

# Cache Coherence Protocol verification using $\Omega$ mega

Ahn, Ki Yung

Portland State University, Portland OR 97207, USA  
Computer Science Technical Report #TR-08-02

**Abstract.** We verify some correctness properties of the DASH cache coherence protocol using  $\Omega$ mega.  $\Omega$ mega is a language with a rich type system featuring GADTs, type functions, and user-guided type checking rules. Cache coherence protocols have both safety properties and liveness properties. We show how to describe some of the safety properties of DASH cache coherence protocol in  $\Omega$ mega. Since liveness properties are not easily expressed by types, we investigate invariants sufficient to imply some of the liveness properties of concern, and assert those invariants as well in the type system of  $\Omega$ mega. Using  $\Omega$ mega, we can have both a working program and an automatically checked proof of its properties because  $\Omega$ mega is both a programming language and a logic. Tightly coupled programs and their properties using types guides us both in the construction of the program and in strategies for modification that preserves the essential properties.

This technical report is based on the paper submitted to the 2007 Spring Research Proficiency Examination of Computer Science Department at Portland State University. The Research Proficiency Examination is a part of the Ph.D. candidacy examination process, which includes an oral presentation and a written paper. I give thanks to my advisor Tim Sheard and my Research Proficiency Examination Committee members: Tom Shrimpton, Andrew Black, Nirupama Bulusu, James G. Hook, and Leonard Shapiro.

# Table of Contents

Cache Coherence Protocol verification using $\Omega$ mega .....	1
<i>Ahn, Ki Yung</i>	
1 Introduction.....	3
1.1 Overview .....	3
1.2 DASH is Like a Library System .....	4
1.3 Properties of the DASH Cache Coherence Protocol .....	5
2 Operational Semantics .....	7
2.1 Notation .....	7
2.2 Overview of the Protocol .....	9
2.3 Communication Rules.....	9
2.4 Reaction Rules.....	10
3 Haskell Implementation .....	16
3.1 Data Definition .....	16
3.2 Implementing the Rules of the Operational Semantics.....	16
3.3 Simulation.....	17
4 Using $\Omega$ mega to Enforce Properties .....	20
4.1 Static Properties on Directories .....	20
4.2 Approach for Liveness Properties .....	24
5 Related Works .....	26
6 Conclusions & Future Work .....	26
7 Acknowledgements .....	27

# 1 Introduction

## 1.1 Overview

We have two goals in mind in this work. First, to investigate how to formalize and verify correctness properties of cache coherence protocols in a dependently typed language like  $\Omega$ mega [She04]. Second, to examine the strength of  $\Omega$ mega on a real world example. We seek useful patterns for programming in  $\Omega$ mega and possible improvements to the  $\Omega$ mega system.

There is a strong need for verifying correctness for distributed algorithms, including cache coherence protocols. The behaviors of distributed systems are much harder to reason about than the behaviors of sequential systems, because distributed systems have larger state space and more intermediate paths between states due to parallelism, or non-determinism. To verify the correctness of the system, we either test exhaustively over all possible states, which is the strategy of model checking, or formally prove the correctness by logic, which is the strategy of theorem proving. Our approach is the latter.

Proofs in theorem proving approach are scalable since they are independent over the size of the system. Model checking has been an effective method for system verification, but it can only verify systems of limited size. The model checking approach are not generally scalable as the number of nodes increases, unless we either use a hybrid approach of abstract interpretation that statically abstracts the system behavior, or devise a clever way to reduce the state space. Because of the limitations on scalability of model checking, the state space reduction is an important issue [ID93]. Verifying coherency for shared memory systems by exhaustively validating their execution path is, in general, NP-Complete [CLS03].

Cache coherence protocols ensures that every read obtains the most recent update<sup>1</sup> in multiprocessor systems. Cache coherence protocols prevent cache coherence problems, which may occur when there are two different cache contents for the same memory location [HP06]. A typical approach is to distinguish between shared cache (read only) and exclusive cache (write allowed) rights. The DASH system is a distributed shared memory systems with a directory based cache coherence protocol.

