftarrow E_2{:}P_2); E_3
\end{array}$$

Run-time heaps are finite maps from locations to terms. These are the objects about which our assertions logic reasons. The soundness of the assertion logic established in Theorem 2 makes the connection between the run-time behavior of HTT and its logical behavior. If HTT shows that at some point in the program the heap should contain a location l pointing to a value $M{:}\tau$, then, when that point in the program is reached at run-time, the heap contains an assignment $l \mapsto_\tau M$.

Abstract machines pair up a run-time heap with an expression to be evaluated. They come in two modes: (1) (χ, E) is the *concurrent mode*, which takes a CMD expression E describing the concurrent execution of a number of processes; and (2) $(\chi, E_1 \mid x{:}A.\, E_2)$ is the *atomic mode*. In the atomic mode, E_1 is an ST-computation, which must be executed before returning to the (concurrent) continuation E_2. The value of E_1 is bound to the variable $x{:}A$ in E_2.

Stacks are used to select an expression from a set of parallel expressions in E. The selected expression will be advanced one step according to the operational semantics. Given a list of stacks $\overline{\kappa_i} = (\kappa_1, \ldots, \kappa_n)$, and a list of predicates $\overline{P_i} = (P_1, \ldots, P_n)$, and an expression E, we write $\overline{\kappa_i}\,\overline{P_i}\,E$ as an abbreviation for $\kappa_1[P_1, \cdots \kappa_{n-1}[P_{n-1}, \kappa_n[P_n, E]]]$. Thus, the list $\overline{\kappa_i}$ determines the sequence of parallel nestings, at the bottom of which E appears, and the list $\overline{P_i}$ determines the sequence of footprint annotations provided along the path.

The main judgment of the operational semantics has the form $\mu \hookrightarrow \mu'$. We present selected rules for concurrent configurations.

$$\chi, \overline{\kappa_i}\,\overline{P_i}\,(x \Leftarrow \mathsf{atomic}_{A,R,Q}\, E_1; E) \hookrightarrow \chi, E_1 \mid x{:}A.\, \overline{\kappa_i}\,\overline{(P_i \circ (R \multimap Q))}\, E$$

$$\chi, \overline{\kappa_i}\,\overline{P_i}\,(\mathsf{publish}\ J; E) \hookrightarrow \chi, \overline{\kappa_i}\,\overline{(P_i \circ (J \multimap \nabla\mathsf{emp}))}\, E$$

$$\chi, \overline{\kappa_i}\,\overline{P_i}\,(x \Leftarrow (\mathsf{cmdo}\ E_1){:}\mathsf{CMD}\,\{I\}\{R_1\}\,x{:}A\{Q_1\}; E) \hookrightarrow \chi, \overline{\kappa_i}\,\overline{P_i}\,(\langle E_1/x{:}A\rangle E)$$

$$\chi, \overline{\kappa_i}\,\overline{P_i}\,((x_1{:}A_1 \Leftarrow \mathsf{return}\ M_1{:}P_1 \parallel x_2{:}A_2 \Leftarrow \mathsf{return}\ M_2{:}P_2); E) \hookrightarrow \chi, \overline{\kappa_i}\,\overline{P_i}\,([M_1{:}A_1/x_1, M_2{:}A_2/x_2]E)$$

$$\chi, \overline{\kappa_i}\,\overline{P_i}\,(x \Leftarrow \mathsf{if}_A\ \mathsf{tt}\ \mathsf{then}\ E_t\ \mathsf{else}\ E_f; E) \hookrightarrow \chi, \overline{\kappa_i}\,\overline{P_i}\,(\langle E_t/x{:}A\rangle E)$$

$$\chi, \overline{\kappa_i}\,\overline{P_i}\,(x \Leftarrow \mathsf{if}_A\ \mathsf{ff}\ \mathsf{then}\ E_t\ \mathsf{else}\ E_f; E) \hookrightarrow \chi, \overline{\kappa_i}\,\overline{P_i}\,(\langle E_f/x{:}A\rangle E)$$

$$\chi, \overline{\kappa_i}\,\overline{P_i}\,(x_1 \Leftarrow \mathsf{fix}_B\ f; E) \hookrightarrow \chi, \overline{\kappa_i}\,\overline{P_i}\,(x_1 \Leftarrow f(\mathsf{cmdo}\ (y \Leftarrow \mathsf{fix}_B\ f; \mathsf{return}\ y)); E)$$

The rules use a list of stacks $\overline{\kappa_i}$ $\overline{P_i}$ to select the first command to execute. Many different possibilities may arise corresponding to different selected stacks, reflecting the the non-deterministic nature of concurrent evaluation.

