A Compositional Theory for STM Haskell

Johannes Borgström   Karthikeyan Bhargavan   Andrew D. Gordon

Microsoft Research, Cambridge, UK

{joborg,karthb,adg}@microsoft.com

Abstract

We address the problem of reasoning about Haskell programs that use Software Transactional Memory (STM). As a motivating example, we consider Haskell code for a concurrent non-deterministic tree rewriting algorithm implementing the operational semantics of the ambient calculus. The core of our theory is a uniform model, in the spirit of process calculi, of the run-time state of multi-threaded STM Haskell programs. The model was designed to simplify both local and compositional reasoning about STM programs. A single reduction relation captures both pure functional computations and also effectful computations in the STM and I/O monads. We state and prove liveness, soundness, completeness, safety, and termination properties relating source processes and their Haskell implementation. Our proof exploits various ideas from concurrency theory, such as the bisimulation technique, but in the setting of a widely used programming language rather than an abstract process calculus. Additionally, we develop an equational theory for reasoning about STM Haskell programs, and establish for the first time equations conjectured by the designers of STM Haskell. We conclude that using a pure functional language extended with STM facilitates reasoning about concurrent implementation code.

Categories and Subject Descriptors D.2.4 [*Software/Program Verification*]: Correctness Proofs; D.3.1 [*Formal Definitions and Theory*]: Syntax and Semantics; D.3.3 [*Language Constructs and Features*]: Concurrent Programming Structures—Software Transactional Memory.

General Terms Theory, verification

Keywords Transactional memory, compositional reasoning, ambient calculus

1. Introduction

Software Transactional Memory (STM), introduced by Shavit and Touitou [31], is a promising programming abstraction for shared-variable concurrency. Shared variables may only be accessed within transactions. Ingenious software techniques allow transactions to run in parallel while their semantics is as if they run in series. There is a good deal of promising research on efficient implementations, in the context of various languages [9, 10, 27]. Moreover, several formal techniques have been applied to verifying the underlying algorithms [30] and their implementations [22, 8, 20, 1].

Haskell'09, August 30, 2009, Edinburgh, Scotland, UK.
Copyright © 2009 ACM 978-1-60558-508-6/09/09...$5.00

In this paper, we explore the prospects for reasoning about software written using the STM abstraction. Transactional semantics undoubtedly simplifies the reasoning task compared to say lock-based concurrency [18, 23], but is no panacea.

We pursue the idea that theories of concurrency developed in abstract process calculi can fruitfully be recast in the concrete setting of transactional programming languages. We consider programs written in STM Haskell [10, 11], an embedding of transactional concurrency within the pure functional language Haskell.

As a concrete programming task, we investigate programming, specifying, and reasoning about an STM Haskell implementation of ambients. The ambient calculus [5] is a formalism for expressing and reasoning about mobile computation, more recently applied to biological systems [26]. An ambient process has a hierarchical structure that mutates over time, as ambients within the structure move inside and outside other ambients. Hence, an implementation of the ambient calculus amounts to a concurrent tree rewriting algorithm. The first concurrent implementation, by Cardelli in Java [4], was lock-based; here, we give a lock-free implementation of a programming API for the ambient calculus in STM Haskell.

As a basis for reasoning about this code, we present a core calculus for STM Haskell: a concurrent non-strict lambda calculus with transactional variables and atomic blocks. The syntax of lambda calculus expressions provides a compositional and uniform formalism for both source programs and their run-time states.

The original presentation of STM Haskell separated the transactional heap, its latest checkpoint and the currently running code. In contrast, our syntax uniformly represents all of these as expressions, hence facilitating compositional reasoning by allowing multiple threads with associated pieces of the heap to be composed using a parallel operator and rearranged using structural congruence.

We develop a formal semantics and type system for the calculus. Our semantics is based on a reduction relation $\rightarrow$ on expressions, which range over both effect-free functional computations and effectful concurrent computations. It facilitates local reasoning by having transactions run against a part of the heap, making it easy to exhibit the smallest heap allowing a given transaction to make progress. On the other hand, we can show a precise correspondence with the original (but syntactically more complex) operational semantics of STM Haskell.

We import into STM Haskell behavioural equivalences and proof techniques that originate in process calculi. Notably, in our proofs, the notion of bisimulation up to a relation [19, 28] permits significant reductions in the size of the state space that needs to be considered when reasoning equationally about a program. This state space reduction comes beyond the already significant reduction afforded by the atomicity and serializability guarantees of the STM abstraction.

Using these proof techniques, our first result, Theorem 1, directly relates the operational semantics of ambient processes with expressions representing the run time states of their Haskell implementation. More specifically, Theorem 1 establishes liveness, soundness, completeness, safety, and termination properties. Building on the first theorem, our second result, Theorem 2, is a concise statement of full correctness of our implementation in terms of a bisimulation between the source process and its Haskell translation. The use of transactions makes the Haskell code rather simpler than Cardelli's original Java program; still, it is non-trivial but feasible to establish the intended correspondences between the implementation code and the formal specification of ambients.

Finally, we adopt the standard definition of Morris-style contextual equivalence of expressions, and develop a sound equational theory. We show the monad laws of the STM monad hold, and also establish some equations proposed by the designers of STM Haskell [10].

Our main contributions are the following:

- We use notions of behavioural equivalence and proof techniques from process calculi to specify and prove STM Haskell code for a complex concurrent tree rewriting algorithm (an implementation of the ambient calculus).
- We develop uniform syntax and reduction semantics, in the style of process calculi, for STM Haskell. The uniformity of the syntax facilitates compositional reasoning over multiple threads with associated pieces of the heap.
- We prove soundness of an equational theory for STM Haskell, including monad laws and properties of operators for transactional control flow.

Outline of the Paper Section 2 introduces the formal syntax of core STM Haskell, and reviews the programming model. (Since our motivating example does not use exceptions, we omit them from the core calculus. The extended version of the paper shows how to extend our core calculus with exceptions.) Section 3 describes our code and data structures for programming ambients in Haskell. Section 4 completes our formalization of STM Haskell; we define the operational semantics and type system, and compare with the original semantics. Section 5 recalls the semantics of the ambient calculus, and states and proves the expected correctness properties; the proof depends on a detailed correspondence between ambient processes and expressions representing the corresponding states of our implementations. Section 6 develops an equational theory based on our operational semantics. Section 7 describes related work, and Section 8 concludes, and discusses potential future work.

An extended version of this paper, with additional explanations and proofs, and code listings, is available [3].

2. A Core Calculus for STM Haskell

The GHC implementation of STM [10] uses a monad to encapsulate all accesses to shared transactional variables (TVars). To execute an STM transaction we use the function `atomically` , which has type $\texttt{STM}\ t \rightarrow \texttt{IO}\ t$. If the execution of a transaction `returns` a result, the run-time system guarantees that it executed independently of all other transactions in the system. An STM expression can also `retry`, which may cause the transaction to be rolled back and run at some later time.

The original definition of STM Haskell made use of an implicit functional core language. For verification purposes, and in order to make this paper self-contained, we formalize STM Haskell as a concurrent non-strict lambda calculus, with memory cells (TVars) and atomic blocks.

2.1 Syntax

Our syntax treats source programs, heaps made up of transactional variables (TVars), and concurrent threads uniformly as expressions in the calculus, similarly to Concurrent Haskell [24].

We assume denumerable distinct sets $\mathbf{X}$ of (lambda calculus) variables and $\mathbf{N}$ of (TVar) addresses; we let x, y, z range over $\mathbf{X}$, and a, b range over $\mathbf{N}$. We let f range over ADT constructors.

Expressions of Core STM Haskell:

$M, N \in \mathbf{M} ::=$		expression
x	$(x \in \mathbf{X})$	variable (value)
a	$(a \in \mathbf{N})$	address (value)
$\lambda x.M$		lambda abstraction (value)
$f\ \overline{M}$		construction (value)
$M\ N$		application
$\texttt{case}\ M\ \texttt{of}\ \overline{f\ \overline{x} \rightarrow N}$		case expression
$\texttt{Y}\ M$		fixpoint
$\texttt{equal}\ M\ N$		address equality
$\texttt{readTVar}\ M$		STM read variable
$\texttt{writeTVar}\ M\ N$		STM write variable
$\texttt{return}_{\text{STM}}\ M$		STM return
$\texttt{retry}$		STM retry transaction
$M \gg\!=_{\text{STM}} N$		STM bind
$\texttt{orElse}\ M\ N$		STM prioritized choice
$\texttt{or}\ M\ N$		STM erratic choice
$\texttt{atomically}\ M$		IO execute transaction
$\texttt{return}_{\text{IO}}\ M$		IO return
$M \gg\!=_{\text{IO}} N$		IO bind
$a \mapsto M$		transactional variable (TVar)
$(\nu a)M$		restriction (a bound in M)
$M \mid N$		parallel composition
$\texttt{emp}$		empty heap

Our syntax is close to STM Haskell, but with minor differences. The STM Haskell functions `newTVar` and `fork` are not primitive expressions in our calculus, but are derived forms, explained below. In actual source programs, the monadic bind and return operators do not have the subscripts IO and STM; instead, the monads are inferred by the typechecker. The syntax $a \mapsto M$, $(\nu a)M$, $M \mid N$, `emp`, and addresses a exist as expressions only to represent run-time state, and cannot be written directly in source programs. Actual Haskell includes various abbrevations such as recursive definitions, pattern matching, and do-notation for monads, which can be reduced in standard ways to our core syntax.

We introduce some notational conventions. We write $\overline{M}$ as shorthand for a possibly empty sequence $M_1 \cdots M_n$ (and similarly for $\overline{x}$, $\overline{t}$, etc.) and $\overline{f\ \overline{x} \rightarrow N}$ for a non-empty sequence $f_1\ \overline{x_1} \rightarrow N_1 \mid \cdots \mid f_m\ \overline{x_m} \rightarrow N_m$ where the f_i as well as the x_{ij} are assumed to be pairwise different (and similarly for $\overline{f\ \overline{t}}$). We write the empty sequence as $\circ$ and denote concatenation of sequences using a comma. The length of a sequence $\overline{x}$ is written $|\overline{x}|$. If ϕ is a phrase of syntax (such as an expression), we let $\text{fv}(\phi)$ and $\text{fn}(\phi)$ be the sets of variables and names occuring free in ϕ. We write $\phi\{^{M}/_{x}\}$ for the outcome of the capture-avoiding substitution of M for each free occurrence of x in ϕ.

We let $g\ \overline{M}$ range over the *builtin applications*, which are the expressions listed above starting with $\texttt{Y}\ M$ and ending with $M \gg\!=_{\text{IO}} N$. (This notation is particularly useful in the typing rule (T BUILTIN) in Section 4.2.) If $g\ \overline{M}$ is a builtin application, we say g is a builtin function; $\texttt{Y}$ and $\gg\!=_{\text{IO}}$ are examples. To simplify our reduction semantics, builtin functions are not themselves expressions, and only occur fully applied to their arguments. As usual, we can use lambda abstraction to represent unapplied or partially applied builtin functions as expressions.

2.2 Informal Semantics

Our uniform syntax for expressions covers single-threaded functional computations, as well as heaps, imperative transactions, and concurrent computations. We describe each of these in turn, together with their semantics.

Functional Computations The core of our expression language is a call-by-name lambda calculus with algebraic data types. The simplest expressions are *values*, which return at once. As well as variables x, there are three kinds of value: names a representing the address of a mutable variable in the heap, lambda abstractions $\lambda x.M$, and constructions $f\ \overline{M}$ representing data tagged with the algebraic type constructor f. For example, `True`, `False`, `Nil`, and `Cons` $M_1\ M_2$ are constructions. Since we describe a non-strict language, constructor arguments $\overline{M}$ may be general expressions and not just values.