Although DASH [LLG<sup>+</sup>92] is a well studied system, we have not yet found formalizations of the system outside the model checking community. We formalize the operational semantics of a simplified version of DASH cache coherence protocol described in Lenoski et al. [LLG<sup>+</sup>90]. We built a Haskell simulator following this semantics as a proof of concept. Starting from the Haskell implementation as a reference, we rewrite the program in  $\Omega$ mega, enriching it with proofs of some correctness properties exploiting the type system of  $\Omega$ mega. We have devised reflective data structures to enforce safety properties of the protocol. We are investigating some invariants of liveness properties.

<sup>1</sup> The meaning of ‘most recent’ may differ between memory consistency models.

We are studying cache coherence protocols because we want to apply  $\Omega$ mega to a realistic problem. The duality of  $\Omega$ mega, both a programming language and a logic, allows us to have both proofs of the protocol properties and an implementation of the protocol.  $\Omega$ mega has shown its ability to prove interesting properties of basic elements of programming such as natural numbers, data structures (e.g. lists and trees), and small languages. In this work, we use  $\Omega$ mega to prove properties of a system containing several of those elements. Since  $\Omega$ mega is a programming language, we can also implement the system in  $\Omega$ mega. Having programs with automatically checked proofs by the type system helps ensure correctness both for the first time development and for future changes. In fact, the proof and the implementation are tied together. We prove properties in  $\Omega$ mega by finding an implementation that matches the type signatures that describe the program properties. Cache coherence protocols are an interesting problem of a reasonably challenging complexity. Therefore, we have chosen the DASH cache coherence protocol for the case study.

We believe  $\Omega$ mega has some merits compared to typical formal verification tools like theorem provers or model checkers.  $\Omega$ mega has a lower barrier of entry for programmers without intensive training in logic, because the type system of  $\Omega$ mega embraces the specifications and proofs of program properties. Programmers are more comfortable programming with types than dealing directly with logic. The duality of  $\Omega$ mega, both a programming language and a logic, is based on the Curry-Howard correspondence [How80]: a program  $p$  that has type  $t$  also means that  $p$  is a proof for the proposition  $t$ . Having programs with automatically checked proofs by the type system helps ensure correctness both for the first time development and for future changes.  $\Omega$ mega programs are correct by construction, obeying the properties specified in their types, because we prove the properties as we write the programs. In other words, we derive programs as proofs for the specifications described in their types. This relates closely to Dijkstra's vision of deriving correct programs during program construction [Dij68].

## 1.2 DASH is Like a Library System

DASH is a distributed shared memory system with directory based cache coherence protocol. A *directory* is one of its key components. Directories keep records of cached out memory pages. A local memory page may be cached by either the local processor, or other remote processors, or both. Marking a directory in the node when it grants a cache rights to its memory is like keeping a record in the library when a book is checked out.

In Portland, a dozen or more collaborating libraries share their collections, so that users may check out a book from any local library. Each individual library in Portland is analogous to a *processing node* in the DASH system. A local library should get a book from another remote library before a user checks out the book from the local library. Similarly, a processing node in the DASH system caches a memory page before a processor in the node refers to the memory content in the local cache.

Libraries can serve the community better with a correct and clever book sharing protocol. Most books return to the home library after being lent out to another library. But for some popular books, an interesting situation may arise. What if a copy of a recent bestseller belonging to the Central Library is requested by the North Portland Library while it is currently checked out by a user of the Gresham Library? It is better to tell the Gresham Library to send the book directly to North Portland Library than to have it returned back to the Central Library from Gresham Library and then send it out to North Portland Library. What happens in an efficient library system is almost exactly like what happens in the DASH system. The only fundamental difference between a library system and DASH is that memory is much easier to replicate than making copies of a book. Memory pages can be cached out to several different nodes as a shared read, while books in libraries can only be exclusively lent out. Because of this property of memory, even more interesting things can happen in the DASH protocol than in the library protocol.

### 1.3 Properties of the DASH Cache Coherence Protocol

The DASH cache coherence protocol has *safety properties* which must hold all the time and *liveness properties* which should eventually hold. Distributed systems, in general, have both kinds of these properties [Lyn96].

#### 1.3.1. Safety Properties.

Safety properties are naturally captured by the type system. We can view a type as a simple safety property. When a variable `c` has type `Char`, the value of `c` is always some character, throughout the program execution, regardless of the program input or the program execution path.