In the case of atomic, once a command E_1 is selected for atomic execution, the abstract machine moves into the atomic configuration, where E_1 proceeds to be executed without interference from other processes, and with exclusive access to the heap χ[3].

Upon making a step, both atomic and publish change the local heap, and the annotations encountered along the stack list, $\overline{\kappa_i}$, must be updated in order to reflect the new heap values. In the case of atomic, we use the predicate list $\overline{(P_i \circ (R \multimap Q))}$, because the execution of E_1 is captured by the relation $R \multimap Q$. In the case of publish, we use the predicate list $\overline{(P_i \circ (J \multimap \nabla\mathsf{emp}))}$, because the execution of publish must erase the space described by J, and this operation is captured by the relation $J \multimap \nabla\mathsf{emp}$.

The rules for the atomic configurations are straightforward, so we present the characteristic ones without any comments.

$$(\chi, l \mapsto_\tau M), (x \Leftarrow !_\tau l; E) \mid y{:}A.\, E_1 \hookrightarrow (\chi, l \mapsto_\tau M), [M{:}\tau/x]E \mid y{:}A.\, E_1$$

$$\chi, (x \Leftarrow \mathsf{alloc}_\tau M; E) \mid y{:}A.\, E_1 \hookrightarrow (\chi, l \mapsto_\tau M), [l{:}\mathsf{nat}/x]E \mid y{:}A.\, E_1$$

$$(\chi, l \mapsto -), (l :=_\tau M; E) \mid y{:}A.\, E_1 \hookrightarrow (\chi, l \mapsto_\tau M), E \mid y{:}A.\, E_1$$

$$(\chi, l \mapsto -), (\mathsf{dealloc}\ l; E) \mid y{:}A.\, E_1 \hookrightarrow \chi, E \mid y{:}A.\, E_1$$

$$\chi, (x \Leftarrow (\mathsf{stdo}\ E_2){:}\mathsf{ST}\,\{R_1\}\,x{:}B\{Q_1\}; E) \mid y{:}A.\, E_1 \hookrightarrow \chi, \langle E_2/x{:}B\rangle E \mid y{:}A.\, E_1$$

$$\chi, (\mathsf{return}\ M) \mid y{:}A.\, E_1 \hookrightarrow \chi, [M{:}A/y]E_1$$

In order to prove the adequacy of operational semantics with respect to typing, we need a helper judgment to define the typing for abstract machines. Informally, $I \vdash \mu \Leftarrow x{:}A.\, S$ holds if machine μ preserves the heap invariant I and, if μ terminates, then the ending heap satisfies the predicate ∇S. The formal definition requires a translation from run-time heaps to predicates given inductively as $[\![\cdot]\!] = \mathsf{emp}$ and $[\![\chi, l \mapsto_\tau M]\!] = [\![\chi]\!] * l \mapsto_\tau M$.

DEFINITION 5. *We say that $I \vdash \mu \Leftarrow x{:}A.\, S$ if*

1. *$\mu = \chi, E$ and $\chi = \chi_1, \chi_2$ such that $[\![\chi_1]\!] \vdash I$ and $I; [\![\chi_2]\!] \vdash E \Leftarrow x{:}A.\, \nabla S$, or*
2. *$\mu = \chi, E_1, y{:}B.\, E_2$ then there exists a predicate R such that $[\![\chi]\!] \vdash E_1 \Leftarrow y{:}B.\, \nabla(I * R)$ and $y{:}B; I; R \vdash E_2 \Leftarrow x{:}A.\, \nabla S$.*

Notice that the definition uses unary postcondition S instead of binary ones. Binary postconditions served in the typing judgments to relate the unknown initial heap to the ending heap. But, when executing abstract machines, the initial heaps are always known, explicitly given by χ, so there is no need to have any special abstractions for naming them.

Now our adequacy theorem can be presented in the manner familiar from functional programming, as a combination of preservation and progress theorems.

THEOREM 6 (Preservation). *If $I \vdash \mu \Leftarrow x{:}A.\, S$ and $\mu \hookrightarrow \mu'$, then $I \vdash \mu' \Leftarrow x{:}A.\, S$.*

THEOREM 7 (Progress). *If $I \vdash \mu \Leftarrow x{:}A.\, S$, then either $\mu = (\chi, \mathsf{return}\ M)$ or there exists μ' such that $\mu \hookrightarrow \mu'$.*

The proofs are by case analysis on the structure of the abstract machines. The progress theorem crucially depends on Theorem 2 to argue that an operational step can indeed be made if the step's precondition has been proved in the assertion logic.

[3] The optimistic evaluation usually associated with transactions need not be reflected in the operational semantics: it is an implementation strategy for speeding up the execution that does not change the semantics.