The other expressions of the lambda calculus core are as follows. An application expression $M\ N$ evaluates to $M'\left\{{}^{N}\!/_{x}\right\}$, if evaluation of M returns a function $\lambda x.M'$. A case expression `case` M `of` $\overline{f\ \overline{x} \to N}$ attempts to match one of the clauses $\overline{f\ \overline{x} \to N}$ against M. If $f_j\ \overline{x_j} \to N_j$ is the first of these clauses such that the value of M is $f_j\ \overline{M}$ for some $\overline{M}$, the whole expression behaves as $N_j\{{}^{\overline{M}}\!/_{\overline{x}_j}\}$; if there is no such clause, the whole expression is stuck. A builtin application `Y` M evaluates to M (`Y` M). (We use `Y` to explain recursive definitions in the standard way.) A builtin application `equal` $M\ \ N$ evaluates both M and N, and returns True if they evaluate to the same address, and False otherwise.

Heaps Next, we describe *heap-expressions*, which consist of a possibly empty composition of transaction variables, or TVars, for short. An expression $a \mapsto M$ denotes a heap consisting of a single TVar, with address a and current content M. The expression `emp` represents the empty heap, and the composition $M \mid N$ represents the concatenation of heaps M and N.

More generally, parallel composition $M \mid N$ represents M and N running in parallel, where M and N may be a mixture of heaps, STM-expressions, and IO-expressions. Unusually, the $\mid$ operator is not fully commutative; the result of $M \mid N$, if any, is the result of N. To evaluate a restriction $(\nu a)M$ one creates a fresh address a and then evaluates M; as in process calculi, a restriction may also be read declaratively as meaning that the address a is known only within its scope M.

Imperative Transactions An *STM-expression* typically has the form $H \mid M$, where H is a heap-expression and M is a running (single-threaded) *transaction*. Transactions are composed from reads (`readTVar` a) and writes (`writeTVar` $a\ M$) to TVars. In source programs, we create TVars by the following abbreviation:

$$\texttt{newTVar}\ M := (\nu a)(a \mapsto M \mid \texttt{return}_{\text{STM}}\ a) \text{ where } a \notin \text{fn}(M)$$

A transaction may $\texttt{return}_{\text{STM}}$ a result M (to commit), or `retry` (to roll back any updates and restart). There are two different kinds of choice. Prioritized choice M `orElse` N behaves as M unless M does a `retry` (in which case it behaves as N). Erratic choice `or` $M\ N$ behaves as M or as N nondeterministically.

Concurrency Finally, an *IO-expression* typically has the form $H \mid M_1 \mid \ldots \mid M_n$, where H is a heap-expression and $M_1, \ldots, M_n$ are *threads* running in parallel. A thread M_i may run a transaction to have side-effects on the shared heap H. Each thread may eventually terminate by returning a result. In our source programs, we create parallel threads using the abbreviation:

$$\texttt{fork}\ M := (M \mid \texttt{return}_{\text{IO}}\ ())$$

A thread `atomically` M behaves as if the transaction M executes in isolation against the heap, in one big atomic step. There are two possibilities to consider. If M returns a result M', the updates to the heap are committed, and the whole expression returns M'. If M retries, any updates to the heap are rolled back, and the whole expression snaps back to `atomically` M, ready to try again.

Both IO and STM computations can be sequenced using the bind operator $M \gg\!=_{\text{IO}} N$, which behaves as $N\ M'$ if M returns M', and otherwise as M.

For an example, consider a function `swap` that swaps the values of two TVars:

$$\begin{aligned}\texttt{swap} := \lambda x_a.\lambda x_b.\texttt{readTVar}\ x_a \gg\!=_{\text{STM}} \lambda y.\texttt{readTVar}\ x_b \gg\!=_{\text{STM}} \\ \lambda z.\texttt{writeTVar}\ x_a\ z \gg\!=_{\text{STM}} \lambda w.\texttt{writeTVar}\ x_b\ y\end{aligned}$$

Here, `swap` takes two TVar arguments x_a and x_b and calls the builtin function `readTVar` to read their values. As discussed (infix) function $\gg\!=_{\text{STM}}$ binds the return value of the expression to its left to the function on its right. Hence, y and z get the values of x_a and x_b, respectively; `swap` then calls the function `writeTVar` to write x_a and x_b with z and y, respectively, thus completing the swap. To call the function with two TVars a, b, we can write `swap` $a\ b$. However, two parallel executions of swap could yield an inconsistent state. Instead, we call the function and have it execute atomically, by writing `atomically` (`swap` $a\ b$).

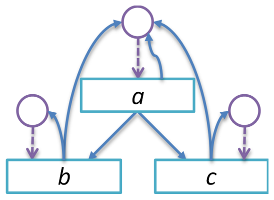
3. Application: An API for Managing Ambients

As a verification challenge, we consider a sizeable program written in STM Haskell. Our target application is an API that implements the mobility primitives of the ambient calculus [5]. Programs that use this API spawn multiple threads that concurrently modify a shared data structure. We consider such programs to be exemplary applications of STM Haskell.

Our implementation is inspired by Cardelli's earlier implementation in Java using locks [4], which, although carefully specified, was never proved correct. A more recent programming comparison between a lock-based and STM-based implementation of ambients finds the "STM implementation to be much easier to reason about and much faster to implement" [32]. We seek to evaluate whether an STM implementation is also easy to verify, by developing the first proof for an implementation of the ambient calculus.

A Tree of TVars. The underlying data structure is a tree of named ambients, implemented as follows. The figure on the right depicts an example tree with a node named `a` that has two children named `b` and `c`.

```
type Ambient = TVar AmbData
data AmbData =
    AD (Name,
        Maybe Handle,
       [Handle],
       [(Name, Ambient)])
type Name = TVar String
type Handle = TVar Ambient
```



An `Ambient` (depicted as a named rectangle in the figure) is a TVar containing an `AmbData`, which consists of four items: a `Name`, a `Handle` pointing to its parent node (if it has one), a list of all incoming handles pointing to the current node, and an assocation list of child nodes, mapping node names to `Ambients`. A `Name` is an identifier (depicted as an italicized variable), implemented as a transactional variable (TVar) containing a string; the same `Name` may appear on multiple nodes. A `Handle` (depicted as a circle) is a pointer to an `Ambient`; in the figure, such pointers are depicted as dashed arrows. The additional level of indirection using `Handles` is used when merging a child node with its parent; all handles pointing to the child can then simply be pointed over to its parent.

We illustrate this data structure through a few simple functions that are used in our subsequent development. These functions can only be called from within STM atomic blocks.

```
readAmb :: Handle -> STM AmbData
readAmb h = do { a <- readTVar h;
                 ad <- readTVar a;
                 return ad}
```

The function `readAmb` takes a `Handle` `h` and returns the `AmbData` it points to (through two levels of indirection). Since `readAmb` reads transactional variables using `readTVar`, the result is an STM action.

```
writeAmb :: Handle -> AmbData -> STM ()
writeAmb h ad = do { a <- readTVar h;
                     writeTVar a ad }
```

The function `writeAmb` both reads and writes transactional variables; it writes an `AmbData` to the location pointed to by a handle.

```
parentOf :: Handle -> STM Handle
parentOf a = do { AD (_,p,_,_) <- readAmb a;
                  case p of
                    Nothing -> retry;
                    Just ph -> return ph}
```

The function `parentOf` takes a `Handle` `h` pointing to a node and returns a handle to the parent of the node; if there is no parent (that is, `h` points to the root node), then it calls the STM `retry` function to rollback the transaction and restart.

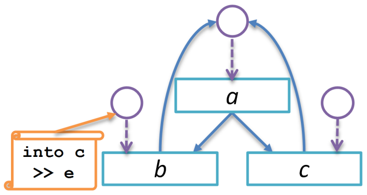
Mobile Agents. An agent represents an IO thread that is initialized with a handle to an ambient. We say that the agent executes within that ambient.

```
data AGENT a = Agent (Handle -> IO a)
type Agent = AGENT ()
nil   :: Agent
root  :: Agent -> IO ()
new   :: String -> AGENT Name
amb   :: Name -> Agent -> Agent
```

The simplest agent is `nil`, which denotes an inactive agent that has finished its computation. The function `root` creates a fresh (unnamed) root node and attaches an agent `A` to this node. The agent `A` can then create subtrees by calling `new` to generate fresh node names, and calling `amb` to create new child nodes.

Using these functions, we can now create our example tree and attach agents to each node:

```
ex = root $ do {
      a <- new "a";
      b <- new "b";
      c <- new "c";
      amb a (do
       amb b (into c);
       amb c nil)}
```

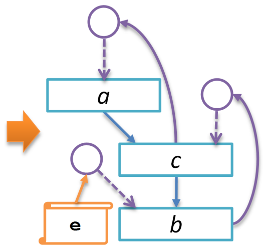


Here, the agent attached to `b` uses a new function `into`:

```
into  :: Name  -> Agent
```

The figure treats the more general case when the agent performs other actions `e` after calling `into`. However, for simplicity, it does not depict `nil` processes or back pointers to handles.

When the agent `into c` is executed, it has the effect of moving the (subtree rooted at) node `b` to become a child of node `c`. The resulting tree is depicted on the right. If there is no neighbouring tree labelled `c`, the operation blocks until one becomes available.



As usual, when concurrent threads modify the tree in this way, there is a risk of the tree ending up in an inconsistent state. Our implementation of `into` in STM Haskell below uses the STM construct `atomically` to avoid inconsistency:

```
into c = Agent $ \bHandle -> atomically $ do {
  bAmbient         <- readTVar bHandle;
  AD (b,bp,bh,bc) <- readAmb bHandle;
  aHandle          <- parentOf bHandle;
  AD (a,ap,ah,ac) <- readAmb aHandle;
  let bSiblings = delete (b,bAmbient) ac in do
  cAmbient         <- lookup' c bSiblings;
  AD (_,cp,ch,cc) <- readTVar cAmbient;
  let cHandle = head ch in do
  writeAmb cHandle (AD (c, cp, ch,
                         (b,bAmbient):cc));
  writeAmb aHandle (AD (a, ap, ah, bSiblings));
  writeAmb bHandle (AD (b,Just cHandle,bh,bc))}
```

The function `into` takes as argument the name `c` of the target ambient and creates an agent parameterized by a handle `bHandle` to the source ambient named `b`. The function proceeds in two phases. First, it reads the values of three ambient nodes: `b`, `b`'s parent `a`, and some sibling of `b` named `c`. Then, it writes updated values to all three nodes.

The function begins by reading the ambient at `b`, `bAmbient`, and calls `readAmb` to read its contents. It calls `parentOf` to find the parent ambient at `a` and reads its contents, including the list of its children `ac`. It computes the siblings of `b` by deleting `(b,bAmbient)` from the association list `ac`. It then finds a target ambient `cAmbient` by calling `lookup'` which non-deterministically chooses some sibling ambient with name `c`. (This non-determinism is motivated by the desired correctness properties of Section 5.) It reads the contents of `cAmbient`, including its children `cc` and one of its handles, `cHandle`.

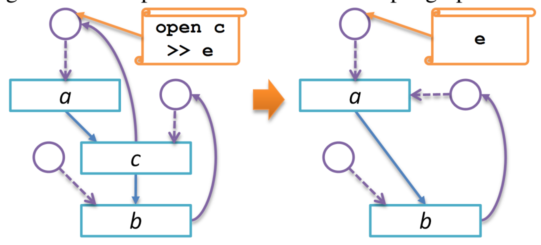
Finally, the function updates the ambient at `c` by adding a child `(b,bAmbient)` to `ch`, it updates the ambient at `a` by deleting the child `b` from `ac`, and it updates the ambient at `b` by changing its parent to `c`.

Note that `into` is a local operation that only modifies three nodes of the tree; Agents manipulating other parts of the tree can be scheduled to run in parallel without causing any conflicts.

The Full API. The full Ambient API consists of several other functions:

```
out   :: Name   -> Agent
open  :: Name   -> Agent
fork  :: Agent -> Agent
```

The agent `out c` is the inverse of `into c`; it moves an ambient out of its parent (if the parent is named `c`). The agent `open c` deletes a child node named `c` and swings all handles of `c` to point to its parent. This has the effect of causing all of `c`'s children to become children of the parent; all agents running on `c` are similarly affected. The figure below depicts its effect on an example graph.



The agent `fork A` forks off a new thread running the agent `A` within the same ambient.