Directories have safety properties. A Directory can have at most one exclusive record. When a directory has an exclusive record, it cannot have shared records. A node cannot give exclusive cache rights of a memory page to two different nodes at the same time, just as a library cannot lend out the same book twice before it is returned. Exclusive record and shared record cannot coexist in a directory; When a certain memory page is exclusively cached, other nodes must have no shared cache rights to that memory page; Otherwise cache coherency breaks down when the node with exclusive cache rights updates the memory content.

#### 1.3.2. Liveness Properties.

Our approach is to ensure liveness properties by asserting invariants that are sufficient to imply the liveness properties of our concern. Invariants are properties that are true of all reachable states of a system [Lyn96]. Invariant assertion is a classical method for proving correctness [Flo67][Hoa69]. The strategy of invariant assertion is claiming that if some properties (invariants) holds all the time then

some other properties (liveness properties) will eventually hold. For example, we can prove the correctness of the insertion sort algorithm based on the invariant that the partial sequence created by insertion is always sorted.

*Invariants for Request Completion.* A request message always invokes either an acknowledgement or a negative acknowledgement, and may invoke some other additional messages. The acknowledgement message invoked by the request message completes the request. The negative acknowledgement message invoked by the request message reinvokes the request message when it is returned the requester. As a result, we observe an invariant. When there is a pending request, the communication channel of the system contains one or more of the following: the request message, an acknowledgement, or a negative acknowledgement.

*Progress and Fair Scheduling.* Proving liveness properties of an asynchronous system with invariant assertion requires certain assumptions. Those assumptions are progress and fair scheduling. Progress means that the system must continually make steps. Otherwise, the requests will never complete regardless of the invariants. In programming languages, we have a classical method for proving progress that preserves invariants, called *subject reduction*. We are trying to prove subject reduction of the protocol with the help of the  $\Omega$ mega type system (Section 4.2). We also need fair scheduling to prevent starvation of any request. However, we do not aim to prove fair scheduling, since scheduling is, in general, orthogonal to the design and implementation of protocols.

## 2 Operational Semantics

We define an operational semantics of a simplified version the DASH cache coherence protocol. This is our original work solely based on Lenoski et al. [LLG<sup>+</sup>92]. The DASH system is a fairly complex distributed shared memory system with multiple processing nodes. Each processing node in DASH has its own processors and memory. We simplify the DASH system in three ways. First, we only consider the cache coherence control structures for one memory page per node. Second, we simplify the communication by using a single bus, a reliable FIFO communication channel. Third, nodes in our model process messages atomically, blocking other node's communication while one node is processing a message. The first is a natural simplification to focus on the essence of the protocol. The second and the third simplification makes the protocol behave less parallel. We simplify the DASH cache coherence protocol to start with a problem of manageable complexity.

### 2.1 Notation

$$\begin{aligned}
\tau \in State &= \{\text{In, Sh, Ex}\} & i, j, k, s \in [N] &= \{0, 1, \dots, N-1\} \\
b, m \in Msg &::= \overset{i \rightarrow j}{\text{Ask}_t \tau k} \mid \overset{i \rightarrow j}{\text{Ack}_t \tau k s} \mid \overset{i \rightarrow j}{\text{Nak}_t \tau k} \mid \overset{i \rightarrow j}{\text{Wrb}_t \tau k} \\
Msg_{\perp} &= Msg \cup \{\perp\} & bs \in Bus &= Msg^* \\
(N, bs) \in World &= ([N] \xrightarrow{\text{fin}} Node) \times Bus \\
N_n = \mathcal{N}(n) \in Node &= Dir \times RAC \times Prg_n \\
D \in Dir &= [N] \xrightarrow{\text{fin}} State \cong State^N \\
(C, P) \in RAC &= Dir \times PMS \\
PMS &= [N] \xrightarrow{\text{fin}} Msg_{\perp} \cong (Msg_{\perp})^N
\end{aligned}$$

$$ms \in Prg_n = \left\{ \overset{n \rightarrow j}{\text{Ask}_t \tau k} \mid \overset{n \rightarrow j}{\text{Ask}_t \tau k} \in Msg, \tau \in \{\text{Sh, Ex}\}, t \text{ is unique} \right\}^*$$

*Natural Numbers.*  $[N]$  is a set of natural numbers less than  $N$ , which is the number of the nodes in a World. We use the symbols  $i, j, k$  node index and  $s$  for pending invalidation counter, which are elements of  $[N]$ .