7. Related and future work

Transactional Memory. Monads for dealing with transactions have been introduced in Haskell [9, 10]. Our approach is similar, however, we have not considered an explicit abort in this paper because we are interested in a high-level semantics where an explicit abort is not necessary [19]. Also, we can state and check the pre-conditions for an atomic block statically, and do not require an explicit abort to ensure correctness of algorithms.

Higher-order and dependent types for effects. Dependently typed systems with stateful features have to date mostly focused on how to appropriately restrict effects from appearing in types, thus polluting the underlying logical reasoning. Such systems have mostly employed singleton types to establish the connection between the pure and the impure levels of the language. Examples include Dependent ML by Xi and Pfenning [33, 34], Applied type systems by Chen and Xi [4] and Zhu and Xi [35], a type system for certified binaries by Shao et al. [29], and the theory of refinements by Mandelbaum et at. [14]. HTT differs from these approaches, because we allow effectful computations to freely appear in types, as the monadic encapsulation facilitates hygienic mixing of types and effects, and thus preserves soundness. There are also several recent proposals for purely functional languages with dependent types, like Cayenne [1], Epigram [16], Omega [30] and Sage [7]. We also list several works that extend Hoare and Separation Logics with higher-order functions, like the work of Honda, Berger and Yoshida [2] and Krishnaswami et al. [12]. To our knowledge, none of the mentioned languages and logics has been extended to a concurrent setting.

Separation Logic and concurrency. Resource invariants in (sequential) Separation Logic were introduced by O'Hearn et al. [26], and an extension to concurrency with shared resources has been considered by Brookes [3], O'Hearn [25], Gotsman et al. [8] and Hobor et al. [11]. These works point out the need for precise invariants on the shared resources, in order to preserve the soundness of the logic. More recently, Vafeiadis and Parkinson [31] and Feng at el. [6] have combined Separation Logic with rely-guarantee reasoning, whereby processes specify upper and lower bounds on the evolution of the shared state.

Our treatment of shared state with invariants was inspired by O'Hearn's presentation in [25]. Using invariants simplifies the reasoning, but seems strictly weaker than rely-guarantee. Invariants only enforce a predetermined property, but otherwise lose information about the actual changes to the shared state. We have found this property somewhat restrictive in several examples, and plan in the future to reformulate HTT with ideas from the rely-guarantee approaches.

Implementation and models of concurrency. The model of HTT described here suffices to argue soundness, but is otherwise quite restrictive, as it cannot support any interesting relations on effectful computations, except the monadic laws. A more refined model of sequential, impredicative, HTT has been developed by Petersen et al. [28]. We hope that this model can be extended to a setting with transactions as well.

To improve usability of HTT, we plan to support automatic inference of (some) pre- and post-conditions and loop invariants. This would avoid the current need to explicitly annotate the concurrency primitives. Currently, HTT rules compute strongest postconditions, but a significant amount of annotations can be inferred if the rules are re-formulated to simultaneously infer the weakest precondition

that guarantees progress, as well as the strongest postcondition with respect to this precondition [23].

8. Conclusion

This paper presented Hoare Type Theory (HTT), which is a dependently typed programming language and logic supporting higher-order programs with transactional shared memory concurrency.

HTT follows the "specifications-as-types" principle, and internalizes specifications in the form of Hoare triples for partial correctness of stateful and concurrent programs into types. This isolates the concerns about side-effects and concurrency from the logical, purely functional foundations of the system, and makes it possible to mix concurrency with various advanced features, like higher-order functions, polymorphism, ADTs, none of which was possible in the previous work on Hoare or Separation Logics for concurrency. In fact, the pure fragment of HTT can soundly be scaled to the Extended Calculus of Constructions ECC [13] and Coq [15].

Hoare specifications in HTT are monads, and we support two different monadic families: $\mathsf{ST}\,\{P\}\,x{:}A\{Q\}$ classifies stateful sequential computations where P and Q are pre- and post-conditions on the state, and $\mathsf{CMD}\,\{I\}\{P\}\,x{:}A\{Q\}$ classifies transactional computations, where I is an invariant on the shared state and P and Q are pre- and post-condition on the local state. Both monads use propositions from Separation Logic to concisely describe the various aspects of the process state. Transactional computations may *atomically* invoke a stateful computation on the *shared state*, if the stateful computation provably preserves the invariant of the shared state. That is, we provide a primitive atomic, which can coerce the type $\mathsf{ST}\,\{P{*}I\}\,x{:}A\{Q*I\}$ into the type $\mathsf{CMD}\,\{I\}\{P\}\,x{:}A\{Q\}$.