Programmatically, agents form a Reader monad, where the value read is a handle to the location in which the agent is running[1]. The $\gg=$ and $\gg$ operators have their standard definition: $(\texttt{Reader}\ f) \gg= g$ running at a location with handle h evaluates $f\ h$ to some v, evaluates $g\ v$ to some $\texttt{Reader}\ g'$, and then evaluates $g'\ h$. Similarly, $(\texttt{Reader}\ f) \gg (\texttt{Reader}\ g)$ reading handle h evaluates $f\ h$, discards the result, and then evaluates $g\ h$.

```
instance Monad AGENT where
 return a = Agent $ \s -> return a
 a >>= g = Agent $ \s -> case a of Agent f -> f s
       >>= \v -> case (g v) of Agent ff -> (ff s)
 a >> b = Agent $ \s -> case a of Agent f -> f s
       >>= \v -> case b of Agent ff -> (ff s)
```

When verifying the ambient API, we are interested in establishing full functional correctness, not only the preservation of certain invariants of the location tree. To do this, we need to give a formal account of the semantics of our core calculus for STM Haskell.

4. The Core Calculus, Concluded

This section concludes the definition of our core calculus, begun in Section 2. We define the operational semantics and type system, and make a comparison with the original semantics. In the next section, we apply the calculus to specifying and verifying the Haskell code from Section 3.

4.1 Operational Semantics

We define a *reduction relation*, $M \rightarrow M'$, which specifies the run time behaviour of STM programs. A single reduction relation captures pure functional computation, imperative transactions, and concurrency. We rely on some auxiliary notions to define reduction. First, we define three kinds of evaluation contexts.

Contexts: Pure ($\mathscr{R}_\beta$), parallel ($\mathscr{R}_|$), and transactional ($\mathscr{R}_\mapsto$)

$$\begin{aligned}
\mathscr{R}_\beta &::= [\cdot] \mid \mathscr{R}_\beta\ M \mid \texttt{case}\ \mathscr{R}_\beta\ \texttt{of}\ \overline{f\ \overline{x} \rightarrow\ N} \mid \texttt{equal}\ \mathscr{R}_\beta\ M \\
&\quad \mid \texttt{equal}\ a\ \mathscr{R}_\beta \mid \texttt{readTVar}\ \mathscr{R}_\beta \mid \texttt{writeTVar}\ \mathscr{R}_\beta\ M \\
\mathscr{R}_| &::= [\cdot] \mid (\nu a)\mathscr{R}_| \mid (\mathscr{R}_| \mid M) \mid (M \mid \mathscr{R}_|) \mid \\
&\quad (\mathscr{R}_| \gg=_{\text{IO}} M) \mid (\mathscr{R}_| \gg=_{\text{STM}} M) \\
\mathscr{R}_\mapsto &::= [\cdot] \mid (\nu a)\mathscr{R}_\mapsto \mid (a \mapsto M \mid \mathscr{R}_\mapsto)
\end{aligned}$$

The second auxiliary notion is *structural equivalence*, $M \equiv M'$. The purpose of this relation is to re-arrange the structure of an expression—for example, by pulling restrictions to the top, or by moving TVars beside reads or writes—so as to enable reduction steps. Structural equivalence is the least equivalence relation closed under the following rules. Let $\text{bn}(\mathscr{R}_|)$ be the names bound by the context $\mathscr{R}_|$, and let $\text{n}(\mathscr{R}_|) = \text{bn}(\mathscr{R}_|) \cup \text{fn}(\mathscr{R}_|)$.

Structural Equivalence: $M \equiv N$

$M \equiv \texttt{emp} \mid M$	(STRUCT EMP)
$M \mid \mathscr{R}_|[N] \equiv \mathscr{R}_|[M \mid N]$ if $\text{bn}(\mathscr{R}_|) \cap \text{fn}(M) = \varnothing$	(STRUCT FLOAT)
$\mathscr{R}_|[(\nu a)M] \equiv (\nu a)\mathscr{R}_|[M]$ if $a \notin \text{n}(\mathscr{R}_|)$	(STRUCT RES CTX)
$M \equiv N \Rightarrow \mathscr{R}_|[M] \equiv \mathscr{R}_|[N]$	(STRUCT CTX)

Let *reduction*, $M \rightarrow M'$, be the least relation closed under the rules in groups (R1), (R2), and (R3) displayed below. The first group consists of standard rules for functional and concurrent computation.

[1] The Haskell programmer familiar with monad transformers will notice that it is effectively a `ReaderT Handle IO a`.

(R1) Reductions without Side-Effects: $M \rightarrow M'$

$(\lambda x.M)\ N \rightarrow M\{^N/_x\}$	(BETA)
$\texttt{case}\ f_j(\overline{M})\ \texttt{of}\ \overline{f\ \overline{x} \rightarrow\ N} \rightarrow N_j\{^{\overline{M}}/_{\overline{x}_j}\}$	(CASE)
$\texttt{Y}\ M \rightarrow M\ (\texttt{Y}\ M)$	(FIX)
$\texttt{equal}\ a\ a \rightarrow \texttt{True}$	(EQUAL TRUE)
$\texttt{equal}\ a\ b \rightarrow \texttt{False}$ if $a \neq b$	(EQUAL FALSE)
$(\texttt{return}_{\text{IO}}\ M \gg=_{\text{IO}} N) \rightarrow N\ M$	(IO BIND RETURN)

(PURE CTX)
$$\frac{M \rightarrow M'}{\mathscr{R}_\beta[M] \rightarrow \mathscr{R}_\beta[M']}$$

(RED CTX)
$$\frac{M \rightarrow M'}{\mathscr{R}_|[M] \rightarrow \mathscr{R}_|[M']}$$

(STRUCT)
$$\frac{M \equiv N \quad N \rightarrow N' \quad N' \equiv M'}{M \rightarrow M'}$$

The second group of reduction rules concerns the core behaviour of STM-expressions. A heap-expression H is a parallel composition of transactional variables

$$\Pi_i(a_i \mapsto M_i) := a_1 \mapsto M_1 \mid \cdots \mid a_n \mapsto M_n \mid \texttt{emp}$$

where the a_i are pair-wise distinct. We write $\rightarrow^*$ for the transitive closure of $\rightarrow$.

(R2) Core Reductions for STM Transactions: $M \rightarrow M'$

(STM READ TVAR)
$(a \mapsto M) \mid \texttt{readTVar}\ a \rightarrow (a \mapsto M) \mid \texttt{return}_{\text{STM}}\ M$

(STM WRITE TVAR)
$(a \mapsto M) \mid \texttt{writeTVar}\ a\ M' \rightarrow (a \mapsto M') \mid \texttt{return}_{\text{STM}}\ ()$

$(\texttt{return}_{\text{STM}}\ M \gg=_{\text{STM}} N) \rightarrow N\ M$	(STM BIND RETURN)
$(\texttt{retry} \gg=_{\text{STM}} N) \rightarrow \texttt{retry}$	(STM BIND RETRY)

(ATOMIC RETURN)
$$\frac{H \mid M \rightarrow^* \mathscr{R}_\mapsto[\texttt{return}_{\text{STM}}\ N]}{H \mid \texttt{atomically}\ M \rightarrow \mathscr{R}_\mapsto[\texttt{return}_{\text{IO}}\ N]}$$

(STM READ TVAR) and (STM WRITE TVAR) allow transactional variable to be read and written within a transaction.

(STM BIND RETURN) allows return values to propagate through the STM bind operator, much as through the IO bind operator, while (STM BIND RETRY) allows retry to propagate directly through the bind operator, much like an exception.

The rule (ATOMIC RETURN) turns a successful many-step transaction of an STM-expression $H \mid M$ into a single-step computation of the IO-expression $H \mid \texttt{atomically}\ M$. If the transaction yields `retry` then (ATOMIC RETURN) is not applicable, so there is no transition in this case. In the STM Haskell implementation, a transaction that `retry`s is aborted by the run-time system and queued for later execution.

The final group of rules concerns choices within transactions.

(R3) Reductions for OrElse and Or: $M \rightarrow M'$

(STM ORELSE RETURN)
$$\frac{H \mid N_1 \rightarrow^* \mathscr{R}_\mapsto[\texttt{return}_{\text{STM}}\ N_1']}{H \mid (N_1\ \texttt{orElse}\ N_2) \rightarrow \mathscr{R}_\mapsto[\texttt{return}_{\text{STM}}\ N_1']}$$

(STM ORELSE RETRY)
$$\frac{H \mid N_1 \rightarrow^* \mathscr{R}_\mapsto[\texttt{retry}]}{H \mid (N_1\ \texttt{orElse}\ N_2) \rightarrow H \mid N_2}$$

$M\ \texttt{or}\ N \rightarrow M$	(STM OR LEFT)
$M\ \texttt{or}\ N \rightarrow N$	(STM OR RIGHT)

Rules (STM OrElse Return) and (STM OrElse Retry) formalize the idea that N_1 `orElse` N_2 behaves as N_1 if N_1 terminates with $\texttt{return}_{\text{STM}}\ N_1'$. If N_1 terminates with `retry` then its effects are discarded, and we instead run N_2 on the original heap H.

Rules (STM Or Left) and (STM Or Right) define M `or` N as making a nondeterministic choice within a transaction. Such choices may be derived at the level of the IO monad, but this operator introduces nondeterminism into transactions (which otherwise are deterministic). Nondeterminism is used in our programming example only to ensure completeness with respect to its specification; without nondeterminism we would still have soundness.

4.2 Type System

We complete our formalization of STM Haskell by defining a simple type system that prevents ill-formed expressions, such as the inappropriate mixing of pure, STM and I/O expressions.

The type system only permits the reading and writing of transactional variables inside transactions, which a fortiori enforces *static separation* [1] and permits us to reason about transactions as if they occur in a single step.

Let the *domain*, $\mathrm{dom}(M)$, of an expression M be the set of (free) addresses of the transactional variables at top level in the expression. We have $\mathrm{dom}(a \mapsto M) = \{a\}$, $\mathrm{dom}(M \gg\!=_{\mathrm{IO}} N) = \mathrm{dom}(M)$, $\mathrm{dom}(M \gg\!=_{\mathrm{STM}} N) = \mathrm{dom}(M)$, $\mathrm{dom}(M \mid N) = \mathrm{dom}(M) \cup \mathrm{dom}(N)$ and $\mathrm{dom}((\nu a)M) = \mathrm{dom}(M) \setminus \{a\}$. Otherwise, $\mathrm{dom}(M) = \varnothing$. In particular, expressions that are not in a top-level evaluation context should have no free transactional variables, so the type system enforces that their domain is empty.

Here is the syntax of types. For the sake of simplicity, we formalize only a monomorphic type system. We make the standard assumption that uses of Hindley-Milner style polymorphism may be represented by monomorphising via code duplication.

Types:

$u ::= t \mid T$	type
$t ::= t \to t \mid X \mid \mathtt{TVar}\, t \mid \mathtt{IO}\, \varnothing\, t \mid \mathtt{STM}\, \varnothing\, t$	expression type
$T ::= \mathtt{IO}\, \overline{a}\, t \mid \mathtt{STM}\, \overline{a}\, t \mid \mathtt{heap}\, \overline{a} \mid \mathtt{proc}\, \overline{a}$	Configuration type

An *expression type t* describes the eventual value of a pure functional computation. They are either function types ($t \to t$), algebraic datatypes (X), TVar reference types ($\mathtt{TVar}\, t$), IO computation types ($\mathtt{IO}\, \varnothing\, t$) or STM transaction types ($\mathtt{STM}\, \varnothing\, t$). We usually write $\mathtt{IO}\, t$ for $\mathtt{IO}\, \varnothing\, t$, and $\mathtt{STM}\, t$ for $\mathtt{STM}\, \varnothing\, t$.

A *configuration type T* describes the structure, heap and potential return value (if any) of imperative and concurrent expressions. Heap-expressions with domain $\overline{a}$ have type $\mathtt{heap}\, \overline{a}$. Both running transactions and STM-expressions with domain $\overline{a}$ have type $\mathtt{STM}\, \overline{a}\, t$ for some t. Both threads and IO-expressions with domain $\overline{a}$ have type $\mathtt{IO}\, \overline{a}\, t$ for some t.

Finally, the type $\mathtt{proc}\, \overline{a}$ consists of concurrent expressions with domain $\overline{a}$ that are executed in the background for their effects, but whose results will be discarded. Given T, we write $\mathrm{dom}(T)$ for its domain.