*Memory State.*  $State$  is the set of three possible cache states: In (invalid or uncached), Sh (shared), and Ex (exclusive or dirty). We use the symbol  $\tau$  for an element of  $State$ .

*Message.*  $Msg$  is the set of messages. There are four types of messages: Ask (request), Ack (acknowledgement), Nak (negative-acknowledgement) and WrB (writeback) messages.  $Msg_{\perp}$  is a set of messages augmented by bottom ( $\perp$ ), which means empty slot for a message.

- An Ask message  $\overset{i \rightarrow j}{\text{Ask}}_t \tau k$  from  $i$  asks  $j$  for the cache access right  $\tau$  (In, Sh, or Ex) to memory  $k$  with message id  $t$ . Each Ask message fetched from the programs of the nodes has a unique message id, and this message id is incorporated into all subsequent messages caused by that original Ask message.
- An Ack message  $\overset{i \rightarrow j}{\text{Ack}}_t \tau k s$  is a positive reply to an Ask message with message id  $t$ , coming from  $i$ , granting  $j$  the cache access right  $\tau$  to memory  $k$ , where  $s$  is pending invalidation counter when  $\tau$  is exclusive (Ex).
- A Nak message  $\overset{i \rightarrow j}{\text{Nak}}_t \tau k$  from  $i$  is a negative reply to the ask message with message id  $t$ , refusing  $j$  the cache access right  $\tau$  to memory  $k$ .
- A WrB message  $\overset{i \rightarrow j}{\text{WrB}}_t \tau k$  with message id  $t$  from  $i$  to  $j$  generated by the ask message  $t$  forwarded from  $k$ , notifies that  $i$  has cache access right  $\tau$  to  $j$ 's memory after sending this writeback message. WrB messages may occurs when an Ask message forwards to a node that holds exclusive cache rights to memory  $k$ .

*World.* A World is a model of the DASH system. We use the notation  $(\mathcal{N}, bs)$  as an element of the *World*. A World is a pair of  $\mathcal{N}$ , a list of  $N$  nodes, and the bus  $bs$ , the reliable FIFO communication channel possibly containing zero or more messages. We refer to the  $n$ th node as either  $\mathcal{N}(n)$  or  $N_n$ .

*Node.* A Node is a triple of a directory, a remote access cache, and a program. We use the notation  $(D, rac, ms)$  to represent the node as a triple. Since  $rac$  is a pair, we more often write  $(D, (C, P), ms)$  to denote a node.

*Directory.* The directory structure represents the node's view of who owns what rights to its own memory. We use the symbol  $D$  to denote a directory. We use notation  $D(k)$  to denote  $k$ th element of  $D$ .

*Remote Access Cache.* RAC is a pair of the node's current cache rights and pending requests for each cache entry. We use the notation  $(C, P)$  to describe RAC. For the cache entry for memory  $k$ ,  $C(k)$  is the cache state and  $P(k)$  is a possibly empty pending message slot.

The cache state list  $C$  is structurally identical to a directory, but describes different information. It describes which kind of access rights the node has to other nodes' memory. The directory and the cache lists separate the memory space orthogonally, like the rows and columns in matrixes. The cache list entry  $C(j)$  of the  $i$ th node corresponds to the directory entry  $D(i)$  of the  $j$ th node.

The node records a request message from memory  $k$  in the the pending message slot  $P(k)$  when it makes a request. The node can only send a cache request when the corresponding entry of the pending message list is empty.

*Program.* The program  $Prg_n$  for node  $N_n$  is a list of Ask messages, which is an abstraction of the activity that the node  $N_n$  will produce when it executes. The program can only contain shared requests and exclusive requests. The message id of each message in the program is unique throughout the World comprising the DASH system.