We have shown that HTT as a logic is sound and compositional, so that it facilitates local reasoning. We have defined its operational semantics, and shown that this semantics is adequate with respect to the specifications from the Hoare types.

References

[1] L. Augustsson. Cayenne – a language with dependent types. In *ICFP'98*, pages 239–250.

[2] M. Berger, K. Honda, and N. Yoshida. A logical analysis of aliasing in imperative higher-order functions. In *ICFP'05*, pages 280–293.

[3] S. Brookes. A semantics for concurrent separation logic. In *CONCUR'04*, pages 16–34.

[4] C. Chen and H. Xi. Combining programming with theorem proving. In *ICFP'05*, pages 66–77.

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

[6] X. Feng, R. Ferreira, and Z. Shao. On the relationship between concurrent separation logic and assume-guarantee reasoning. In *ESOP'07*, pages 173–188.

[7] C. Flanagan. Hybrid type checking. In *POPL'06*, pages 245–256.

[8] A. Gotsman, J. Berdine, B. Cook, N. Rinetzky, and M. Sagiv. Local reasoning for storable locks and threads. In *APLAS'07*, pages 19–38.

[9] T. Harris and K. Fraser. Language support for lightweight transactions. *ACM SIGPLAN Notices*, 38(11):388–402, Nov. 2003.

[10] T. Harris, S. Marlow, S. L. P. Jones, and M. Herlihy. Composable memory transactions. In *PPoPP'05*, pages 48–60.

[11] A. Hobor, A. W. Appel, and F. Z. Nardelli. Oracle Semantics for Concurrent Separation Logic. In *ESOP'08*, pages 353–367.

[12] N. Krishnaswami. Separation logic for a higher-order typed language. In *SPACE'06*, pages 73–82.

[13] Z. Luo. *An Extended Calculus of Constructions*. PhD thesis, University of Edinburgh, 1990.

[14] Y. Mandelbaum, D. Walker, and R. Harper. An effective theory of type refinements. In *ICFP'03*, pages 213–226.

[15] The Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2004. Version 8.0.

[16] C. McBride and J. McKinna. The view from the left. *Journal of Functional Programming*, 14(1):69–111, January 2005.

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

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

[19] K. F. Moore and D. Grossman. High level small step operational semantics for transactions. In *Workshop on Transactional Computing*, August 2007.

[20] A. Nanevski, A. Ahmed, G. Morrisett, and L. Birkedal. Abstract Predicates and Mutable ADTs in Hoare Type Theory. In *ESOP'07*, pages 189–204.

[21] A. Nanevski, P. Govereau, and G. Morriset. Type-theoretic semantics for transactional concurrency. Technical Report TR-09-07, Harvard University, July 2007.

[22] A. Nanevski, G. Morrisett, and L. Birkedal. Hoare Type Theory, Polymorphism and Separation. *Journal of Functional Programming*, 18(5&6):865–911, 2008.

[23] A. Nanevski, G. Morrisett, A. Shinnar, P. Govereau, and L. Birkedal Ynot: dependent types for imperative programs In *ICFP'08*, pages 229–240.

[24] P. O'Hearn, J. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *CSL'01*, pages 1–19.

[25] P. W. O'Hearn. Resources, concurrency and local reasoning. *Theoretical Computer Science*, 375(1–3):271–307, May 2007.

[26] P. W. O'Hearn, H. Yang, and J. C. Reynolds. Separation and information hiding. In *POPL'04*, pages 268–280.

[27] S. Owicki and D. Gries. Verifying properties of parallel programs: an axiomatic approach. *Communications of the ACM*, 19(5):279–285, 1976.

[28] R. L. Petersen, L. Birkedal, A. Nanevski, and G. Morrisett. A realizability model for impredicative Hoare Type Theory. In *ESOP'08*, pages 337–352.

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

[30] T. Sheard. Languages of the future. In *OOPSLA'04*, pages 116–119.

[31] V. Vafeiadis and M. Parkinson. A marriage of rely/guarantee and separation logic. In *CONCUR'07*, pages 256–271.

[32] K. Watkins, I. Cervesato, F. Pfenning, and D. Walker. A concurrent logical framework: The propositional fragment. In *Types for Proofs and Programs*, volume 3085 of *Lecture Notes in Computer Science*, pages 355–377.

[33] H. Xi and F. Pfenning. Eliminating array bound checking through dependent types. In *PLDI'98*, pages 249–257.

[34] H. Xi and F. Pfenning. Dependent types in practical programming. In *POPL'99*, pages 214–227.

[35] D. Zhu and H. Xi. Safe programming with pointers through stateful views. In *PADL'05*, pages 83–97.