We assume that all polymorphic algebraic datatypes X and their constructors f have been monomorphized by instantiating each of their occurrences. For instance, the type `Maybe` a is instantiated at the unit type () as $\mathtt{data\ Maybe}_{()} = \mathtt{Nothing}_{()} \mid \mathtt{Just}_{()}\ ()$. We assume a set of predefined algebraic types (), `Error`, $\mathtt{List}_t$, `Bool`, and $\mathtt{Maybe}_t$, with constructors (), $\mathtt{Nil}_t$, $\mathtt{Cons}_t$, `True`, `False`, $\mathtt{Nothing}_t$, and $\mathtt{Just}_t$.

The return type of an expression is the type of its rightmost thread. The typing rule for parallel composition guarantees that an expression consists of some transactional variables together with either several IO threads or a single rightmost STM thread (currently running a transaction). Moreover, it ensures that there is at most one transactional variable at each location a. It uses the partial non-commutative operation $T \otimes T'$, defined as follows, where $\overline{a} \uplus \overline{b}$ is $\overline{a} \cup \overline{b}$ if $\overline{a}$ and $\overline{b}$ are disjoint.

$$
\begin{array}{rcl}
\mathtt{heap}\, \overline{a} \otimes \mathtt{heap}\, \overline{b} & := & \mathtt{heap}\, \overline{a} \uplus \overline{b} \\
\mathtt{proc}\, \overline{a} \otimes \mathtt{heap}\, \overline{b} & := & \mathtt{proc}\, \overline{a} \uplus \overline{b} \\
\mathtt{heap}\, \overline{a} \otimes \mathtt{STM}\, \overline{b}\, t & := & \mathtt{STM}\, \overline{a} \uplus \overline{b}\, t \\
\mathtt{IO}\, \overline{a}\, t \otimes \mathtt{heap}\, \overline{b} & := & \mathtt{proc}\, \overline{a} \uplus \overline{b} \\
T \otimes \mathtt{proc}\, \overline{a} & := & \mathtt{proc}\, \mathrm{dom}(T) \uplus \overline{a} \text{ if } T \neq \mathtt{STM}\, \overline{b}\, t' \\
T \otimes \mathtt{IO}\, \overline{a}\, t & := & \mathtt{IO}\, \mathrm{dom}(T) \uplus \overline{a}\, t \text{ if } T \neq \mathtt{STM}\, \overline{b}\, t'
\end{array}
$$

In particular, note that $\mathtt{STM}\, \overline{a}\, t \otimes \mathtt{STM}\, \overline{b}\, t'$ is undefined, and hence the type system does not allow two transactions to run at once.

Lemma 1. $(T_1 \otimes T_2) \otimes T_3 = T_1 \otimes (T_2 \otimes T_3) = T_2 \otimes (T_1 \otimes T_3) = (T_2 \otimes T_1) \otimes T_3$.

A typing environment $E \in \mathbf{E}$ is a finite mapping from $\mathbf{X} \cup \mathbf{N}$ to types. Each individual map is written as $a :: \mathtt{TVar}\, t$ or $x :: t$. We write $\overline{x :: t}$ for the environment $x_1 :: t_1, \ldots, x_n :: t_n$ where n is the length of $\overline{x}$ and $\overline{t}$. We write E, E' for the union of E and E' when E and E' have disjoint domains. The full typing rules are given in Figure 1 on page 7.

The rule (T BUILTIN) appeals to a relation $g :: u_1 \to \cdots \to u_n \to u'$, defined as follows, which gives a type for each application of a builtin function g. In the following, all types t, t' and domains $\overline{a}$ are universally quantified, and $\overline{u} \to u'$ stands for u' when $|\overline{u}| = 0$, and otherwise for $u_1 \to \cdots \to u_n \to u'$.

Types for Builtin Functions: $g :: \overline{u} \to u'$

`Y`	$:: (t \to t) \to t$
`equal`	$:: \mathtt{TVar}\, t' \to \mathtt{TVar}\, t' \to \mathtt{Bool}$
`readTVar`	$:: \mathtt{TVar}\, t \to \mathtt{STM}\, t$
`writeTVar`	$:: \mathtt{TVar}\, t \to t \to \mathtt{STM}\, ()$
$\mathtt{return}_{\mathrm{STM}}$	$:: t \to \mathtt{STM}\, t$
`retry`	$:: \mathtt{STM}\, t$
$\gg\!=_{\mathrm{STM}}$	$:: \mathtt{STM}\, \overline{a}\, t' \to (t' \to \mathtt{STM}\, t) \to \mathtt{STM}\, \overline{a}\, t$
`orElse`	$:: \mathtt{STM}\, t \to \mathtt{STM}\, t \to \mathtt{STM}\, t$
`or`	$:: \mathtt{STM}\, t \to \mathtt{STM}\, t \to \mathtt{STM}\, t$
`atomically`	$:: \mathtt{STM}\, t \to \mathtt{IO}\, t$
$\mathtt{return}_{\mathrm{IO}}$	$:: t \to \mathtt{IO}\, t$
$\gg\!=_{\mathrm{IO}}$	$:: \mathtt{IO}\, \overline{a}\, t' \to (t' \to \mathtt{IO}\, t) \to \mathtt{IO}\, \overline{a}\, t$

For example, the function `swap` has type $\mathtt{TVar}\, t \to \mathtt{TVar}\, t \to \mathtt{STM}\, ()$ for each t. Hence, the expression $a \mapsto M \mid b \mapsto N \mid \mathrm{swap}\, a\, b$ is well-typed, by (T PAR), (T CELL), and (T APP). But the expression $a \mapsto M \mid b \mapsto N \mid \mathrm{swap}\ a\ b \mid \mathrm{swap}\ a\ b$ is not well-typed, since it has two STM threads and $\mathtt{STM}\, t \otimes \mathtt{STM}\, t'$ is undefined. As a second example, the expression $\lambda x.(x \mid x)\ (a \mapsto ())$ is not well-typed since the transactional variable $a \mapsto ()$ has type $\mathtt{heap}\, \overline{a}$; $\mathtt{heap}\, \overline{a}$ is not an expression type, so we cannot derive any valid function type $t \to t'$ for the lambda-expression. Indeed, this expression would yield $a \mapsto () \mid a \mapsto ()$, which has two transactional variables with the same location. Such ill-formed expressions are untypable, due to the disjointness conditions of $\otimes$ (see (T PAR)). Similarly, the expression $\lambda x.(x \mid x)\ (a \mapsto () \mid \mathtt{return}_{\mathrm{IO}}\ ())$ is not well-typed since x must have an expression type, which always has empty domain. However, $\lambda x.(x \mid x)$ has type $\mathtt{IO}\, t \to \mathtt{IO}\, t$ for each t, by (T PAR) and (T LAMBDA). Thus, the expression $\lambda x.(x \mid x)\ (\nu a)(a \mapsto () \mid \mathtt{return}_{\mathrm{IO}}\ ())$ is well-typed.

For example, for a well-typed application of `swap`, we have the expected result,

$$a \mapsto M \mid b \mapsto N \mid \mathrm{swap}\ a\ b \quad \to^* \quad a \mapsto N \mid b \mapsto M \mid \mathtt{return}_{\mathrm{STM}}\ ()$$

but an ill-typed application may have an undesirable outcome.

$$
\begin{array}{l}
a \mapsto M \mid b \mapsto N \mid \mathrm{swap}\ a\ b \mid \mathrm{swap}\ a\ b \to^* \\
\qquad\qquad a \mapsto N \mid b \mapsto N \mid \mathtt{return}_{\mathrm{STM}}\ () \mid \mathtt{return}_{\mathrm{STM}}\ ()
\end{array}
$$

$$\text{(T VAR)}\quad \frac{}{E, x :: t \vdash x :: t} \qquad \text{(T ADDR)}\quad \frac{}{E, a :: \mathtt{TVar}\, t \vdash a :: \mathtt{TVar}\, t} \qquad \text{(T EMP)}\quad \frac{}{E \vdash \mathtt{emp} :: \mathtt{heap}\, \varnothing}$$

$$\text{(T LAMBDA)}\quad \frac{E, x :: t \vdash M :: t'}{E \vdash \lambda x.M :: (t \to t')} \qquad \text{(T APP)}\quad \frac{E \vdash M :: t \to t' \quad E \vdash N :: t}{E \vdash M\, N :: t'} \qquad \text{(T BUILTIN)}\quad (g :: \overline{u} \to u')\quad \frac{E \vdash M_1 :: u_1 \quad \dots \quad E \vdash M_n :: u_n}{E \vdash g\, M_1\, \cdots\, M_n :: u'}$$

$$\text{(T ADT)}\quad \left(\mathtt{data}\, X = f_1\, \overline{t_1} \mid \cdots \mid f_m\, \overline{t_m},\ |\overline{t_i}| = |\overline{M}|\right)\quad \frac{E \vdash M_1 :: t_i^1 \quad \dots \quad E \vdash M_m :: t_i^m}{E \vdash f_i\, \overline{M} :: X}$$

$$\text{(T CASE)}\quad (\mathtt{data}\, X = f_1\, \overline{t_1} \mid \cdots \mid f_m\, \overline{t_m})\quad \frac{E \vdash M :: X \quad E, \overline{x_1 :: t_1} \vdash N_1 :: t' \quad \cdots \quad E, \overline{x_m :: t_m} \vdash N_m :: t'}{E \vdash \mathtt{case}\, M\, \mathtt{of}\, \overline{f\, \overline{x} \to\ N} :: t'}$$

$$\text{(T CELL)}\quad \frac{E, a :: \mathtt{TVar}\, t \vdash N :: t}{E, a :: \mathtt{TVar}\, t \vdash a \mapsto N :: \mathtt{heap}\, a} \qquad \text{(T PAR)}\quad \frac{E \vdash M :: T_M \quad E \vdash N :: T_N}{E \vdash M \mid N :: T_M \otimes T_N} \qquad \text{(T RES)}\quad \frac{E, b :: \mathtt{TVar}\, t \vdash M :: \mathtt{heap}\, b \otimes T}{E \vdash (\nu b)M :: T}$$

Figure 1. Type system

Lemma 2 (Subject Reduction)**.**
If $E \vdash M :: u$ and $M \to M'$ then $E \vdash M' :: u$.

From this point, we only consider well-typed processes (that is, such that there is a typing environment under which they have a type). This is motivated by Lemma 2. Moreover, due to the structural definition of the type system, every subexpression of a well-typed process is well-typed. In order to reason compositionally about multi-step reductions, we develop some simple conditions for when two reductions are independent. We use these conditions in our correctness proofs, where we often consider only transactions and reason up to β-equivalence. We begin by dividing reductions into pure $\to_\beta$ and impure $\to_{STM}$. (This distinction is different from the one in [10], where the transition relation is stratified and there is only one kind of top-level transition.)

Definition 3. *We write $M \to_\beta N$ if $M \to N$ can be derived using only the rules in group (R1). We write $\to_{STM}$ for $(\to \setminus \to_\beta)$ and $\twoheadrightarrow$ for $\to_\beta^* \to_{STM}$ (the composition of $\to_\beta^*$ and $\to_{STM}$). We let $=_\beta$ be the smallest equivalence relation containing $\to_\beta$ and $\equiv$.*

Using Lemma 2, we can show that the pure reductions of a single thread are deterministic, and that they commute with reductions in other threads. β-reduction thus enjoys the diamond property.

Lemma 4. *If $M \to M_1$ and $M \to_\beta M_2$ with $M_1 \not\equiv M_2$ then $M_1 \to_\beta M'$ and $M_2 \to M'$ for some M'.*

4.3 Comparison with the Original Semantics

The original STM Haskell semantics [10] is based on three different transition relations: I/O transitions, administrative transitions, and STM transitions. These are defined on structures built from expressions, heaps, and multiple threads. In contrast, our semantics of STM Haskell is in the style of a process calculus (like the semantics of Concurrent Haskell [24], for example) and consists of a single reduction relation defined on expressions, whose syntax subsumes heaps and concurrent threads.

The difference in styles, though, is essentially syntactic. We can show that our reduction relation is equivalent to the original semantics. In the extended version of this paper we show a straightforward translation between our syntax and the original run-time syntax, which yields a strong operational correspondence.

Having finished the development of our theory, we suspect it would be quite possible to recast it directly on top of the original semantics.

Still, we contend that our use of a uniform syntax of expressions is better suited to the development of theories for reasoning about STM Haskell programs. One reason is because it allows us to define contextual equivalence (in Section 6) in the standard way, and to import ideas from process calculus, such as bisimulation, directly. Another reason is that our STM reduction rules (in groups (R2) and (R3)) operate on the adjacent piece H of the heap, as opposed to the full heap; this facilitates reasoning about the part of the heap that is actually used by a transaction. Moreover, we can easily represent parts of the run-time state, such as a thread together with a small piece of the heap. The syntax also allows multiple threads with local state to be composed using the parallel operator.

On the other hand, although our expression syntax is uniform, we need to introduce configuration types, as well as conventional types, to rule out certain ill-formed expressions. This is certainly a cost we must pay for the uniform syntax, but we have not found it so onerous; we need a type system anyway, and the additional rules are not hard to work with.

5. Verifying the Ambient API

We are now in a position to specify the expected behaviour of the Haskell code for the ambient API in Section 3, and to verify it. We do so by showing that the API is a fully abstract implementation of the *ambient calculus*, a small calculus of tree-manipulating processes. Theorem 1, below, shows soundness and completeness of the API, while Theorem 2 shows that ambient processes and their Haskell implementations are in fact bisimilar.

Although the high-level statement of correctness is fairly intuitive, the definitions of correspondence between the run time states of our Haskell code and the ambient calculus are rather detailed and technical. The proofs themselves, in the long version of this paper, are also rather complicated. Still, the theorems and their proofs show the viability of our theory for reasoning about STM Haskell code. To the best of our knowledge, ours is the first theory for equational reasoning about concurrent Haskell programs (as opposed to say the correctness of implementations).

5.1 An (Imperative) Ambient Calculus

Our Haskell API is intended to implement the primitives of an ambient calculus, defined as follows. calculus [5]. Readers familiar with the ambient calculus will notice that every syntactic form of the original calculus also exists as an imperative operation in iAmb.

Syntax of the Ambient Calculus:

$\pi ::=$	simple capability
$\mathsf{into}\ a$	enter a
$\mathsf{out}\ a$	leave a
$\mathsf{open}\ a$	open a
$\mathsf{amb}\ a\ C$	create ambient $a[C]$
$\mathsf{fork}\ C$	fork thread C
$\mathsf{new}(a)\ C$	a fresh in C
$C ::= \pi \mid \mathsf{nil} \mid C.C$	capabilities
$P ::=$	Process
$\mathbf{0}$	inactivity
$\mid a[P]$	ambient
$\mid C.P$	prefixed thread
$\mid (\nu a)P$	restriction
$\mid P \mid P$	parallel
$\mathscr{R} ::= [\cdot] \mid a[\mathscr{R}] \mid (\nu a)\mathscr{R} \mid \mathscr{R} \mid P \mid P \mid \mathscr{R}$	Reduction context

We often omit the $\mathbf{0}$ in $C.\mathbf{0}$ and $a[\mathbf{0}]$. Free and bound names of capabilities and processes are defined as expected. The scope of the bound name a extends to P in $(\nu a).P$ and to C in $\mathsf{new}(a)\ C$.

The reduction semantics of the ambient calculus are defined as follows. Structural equivalence $\equiv$ is the least congruence on processes, with respect to the reduction ($\mathscr{R}$) contexts, that satisfies commutative monoid laws for $\mid$ with $\mathbf{0}$ as unit and the rules below.

Structural Equivalence for Ambient Processes: $P \equiv Q$

$\mathsf{nil}.P \equiv P$	(A EPS)
$(C_1.C_2).P \equiv C_1.(C_2.P)$	(A ASSOC)
$\mathscr{R}[(\nu a)P] \equiv (\nu a)\mathscr{R}[P]$ if $n \notin \mathrm{n}(\mathscr{R})$	(A RES)

Reduction $\rightarrow$ of processes is the least relation satisfying the following rules.

Reduction for Ambient Processes: $P \rightarrow Q$

$b[\mathsf{into}\ a.P \mid Q] \mid a[R] \rightarrow a[b[P \mid Q] \mid R]$	(A IN)
$a[b[\mathsf{out}\ a.P \mid Q] \mid R] \rightarrow b[P \mid Q] \mid a[R]$	(A OUT)
$\mathsf{open}\ a.P \mid a[Q] \rightarrow P \mid Q$	(A OPEN)
$(\mathsf{new}(a)\ C).P \rightarrow (\nu a)C.P$ if $a \notin \mathrm{fn}(P)$	(A NEW)
$\mathsf{amb}\ a\ C.P \rightarrow a[C.\mathbf{0}] \mid P$	(A AMB)
$\mathsf{fork}\ C.P \rightarrow C.\mathbf{0} \mid P$	(A FORK)
$P \rightarrow P' \implies \mathscr{R}[P] \rightarrow \mathscr{R}[P']$	(A R CTX)
$P \equiv\rightarrow\equiv P' \implies P \rightarrow P'$	(A STRUCT)

The first three rules specify how the tree structure can be modified. If $\mathsf{into}\ a$ is executed inside a location b that has a sibling a, then b is moved inside a. Conversely, if $\mathsf{out}\ a$ is executed inside a location b that is a child of a, then b is moved outside a. Finally, $\mathsf{open}\ a$ opens a single child named a of the ambient it is running in.

As a simple example, we take the ambient tree $a[p[\mathsf{out}\ a.\mathsf{into}\ b]] \mid b[]$, where the ambient p represents a packet that intends to move from a to b: $a[p[\mathsf{out}\ a.\mathsf{into}\ b]] \mid b[] \rightarrow a[] \mid p[\mathsf{into}\ b] \mid b[] \rightarrow a[] \mid b[p[]]$. We define the delay operator τ as $\tau.P := a_\tau[] \mid \mathsf{open}\ a_\tau.P$ for some distinguished a_τ.

In this setting, processes such as $C.a[P]$ are ill-formed, since they have no direct correspondent in the API. We instead use $C.\mathsf{amb}\ a\ P$. Formally, we treat only the following subcalculus; processes that result from the execution of a closed process $C.\mathbf{0}$.

Normal form for a subcalculus of iAmb

$$P_N ::= a[P_N] \mid (\nu a)P_N \mid (P_N \mid P_N) \mid C.\mathbf{0} \mid \mathbf{0}$$

We write $\mathbf{P_N}$ for the set of all P_N. As an example, $(\mathsf{out}\ a.\mathsf{into}\ b).\mathbf{0} \in P_N$, but $\mathsf{out}\ a.(\mathsf{into}\ b.\mathbf{0}) \notin \mathbf{P_N}$. Note that $\mathbf{P_N}$ is not closed under structural equivalence, although it is closed (modulo structural equivalence) under reduction. We write $\rightarrow_N$ for $\rightarrow$ restricted to $\mathbf{P_N} \times \mathbf{P_N}$. In the remainder of the paper, we only consider processes $P \in \mathbf{P_N}$. Continuing the running example:

$$\begin{aligned}
&\mathsf{amb}\ a\ (\mathsf{amb}\ p\ (\mathsf{out}\ a.\mathsf{into}\ b)).\mathsf{amb}\ b\ \mathsf{nil}.\mathbf{0}\\
&\quad \rightarrow_N\ a[\mathsf{amb}\ p\ (\mathsf{out}\ a.\mathsf{into}\ b).\mathbf{0}] \mid \mathsf{amb}\ b\ \mathsf{nil}.\mathbf{0}\\
&\qquad \rightarrow_N\ a[p[\mathsf{out}\ a.\mathsf{into}\ b.\mathbf{0}]] \mid \mathsf{amb}\ b\ \mathsf{nil}.\mathbf{0}\\
&\qquad\qquad \rightarrow_N\ a[p[\mathsf{out}\ a.\mathsf{into}\ b.\mathbf{0}]] \mid b[]
\end{aligned}$$

5.2 Statement of Correctness

Cardelli [4] defined a notion of correctness for implementations of the ambient calculus, which we quote here:

> **The problem.** We want to find a (nondeterministic) implementation of the reduction relation $\rightarrow^*$, such that each P_i in an ambient is executed by a concurrent thread (and so on recursively in the subambients $m_j[...]$).
> Desirable properties of the implementation are:
> - *Liveness*: If $P \rightarrow Q$ then the implementation must reduce P.
> - *Soundness*: If the implementation reduces P to Q, then we must have $P \rightarrow^* Q$.
> - *Completeness*: If $P \rightarrow^* Q$, then the implementation must be able (however unlikely) to reduce P to some $Q' \equiv Q$.

Additional Properties. In addition to the three properties proposed by Cardelli, we formalize the following two, and establish all five as Theorem 1.

- *Safety*: If the implementation reduces P to M then M can reduce further to some Q.
- *Termination*: If the implementation of P has an infinite reduction, then P also does.

Compared to [4], we additionally treat the open capability (and in an extended version of this paper, communication of both names and capabilities).

The proof of Theorem 1 proceeds as follows: We begin by giving a simple correspondence between ambient capabilities and their Haskell implementation. In Definition 5, we define how an ambient process is implemented as a Haskell expression, including heap and running capabilities. Definition 6 bridges the gap beween this intensional specification and the expressions that arise when executing the expressions; the main difference is due to the lack of garbage collection in our semantics. Then, Lemma 7 guarantees that the correspondence does not confuse unrelated ambient processes.

With the static correspondence in place, we can then show how it is preserved by execution. Lemma 8 details how the execution of the implementation of a prefix corresponds to its semantics in the ambient calculus. Finally, in the proof of Theorem 1 we close the result of Lemma 8 under contexts, yielding a strong operational correspondence.

5.3 Correspondence between Haskell Code and Ambients

The encoding $\llbracket C \rrbracket$ into Haskell of imperative ambient capabilities is homomorphic, except for two cases:

$$\begin{aligned}
\llbracket \mathsf{new}(a)\ C \rrbracket &:= (\mathtt{new}\ []) \gg\!= \lambda a \rightarrow \llbracket C \rrbracket\\
\llbracket C'.C \rrbracket &:= \llbracket C' \rrbracket \gg \llbracket C \rrbracket
\end{aligned}$$

Continuing the running example, we have:

$$\begin{aligned}
&\llbracket \mathsf{amb}\ a\ (\mathsf{amb}\ p\ (\mathsf{out}\ a.\mathsf{into}\ b)).\mathsf{amb}\ b\ \mathsf{nil} \rrbracket\\
&\quad = \mathtt{amb}\ a\ (\mathtt{amb}\ p\ (\mathtt{out}\ a \gg \mathtt{into}\ b)) \gg \mathtt{amb}\ b\ \mathtt{nil}
\end{aligned}$$

We can then give a compositional definition of what it means for the run-time state of a Haskell program to correspond to (the structure of) a given iAmb process. This definition encapsulates both the heap shape invariant preserved by the functions of the API, and how a given ambient calculus process is represented in the heap. The definition has two levels. At the inner level (Definition 5), we inductively match the structure of an ambient process against a structured decomposition of a process term. At the outer level (Definition 6), we perform sanity checks, open restrictions, discard unused heap items and identify the root ambient.

Definition 5. *We identify association lists with the corresponding binary relations, that must be injective. We identify other lists with multisets. We then say that $(D_n, D_p, D_h, D_c) \in (D_n, D_p, D_h, D'_c) \oplus (D_n, D_p, D_h, D''_c)$ if $D_c \in D'_c \cup D''_c$. We write D for an $\mathtt{AD}(D_n, D_p, D_h, D_c)$. An agent C at location h is $\llbracket C.\mathbf{0} \rrbracket_h := \mathtt{case}\ \llbracket C \rrbracket\ \mathtt{of\ Agent}\ x \to x\ h$.*

Informally, we write $(a \mapsto D, H_h, H, M) \in \mathbf{M}(P)$ if $a \mapsto D$ is the current ambient, H_h its handles, H the data and handles of all its subambients and M the running capabilities in P. $\mathbf{M}(P)$ is inductively defined as follows:

(Completed agent)
$(a \mapsto (D_n, D_p, D_h, []), \Pi_{h \in D_h} h \mapsto a, \mathtt{emp}, \mathtt{return}_{\mathrm{IO}}\ ()) \in \mathbf{M}(P)$ *if* $P \equiv \mathbf{0}$.

(Agent running in the current ambient)
$(a \mapsto (D_n, D_p, D_h, []), \Pi_{h \in D_h} h \mapsto a, \mathtt{emp}, \llbracket C \rrbracket_h) \in \mathbf{M}(P)$ *if* $P \equiv C.\mathbf{0}$ *and* $h \in D_h$

(Child of the current ambient)
$(a \mapsto (D_n, D_p, D_h, [(b,c)]), H_h, H, M) \in \mathbf{M}(P)$ *if* $P \equiv b[Q]$ *and* $H \equiv c \mapsto D' \mid \Pi_{h \in D'_h} h \mapsto c \mid H'$ *where* $(c \mapsto D', \Pi_{h \in D'_h} h \mapsto c, H', M) \in \mathbf{M}(Q)$, $D'_n = b$ *and* $D'_p = \mathtt{Some}\ h'$ *with* $h' \in D_h$

(Parallel decomposition)
$(a \mapsto D, H_h, H, M) \in \mathbf{M}(P)$ *if* $P \equiv Q_1 \mid Q_2$, $H \equiv H_1 \mid H_2$, $M \equiv M_1 \mid M_2$, $D \in D_1 \uplus D_2$ *with* $(a \mapsto D_1, H_h, H_1, M_1) \in \mathbf{M}(Q_1)$ *and* $(a \mapsto D_2, H_h, H_2, M_2) \in \mathbf{M}(Q_2)$.

We can then define what it means for M to be a run-time state corresponding to an ambient process P_0.

Definition 6. *$M \in \mathscr{M}(P_0)$ iff*

1. *There are $P, \overline{e}$ such that $P_0 \equiv (\nu\overline{e})P$ and P is not a $\mathscr{R}[(\nu a)Q]$ (the top-level restrictions of P_0 are $\overline{e}$);*
2. *$\mathrm{fn}(P_0) \subseteq \mathrm{dom}(M)$ and $E \vdash M :: \mathtt{IO}\ \overline{a}\ ()$ for $E := \{a_i :: \mathtt{TVar}\ [\mathtt{Char}] \mid a_i \in \mathrm{dom}(M)\}$ (M has the free names of P_0 in its domain, and is well-typed);*
3. *$M \equiv (\nu ab\overline{ce})(a \mapsto [] \mid b \mapsto (a, \mathtt{None}, D_h, D_c) \mid H_0 \mid H_1 \mid H_2 \mid H_3 \mid M')$ (we can split M into the root ambient, some heaps and some running code);*
4. *$H_0 = \Pi_i d_i \mapsto N_i$ with $\overline{d} \cap \mathrm{fn}(D_h \mid D_c \mid H_1 \mid H_2 \mid H_3 \mid M') = \varnothing$. Moreover, if $N_i = D'$ then $D'_p \neq \mathtt{None}$ (H_0 is unreachable garbage not containing a root ambient);*
5. *$H_1 = \Pi_{n \in \mathrm{fn}(P)} n \mapsto s_n$ with $\varnothing \vdash s_n :: \mathtt{String}$ (H_1 is the free names of P, and is well-typed);*
6. *$H_2 = \Pi_{h \in D_h} h \mapsto b$ (H_2 is the handles of the root ambient);*
7. *There are no $\mathscr{R}_|, a, M''$ such that $H_3 \mid M' \equiv \mathscr{R}_|[(\nu a)M'']$ (there are no further restricted heap cells at the top level); and*
8. *$(a \mapsto D, H_2, H_3, M') \in \mathbf{M}(P)$.*

Both $\mathbf{M}$ and $\mathscr{M}$ characterize $\mathbf{P_N}$ modulo structural equivalence.

Lemma 7. *If $P \equiv Q$ then $\mathscr{M}(P) = \mathscr{M}(Q)$ and $\mathbf{M}(P) = \mathbf{M}(Q)$. Conversely, if $\mathbf{M}(P) \cap \mathbf{M}(Q) \neq \varnothing$ or $\mathscr{M}(P) \cap \mathscr{M}(Q) \neq \varnothing$ then $P \equiv Q$.*

5.4 Operational Semantics of the Implementation

The transactions of the implementations of prefixes exactly correspond to the axioms of the ambient calculus operational semantics, lifted to Haskell using the $\mathbf{M}$ function. We show the case of the into prefix.

Lemma 8. *If $C.\mathbf{0} \equiv \mathsf{into}\ a.P$ and $(d \mapsto D, H_2, H_3, M) \in \mathbf{M}(a[Q] \mid b[C.\mathbf{0} \mid R_1] \mid R_2)$, $M = \mathscr{R}_|[\llbracket C.\mathbf{0} \rrbracket_{h_3}]$, $\{(a, d_2), (b, d_3)\} \in D_c$ with $d_2 \neq d_3$, $H_3 \equiv d_2 \mapsto D2 \mid h_3 \mapsto d_3 \mid d_3 \mapsto D3 \mid H'_3$ with $D3_p = \mathtt{just}\ h$ and $H_2 \equiv h \mapsto d \mid H'_2$, then*
$d \mapsto D \mid H_2 \mid H_3 \mid M \twoheadrightarrow =_\beta d \mapsto D' \mid H_2 \mid d_2 \mapsto D2' \mid h_3 \mapsto d_3 \mid d_3 \mapsto D3' \mid H'_3 \mid \mathscr{R}_|[\llbracket C'.\mathbf{0} \rrbracket_{h_3}]$ where $C'.\mathbf{0} \equiv P$ and
$(d \mapsto D', H_2, d_2 \mapsto D2' \mid h_3 \mapsto d_3 \mid d_3 \mapsto D3' \mid H'_3, \mathscr{R}_|[\llbracket C'.\mathbf{0} \rrbracket_{h_3}]) \in \mathbf{M}(a[Q \mid C'.\mathbf{0} \mid R_1] \mid R_2)$.

5.5 Main Results About the Haskell Code

Our first correctness result establishes direct correspondences between ambient processes and the states of the Haskell implementation; the different properties in this theorem generalize the properties sought by Cardelli [4]. Recall the definition of $\twoheadrightarrow := \rightarrow_\beta^* \rightarrow_{STM}$, intuitively "performing a transaction".

Theorem 1.

- Liveness, Completeness:
If $P \rightarrow_N Q$ and $M \in \mathscr{M}(P)$ then $M \twoheadrightarrow =_\beta \in \mathscr{M}(Q)$.
- Safety, Soundness:
If $M \in \mathscr{M}(P)$ and $M \twoheadrightarrow M'$ then $P \rightarrow_N Q$ with $M' =_\beta \in \mathscr{M}(Q)$.
- Termination:
If $M \in \mathscr{M}(P)$ and M has an infinite reduction then P has an infinite reduction.

Proof sketch.

1. Assume that $M \twoheadrightarrow M'$ and that $M \in \mathscr{M}(P)$ where $P \equiv (\nu\overline{e})P_0$ such that P_0 does not have any top-level restrictions. By assumption, $M \equiv (\nu ab\overline{ce})(a \mapsto$ "" $\mid b \mapsto (a, \mathtt{None}, D_h, D_c) \mid H_0 \mid H_1 \mid H_2 \mid H_3 \mid N)$ such that $H_1 \mid H_2 \mid H_3 \mid N \twoheadrightarrow H'_1 \mid H'_2 \mid H'_3 \mid N'$ and $A := (b \mapsto (a, \mathtt{None}, D_h, D_c), H_2, H_3, N) \in \mathbf{M}(P_0)$. By induction on the derivation of $A \in \mathbf{M}(P_0)$, $N = \Pi_i N_i$ is a parallel composition of several $N_i = \llbracket C_i \rrbracket_{h_i}$. Then there is j such that $H_1 \mid H_2 \mid H_3 \mid \llbracket C_j \rrbracket_{h_j} \twoheadrightarrow H'_1 \mid H'_2 \mid H'_3 \mid N'_j$ with $N' =_\beta N'_j \mid \Pi_{i \neq j} N_i$.

   As shown in Lemma 8 for the in prefix, and in the extended version for the other prefixes, we then have $H_1 \mid H_2 \mid H_3 \equiv H_R \mid d \mapsto D \mid H_h \mid H_S$ such that $P_0 \equiv \mathscr{R}[\mathscr{R}_2[C'_j.Q]]$, $(d \mapsto D, H_h, H_S) \in \mathbf{M}(\mathscr{R}_2[C_j])$ and $H'_1 \mid H'_2 \mid H'_3 \equiv H_R \mid d \mapsto D' \mid H'_h \mid H'_S$ such that $(d \mapsto D', H'_h, H'_S) \in \mathbf{M}(\mathscr{R}'_2[Q])$ where $C_j.\mathbf{0} \equiv C'_j.Q'$ and $\mathscr{R}_2[C'_j.Q'] \rightarrow \mathscr{R}'_2[Q']$ is an axiom. By induction on the derivation of $A \in \mathbf{M}(P_0)$, $M' =_\beta (\nu ab\overline{ce})(a \mapsto$ "" $\mid b \mapsto (a, \mathtt{None}, D_h, D_c) \mid H_0 \mid H'_1 \mid H'_2 \mid H'_3 \mid N'_j \mid \Pi_{i \neq j} N_i)$.
   $M'_\beta \in \mathscr{M}(\mathscr{R}[\mathscr{R}'_2[Q]])$ follows by Lemma 7.

2. Assume that $P \rightarrow P'$. Let $\overline{e}$ be the top-level restrictions of P. If the reduction occured inside an ambient, then there are a, Q, R and contexts $\mathscr{R}_1, \mathscr{R}_2$ where $P \equiv (\nu\overline{e})\mathscr{R}_1[a[\mathscr{R}_2[\pi.Q] \mid R]]$, $\mathscr{R}_2[\pi.Q] \rightarrow \mathscr{R}'_2[Q]$ is an instance of an axiom and $P' \equiv (\nu\overline{e})\mathscr{R}[a[\mathscr{R}'_2[Q] \mid R]]$.

   By assumption $M \in \mathscr{M}(P)$, so $N \equiv \mathscr{R}_|[d \mapsto D \mid H_h \mid H \mid N]$ such that $(d \mapsto D, H_h, H, N) \in \mathbf{M}(a[\mathscr{R}_2[\pi.Q] \mid R])$. Thus, $H \equiv c \mapsto D' \mid H_1 \mid H_2 \mid \Pi_{h \in D'_h} h \mapsto c$ and $N \equiv N_1 \mid N_2$ with $D'_n = b$, $D'_p = \mathtt{Some}\ h'$, $h' \in D_h$ and $D \in D'_1 \uplus D'_2$ with $A := (c \mapsto D'_1, \Pi_{h_i \in D'_h} h_i \mapsto c, H_1, N_1) \in \mathbf{M}(\mathscr{R}_2[\pi.Q])$ and $(c \mapsto D'_2, \Pi_{h \in D'_h} h \mapsto c, H_2, N_2) \in \mathbf{M}(R)$.

   By induction on the derivation of $A \in \mathbf{M}(\mathscr{R}_2[\pi.Q])$, we have $N_1 \equiv \llbracket C' \rrbracket_{h_i} \mid N'_1$ with $C'.\mathbf{0} \equiv \pi.Q$. We treat the case where π is not $\mathsf{new}(a)C$. As shown in Lemma 8 for the into prefix, and in the extended version for the other prefixes, $c \mapsto D'_1 \mid$

$\Pi_{h_i \in D'_h} h_i \mapsto c \mid H_1 \mid [\![C']\!]_{h_i} \twoheadrightarrow c \mapsto D''_1 \mid H'_h \mid H'_1 \mid [\![C_Q]\!]_{h_i}$ with $C_Q.\mathbf{0} \equiv Q$ and $(c \mapsto D''_1, H'_h, H'_1, [\![C_Q]\!]_{h_i}) \in \mathbf{M}(\mathscr{R}'_2[C_Q.\mathbf{0}])$.

If the reduction occurs at top level, we have $P \equiv (\nu\overline{e})(Q \mid R)$, and $N \equiv \mathscr{R}_{|}[d \mapsto D \mid H_h \mid H \mid N]$ such that $(d \mapsto D, H_h, H, N) \in \mathbf{M}(Q \mid R)$. The rest of the proof proceeds analogously.

3. This follows from the completeness above and the fact that $\mathscr{M}(P)$ is $\rightarrow_\beta$-convergent (modulo $\equiv$). □

The proof of this theorem uses Lemma 8 to prove that an agent can progress whenever the corresponding ambient process does and to get the shape of the result of the transition. The proof also uses the compositionality of the calculus; specifically in order to separate an agent (running as part of an expression in the IO monad) and the heap it needs to progress.

Next, we define a notion of bisimulation between ambient processes and STM Haskell expressions.

Definition 9. $\mathbf{R} \subseteq \mathbf{M} \times \mathbf{P}_N$ *is a bisimulation iff for all* $(M,P) \in \mathbf{R}$

- *If* $M \twoheadrightarrow M'$ *then* $P \rightarrow_N P'$ *with* $(M',P') \in \mathbf{R}$*; and*
- *If* $P \rightarrow_N P'$ *then* $M \twoheadrightarrow M'$ *with* $(M',P') \in \mathbf{R}$.

The expression M *is bisimilar to the process* P *if there is some bisimulation* $\mathbf{R}$ *with* $M\ \mathbf{R}\ P$.

Theorem 2. $H_C \mid \mathtt{root}\ [\![C]\!]$ *is bisimilar to* $\tau.C.\mathbf{0}$*, where* $H_c := \Pi_{a_i \in \mathrm{fn}(C)} a_i \mapsto$ *"".*

Bisimulation between the expressions of our calculus and processes of the ambient calculus allows a succinct statement of the theorem. The proof relies on the soundness of bisimulation up to $=_\beta$. We could probably replicate this definition using the original semantics of STM Haskell, but it would require many cases; our reformulated semantics allows a simple and direct definition.

6. Equational Reasoning

One of the nice things about functional programming is that we can hope for two expressions to be equivalent, in the sense that they can be substituted for each other in any context. In this section, we develop a proof technique for a Morris-style contextual equivalence. In particular, we prove a number of equations asserted in [10].

6.1 Contextual Equivalence

We begin by defining a notion of a typed relation, stating that two terms are related at a given type under a typing environment.

Definition 10 (Typed Relation)**.** $\mathbf{R} \subset \mathbf{E} \times \mathbf{M} \times \mathbf{M} \times \mathbf{T}$ *is a* typed relation *if whenever* $(E, M_1, M_2, u) \in \mathbf{R})$ *we have* $E \vdash M_1 :: u$ *and* $E \vdash M_2 :: u$*. We write* $E \vdash M_1\ \mathbf{R}\ M_2 :: u$ *for* $(E, M_1, M_2, u) \in \mathbf{R})$.

An expression M has terminated, written $M\downarrow$, if its rightmost thread `returns`. Termination is our only top-level observation.

Termination

$$\text{(TERM RETURN)}\ \frac{}{\mathtt{return_{IO}}\ M\downarrow} \qquad \text{(TERM RES)}\ \frac{M\downarrow}{(\nu a)M\downarrow} \qquad \text{(TERM PAR)}\ \frac{M\downarrow}{N \mid M\downarrow}$$

An expression M terminates, written $M \Downarrow$, if $M \rightarrow^* N$ such that $N\downarrow$.

Definition 11. Contextual equivalence*, written* $\simeq$*, is the typed relation such that* $E \vdash M_1 \simeq M_2 :: u$ *if and only if for all contexts* $\mathscr{C}$ *such that* $\circ \vdash \mathscr{C}[M_1] :: \mathtt{IO}\ \overline{a}\ ()$ *and* $\circ \vdash \mathscr{C}[M_2] :: \mathtt{IO}\ \overline{a}\ ()$ *we have* $\mathscr{C}[M_1] \Downarrow$ *if and only if* $\mathscr{C}[M_2] \Downarrow$.

6.2 STM Expressions as Heap Relations

Because of the isolation between different transactions provided by the run-time systems, STM expressions are completely defined by their effect on the transactional heap. For simplicity (cf. [16, 17, 29]), we work with a pure heap, where the types of elements in the heap do not mention the STM or IO monads.

Definition 12. *A type* t *is* pure *if it is either* $t_1 \rightarrow t_2$ *where* t_1 *and* t_2 *are pure, if it is* $\mathtt{TVar}\ t'$ *where* t' *is pure, or if it is* X *such that* $\mathtt{data}\, X = f_1\ \overline{t_1} \mid \cdots \mid f_m\ \overline{t_m}$ *where all* t^i_m *are pure. An environment* E *is a* pure store environment *if* E *is of the form* $\cup_i b_i :: \mathtt{TVar}\ t_i$ *where all* t_i *are pure.*

A derivation $E \vdash M :: u$ *is pure, written* $E \vdash_p M :: u$*, if* E *is a pure store environment and* t *is pure in all occurrences of* $\mathtt{TVar}\ t$ *in the derivation. We then say that* M *uses only pure heap.*

Two STM threads that only use pure heap are equivalent if they modify the heap in the same way and return the same result.

Definition 13. Heap transformer equivalence*, written* $=_{HT}$*, is defined by* $E \vdash M =_{HT} N :: u$ *if and only if* $u = \mathtt{STM}\, t$*,* $E \vdash_p M :: u$*,* $E \vdash_p N :: u$*,* M *and* N *are* β*-threads, and for all STM contexts* $\mathscr{R}_{\mapsto}, \mathscr{R}'_{\mapsto}$*, and heaps* H *such that* $E \vdash H :: \mathtt{heap}\ \overline{a}$ *we have* $H \mid M \rightarrow^* \mathscr{R}_{\mapsto}[\mathtt{return_{STM}}\ M']$ *iff* $H \mid N \rightarrow^* \mathscr{R}_{\mapsto}[\mathtt{return_{STM}}\ M']$*; and* $H \mid M \rightarrow^* \mathscr{R}_{\mapsto}[\mathtt{retry}]$ *iff* $H \mid N \rightarrow^* \mathscr{R}'_{\mapsto}[\mathtt{retry}]$.

Theorem 3. *The relation* $=_{HT}$ *is sound, that is,* $=_{HT} \subseteq \simeq$.

Proof. We let $=^{\mathscr{C}}_{HT}$ be the smallest typed congruence containing $=_{HT}$. We prove that $=^{\mathscr{C}}_{HT} \subseteq \simeq$. The proof has three parts:

1. If $E \vdash_P M :: t$ and $E \vdash H :: \mathtt{heap}\ \overline{a}$ then reductions of $H \mid M$ only depend on the pure cells in H.

2. Let $\cong^{\mathscr{C}}_{HT}$ be the smallest typed congruence such that $E \vdash M =^{\mathscr{C}}_{HT} N :: t$ with t pure and M,N closed implies $E \vdash M \cong^{\mathscr{C}}_{HT} N :: t$.

   If $E \vdash_P M :: t$, and G and H are pure heaps related by $\cong^{\mathscr{C}}_{HT}$, then derivatives of $G \mid M$ and $H \mid M$ are related by $\cong^{\mathscr{C}}_{HT}$.

3. We can then derive that $=^{\mathscr{C}}_{HT}$ is a barbed bisimulation, so it is contained in $\simeq$. The interesting case is as follows:

   Assume that $E \vdash M =_{HT} N :: \mathtt{STM}\, t$, $E \vdash H =^{\mathscr{C}}_{HT} G :: \mathtt{heap}\ \overline{c}$ and $H \mid M \rightarrow^* \mathscr{R}_{\mapsto}[B]$. To prove that $G \mid N \rightarrow^* \mathscr{R}'_{\mapsto}[B']$ such that $E \vdash \mathscr{R}_{\mapsto}[B] =^{\mathscr{C}}_{HT} \mathscr{R}'_{\mapsto}[B'] :: \mathtt{STM}\ \overline{c}\ t$ we first use 1. and 2. to prove that $G \mid M \rightarrow^* \mathscr{R}''_{\mapsto}[B'']$ such that $E \vdash \mathscr{R}_{\mapsto}[B] =^{\mathscr{C}}_{HT} \mathscr{R}''_{\mapsto}[B''] :: \mathtt{STM}\ \overline{c}\ t$.

   Then $G \mid N \rightarrow^* \mathscr{R}'_{\mapsto}[B']$ such that $E \vdash \mathscr{R}''_{\mapsto}[B''] =^{\mathscr{C}}_{HT} \mathscr{R}'_{\mapsto}[B'] :: \mathtt{STM}\ \overline{c}\ t$ by the definition of $=_{HT}$. By transitivity, $E \vdash \mathscr{R}_{\mapsto}[B] =^{\mathscr{C}}_{HT} \mathscr{R}'_{\mapsto}[B'] :: \mathtt{STM}\ \overline{c}\ t$. □

We write $M \leftrightarrow N$ if for all pure store environments E and types t such that $E \vdash_p M :: \mathtt{STM}\, t$ and $E \vdash_p N :: \mathtt{STM}\, t$ we have $E \vdash M =_{HT} N :: \mathtt{STM}\, t$. We can now use Theorem 3 to prove classic equations between expressions.

6.3 Proving the Monad Laws

To be a proper monad, the $\mathtt{return_{STM}}$ and $\gg\!=_{\mathrm{STM}}$ functions must work together according to three laws:

Lemma 14.

1. $((\mathtt{return_{STM}}\ M) \gg\!=_{\mathrm{STM}} N) \leftrightarrow NM$*.*
2. $(M \gg\!=_{\mathrm{STM}} \lambda x.\mathtt{return_{STM}}\ x) \leftrightarrow M$
3. $((M \gg\!=_{\mathrm{STM}} f) \gg\!=_{\mathrm{STM}} g) \leftrightarrow (M \gg\!=_{\mathrm{STM}} (\lambda x.fx \gg\!=_{\mathrm{STM}} g))$

Proof.

1. The only transition of $H \mid (\mathtt{return_{STM}}\ M) \gg\!=_{\mathrm{STM}} N$ is $H \mid (\mathtt{return_{STM}}\ M) \gg\!=_{\mathrm{STM}} N \rightarrow\equiv H \mid NM$

2. Take $M' \in \{\mathtt{retry}, \mathtt{return_{STM}}\ M''\}$.
   We then have $H \mid M \rightarrow^* \mathscr{R}_{\mapsto}[M']$ iff

$M \gg\!=_{\mathrm{STM}} \to^{*} \mathscr{R}_{\mapsto}[M'] \gg\!=_{\mathrm{STM}} \lambda x.\mathtt{return}_{\mathrm{STM}}\, x.$
$\equiv \mathscr{R}_{\mapsto}[M' \gg\!=_{\mathrm{STM}} \lambda x.\mathtt{return}_{\mathrm{STM}}\, x]$

We proceed by case analysis on M'.

- $M' = \mathtt{retry}$ iff, using (STM BIND RETRY), $\mathscr{R}_{\mapsto}[M' \gg\!=_{\mathrm{STM}} \lambda x.\mathtt{return}_{\mathrm{STM}}\, x] \to \mathscr{R}_{\mapsto}[\mathtt{retry}]$.
- $M' = \mathtt{return}_{\mathrm{STM}}\, M''$ iff $\mathscr{R}_{\mapsto}[M' \gg\!=_{\mathrm{STM}} \lambda x.\mathtt{return}_{\mathrm{STM}}\, x] \to$ $\to \mathscr{R}_{\mapsto}[\mathtt{return}_{\mathrm{STM}}\, M'']$, using (STM BIND RETURN) and (BETA).

3. as 2. □

6.4 Proving Other Equations

We prove classical single-threaded imperative equivalences, such as the commutativity of accesses to independent memory cells.

Lemma 15.

- $(\mathtt{readTVar}\, a \gg\!=_{\mathrm{STM}} \lambda x.\mathtt{writeTVar}\, a\, x) \leftrightarrow \mathtt{return}_{\mathrm{STM}}\, ()$.
- $(\mathtt{writeTVar}\, a\, M \gg_{\mathrm{STM}} \mathtt{writeTVar}\, b\, N) \leftrightarrow$ $(\mathtt{writeTVar}\, b\, N \gg_{\mathrm{STM}} \mathtt{writeTVar}\, a\, M)$ *if* $a \neq b$.
- $(\mathtt{readTVar}\, a \gg\!=_{\mathrm{STM}} \lambda x.\mathtt{writeTVar}\, b\, M \gg\!=_{\mathrm{STM}} \mathtt{return}_{\mathrm{STM}}\, x)$ $\leftrightarrow (\mathtt{writeTVar}\, b\, M \gg_{\mathrm{STM}} \mathtt{readTVar}\, a)$ *if* $a \neq b$

We also prove absorption and associativity laws for `orElse`, as proposed in [10], and associativity and commutativity laws for `or`.

Lemma 16.

1. $\mathtt{orElse}\, \mathtt{retry}\, M \leftrightarrow M$
2. $\mathtt{orElse}\, M\, \mathtt{retry} \leftrightarrow M$
3. $\mathtt{orElse}\, M_1\, (\,\mathtt{orElse}\, M_2\, M_3) \leftrightarrow \mathtt{orElse}\, (\,\mathtt{orElse}\, M_1\, M_2)\, M_3$
4. $\mathtt{or}\, M\, N \leftrightarrow \mathtt{or}\, N\, M$
5. $\mathtt{or}\, M_1\, (\,\mathtt{or}\, M_2\, M_3) \leftrightarrow \mathtt{or}\, (\,\mathtt{or}\, M_1\, M_2)\, M_3$

7. Related Work

Prior semantics for languages with STM, such as STM Haskell [10], were developed with an aim to specify and compare transaction models [33] and their implementations [30, 14, 1], or to study the interaction of transactions with other language features [20]. Hu and Hutton [13] show correctness for a compiler for a small transaction language, inspired by STM Haskell. In contrast, our semantics is designed to enable equational reasoning about source programs. In this respect, the development closest to ours is of an equational theory for a process algebra with STM [2]; this work is not about actual code, and includes no substantial example.

Proof techniques for STM programs have focused on checking invariants of shared transactional state, not on equational reasoning. An extension of STM Haskell with run time invariant checking [12] defines a semantics and implementation but does not attempt program verification. A program logic [21] allows invariants to be specified as pre- and post-conditions within a dependent type system and proofs are by typechecking; unlike STM Haskell, this system has no explicit transaction abort or `retry`.

Our main case study is a verification of a centralized shared-memory implementation of ambients. There are several distributed implementations of ambients described in the literature [6, 25, 7]. These have also been verified using techniques from process calculus, but the algorithms are based on message-passing rather than transactional memory. We recently learnt of an independent, but unverified, implementation of ambients within STM Haskell [32]. We intend to investigate whether our verification techniques also apply to this code.

8. Conclusions

It has been prominently argued that functional progamming in pure languages like Haskell facilitates equational reasoning [15] and that transactional memory enables compositional reasoning about concurrent programs [11]. Here we realize this promise in the context of STM Haskell and show how to verify equational properties of a sizeable STM program.

As future work, we want to extend our proof techniques to statically check invariants, and to investigate connections between our model of heaps and concurrency, spatial logics for process calculi, and separation logics for imperative programming languages. A possible further case study to exercise our theory would be to verify an STM implementation of the join calculus.

Acknowledgements Discussions with Cédric Fournet, Tim Harris, Simon Peyton Jones, and Claudio Russo were useful.

A. Source code

This appendix contains the remainder of the source code for the ambient API of Section 3.

Ambient Functions

```
nil = Agent $ \s -> return ()

new arg = Agent $ \s ->
            atomically $ newTVar arg

root agent = do
   rHandle <- (atomically $
               do rName <- newTVar "root";
                  newAmb Nothing rName);
   case agent of Agent f -> f rHandle

amb a agent = Agent $ \bHandle -> do {
  aHandle <- atomically $ do {
    aHandle  <- newAmb (Just bHandle) a;
    aAmbient <- readTVar aHandle;
    AD (n,p,h,c) <- readAmb bHandle;
    writeAmb bHandle (AD (n,p,h,(a,aAmbient):c));
    return aHandle};
  forkIO $ case agent of Agent f -> f aHandle
  ;return ()}

out c = Agent $ \bHandle -> atomically $ do {
  bAmbient        <- readTVar bHandle;
  AD (bn,bp,bh,bc) <- readAmb bHandle;
  cHandle         <- parentOf bHandle;
  AD (cn,cp,ch,cc) <- readAmb cHandle;
  aHandle         <- if (cn == c)
                       then parentOf cHandle
                       else retry;
  AD (an,ap,ah,ac) <- readAmb aHandle;
  writeAmb aHandle (AD (an,ap,ah,
                        (bn,bAmbient):ac));
  writeAmb cHandle (AD (cn,cp,ch,
                   delete (bn,bAmbient) cc));
  writeAmb bHandle (AD (bn,Just aHandle,bh,bc))}

open c = Agent $ \aHandle -> atomically $ do {
  aAmbient        <- readTVar aHandle;
  AD (an,ap,ah,ac) <- readAmb aHandle;
  cAmbient        <-  lookup' c ac;
  AD (cn,cp,ch,cc) <- readTVar cAmbient;
  rePoint aAmbient ch;
  writeAmb aHandle
         (AD (an, ap, ah++ch,
             (delete (cn,cAmbient) ac)++cc))}
```

```
fork agent = Agent $ \s -> do {
    atomically $ return ();
  ; forkIO $ case agent of
                Agent f -> f s
  ; return ()}
```

Helper Functions

```
newAmb  :: (Maybe Handle) -> Name -> STM Handle
newAmb p n = do {
  me  <- newTVar (AD (n, p, [], []));
  pMe <- newTVar me;
  writeTVar me (AD (n, p, [pMe], []));
  return pMe}

rePoint :: Ambient -> [Handle] -> STM ()
rePoint a []    = return ()
rePoint a (x:xs) = do writeTVar x a;
                   rePoint a xs
```

Non-deterministic Lookup

```
choose :: [a] -> STM a
choose [] = retry
choose (x:[]) = return x
choose (x:xs) = or (return x) (choose xs)

assoc :: Name -> [(Name,Ambient)] -> [Ambient]
assoc f [] = []
assoc f ((a,x):xs) =
     if f==a then (x:assoc f xs)
     else assoc f xs
lookup' x l = choose (assoc x l)
```

References

[1] Abadi, M., Birrell, A., Harris, T., and Isard, M. Semantics of transactional memory and automatic mutual exclusion. In *Proc. POPL'08* (2008), pp. 63–74.

[2] Acciai, L., Boreale, M., and Dal-Zilio, S. A concurrent calculus with atomic transactions. In *Proc. ESOP'07* (2007), R. D. Nicola, Ed., vol. 4421 of *LNCS*, Springer, pp. 48–63.

[3] Borgström, J., Bhargavan, K., and Gordon, A. D. A compositional theory for STM Haskell. Tech. Rep. MSR-TR-2009-66, Microsoft Research, 2009.

[4] Cardelli, L. Mobile ambient synchronization. Technical Note 1997-013, Digital Equipment Corporation, Systems Research Center, 1997.

[5] Cardelli, L., and Gordon, A. D. Mobile ambients. *Theoretical Computer Science 240* (2000), 177–213.

[6] Fournet, C., Lévy, J.-J., and Schmitt, A. An asynchronous, distributed implementation of mobile ambients. In *Proc. TCS'00* (2000), Springer, pp. 348–364.

[7] Giannini, P., Sangiorgi, D., and Valente, A. Safe ambients: Abstract machine and distributed implementation. *Science of Computer Programming 59*, 3 (2006), 209–249.

[8] Guerraoui, R., Henzinger, T. A., and Singh, V. Completeness and nondeterminism in model checking transactional memories. In *Proc. CONCUR'08* (2008), F. van Breugel and M. Chechik, Eds., vol. 5201 of *LNCS*, Springer, pp. 21–35.

[9] Harris, T., and Fraser, K. Language support for lightweight transactions. In *Proc. OOPSLA'03* (2003), pp. 388–402.

[10] Harris, T., Marlow, S., Peyton Jones, S., and Herlihy, M. Composable memory transactions. In *Proc. PPOPP'05* (2005), K. Pingali, K. A. Yelick, and A. S. Grimshaw, Eds., ACM, pp. 48–60.

[11] Harris, T., Marlow, S., Peyton Jones, S., and Herlihy, M. Composable memory transactions. *Communications of ACM 51*, 8 (2008), 91–100.

[12] Harris, T., and Peyton Jones, S. Transactional memory with data invariants. In *Proc. TRANSACT'06* (2006).

[13] Hu, L., and Hutton, G. Towards a verified implementation of software transactional memory. In *The Symposium on Trends in Functional Programming* (2008). To appear.

[14] Huch, F., and Kupke, F. A high-level implementation of composable memory transactions in Concurrent Haskell. In *Proc. Implementation and Application of Functional Languages* (2005), vol. 4015 of *LNCS*, Springer, pp. 124–141.

[15] Hughes, J. Why functional programming matters. *Computer Journal 32*, 2 (Apr. 1989), 98–107.

[16] Jeffrey, A., and Rathke, J. A theory of bisimulation for a fragment of concurrent ML with local names. *Theoretical Computer Science 323*, 1–3 (2004), 1–48.

[17] Koutavas, V., and Wand, M. Small bisimulations for reasoning about higher-order imperative programs. In *Proc. POPL '06* (2006), ACM, pp. 141–152.

[18] Lee, E. The problem with threads. *COMPUTER* (2006), 33–42.

[19] Milner, R. *Communication and Concurrency*. Prentice Hall, 1989.

[20] Moore, K. F., and Grossman, D. High-level small-step operational semantics for transactions. In *Proc. POPL'08* (2008), pp. 51–62.

[21] Nanevski, A., Govereau, P., and Morrisett, G. Type-theoretic semantics for transactional concurrency. Tech. Rep. TR-08-07, Harvard University, July 2007.

[22] O'Leary, J., Saha, B., and Tuttle, M. R. Model checking transactional memory with Spin. In *Proc. PODC'08* (2008), R. A. Bazzi and B. Patt-Shamir, Eds., ACM, p. 424.

[23] Ousterhout, J. Why threads are a bad idea (for most purposes). In *Presentation given at the 1996 Usenix Annual Technical Conference, January* (1996).

[24] Peyton Jones, S., Gordon, A., and Finne, S. Concurrent Haskell. In *Proc. POPL'96* (1996), pp. 295–308.

[25] Phillips, A., Yoshida, N., and Eisenbach, S. A distributed abstract machine for boxed ambient calculi. In *Proc. ESOP'04* (2004), D. A. Schmidt, Ed., vol. 2986 of *LNCS*, Springer, pp. 155–170.

[26] Regev, A., Panina, E. M., Silverman, W., Cardelli, L., and Shapiro, E. BioAmbients: An abstraction for biological compartments. *Theoretical Computer Science 325*, 1 (2004), 141–167.

[27] Ringenburg, M. F., and Grossman, D. AtomCaml: first-class atomicity via rollback. In *Proc. ICFP '05* (2005), ACM, pp. 92–104.

[28] Sangiorgi, D. On the bisimulation proof method. *Mathematical Structures in Computer Science 8* (1998), 447–479.

[29] Sangiorgi, D., Kobayashi, N., and Sumii, E. Environmental bisimulations for higher-order languages. In *Proc. LICS'07* (2007), IEEE Computer Society, pp. 293–302.

[30] Scott, M. L. Sequential specification of transactional memory semantics. In *Proc. TRANSACT'06* (2006).

[31] Shavit, N., and Touitou, D. Software transactional memory. *Distributed Computing 10*, 2 (1997), 99–116.

[32] Sunshine-Hill, B., and Zarko, L. STM versus locks, ambiently, May 2008. CIS 552 Final Project, University of Pennsylvania.

[33] Vitek, J., Jagannathan, S., Welc, A., and Hosking, A. L. A semantic framework for designer transactions. In *Proc. ESOP'04* (2004), D. A. Schmidt, Ed., vol. 2986 of *LNCS*, Springer, pp. 249–263.