*Updates on Mappings.* We have several structures that map  $[N]$  to some set of objects. Directories, cache state lists in a RAC, and node lists in a World are such structures. The notation  $D[k \mapsto v]$  denotes the mapping that acts like  $D$  except for  $k$ , which maps to  $v$ . That is,

$$D[k \mapsto v](x) = \begin{cases} D(x) & \text{when } x \neq k \\ v & \text{when } x = k. \end{cases}$$

## 2.2 Overview of the Protocol

We briefly describe the protocol by stating what kind of reaction each type of message causes.

- On receiving  $\overset{i \rightarrow j}{\text{Ask}}_t \tau k$  message, the receiving node  $j$  may grant the request by sending out an Ack or refuse the request by sending out a Nak. The node  $j$  may forward the message or send a writeback message to  $k$  if needed.
- On receiving  $\overset{i \rightarrow j}{\text{Ack}}_t \tau k s$  message, the node completes the request by updating the RAC, only if the message is still pending in the RAC. The Ack message is ignored when there is no corresponding pending message in the RAC. The node updates  $C(k)$ , its cache rights to memory  $k$ , to  $\tau$  and clears the  $k$ 'th pending message slot  $P(k)$ , which contains the corresponding pending message (e.g.  $\overset{j \rightarrow i}{\text{Ask}}_t \tau k$ ). For exclusive acknowledgements, the node may invoke additional invalidation requests depending on the value  $s$ .
- On receiving  $\overset{i \rightarrow j}{\text{Nak}}_t \tau k$  message, node  $j$  will resend the Ask message  $t$  if the message is still pending in the RAC. Otherwise the Nak message is ignored.
- On receiving  $\overset{i \rightarrow j}{\text{Wrb}}_t \tau k$ , node  $j$  will update its directory entry, and may generate a forwarding Ack message in case of an invalidating writeback. The DASH cache coherence protocol performs two way acknowledgement for forwarded exclusive requests. These forwarded exclusive requests generate invalidation writebacks.

We describe the protocol in detail, in Sections 2.3 and 2.4.

## 2.3 Communication Rules

Communication rules describe how the World makes legal transitions. We write  $(\mathcal{N}, bs) \rightarrow (\mathcal{N}', bs')$  when there is a transition step from World  $(\mathcal{N}, bs)$  to World  $(\mathcal{N}', bs')$  by one of the rules  $recvMsg$ ,  $sendMsg$ , or  $grntMsg$ . We call these

rules communication rules since they capture the behaviors of the nodes sending and receiving messages on the bus. The communication rules depend on the reaction rules of the form  $N_n, bs \models b \Downarrow N'_n, bs'$ , which defines the behavior of the node when it receives message  $b$ , possibly changing the bus configuration  $bs$  to  $bs'$  as well as the state of the node  $N_n$  to  $N'_n$ . We will describe the reaction rules in detail, in Section 2.4.

$$\frac{\mathcal{N}(n), bs \models b \Downarrow N'_n, bs'}{(\mathcal{N}, b : bs) \rightarrow (\mathcal{N}[n \mapsto N'_n], bs')} \text{recvMsg}$$

The *recvMsg* rule can fire on the World  $(\mathcal{N}, b : bs)$ , which has at least one message on the bus, and one of its nodes  $\mathcal{N}(n)$  can receive the first message  $b$  to react according to one of the reaction rules. The resulting World after the transition step is the same as the original World except for the changes from the reaction.

$$\frac{\mathcal{N}(n) = (D, (C, P), \overset{n \rightarrow j}{\text{Ask}}_t \tau k : ms) \quad C(k) \neq \tau \quad P(k) = \perp}{(\mathcal{N}, bs) \rightarrow (\mathcal{N}[n \mapsto (D, (C, P[k \mapsto \overset{n \rightarrow j}{\text{Ask}}_t \tau k]), ms)], bs : \overset{n \rightarrow j}{\text{Ask}}_t \tau k)} \text{sendMsg}$$

$$\frac{\mathcal{N}(n) = (D, (C, P), \overset{n \rightarrow j}{\text{Ask}}_t \tau k : ms) \quad C(k) = \tau \quad P(k) = \perp}{(\mathcal{N}, bs) \rightarrow (\mathcal{N}[n \mapsto (D, (C, P), ms)], bs)} \text{grntMsg}$$

Either the *sendMsg* rule or the *grntMsg* rule can fire on the World  $(\mathcal{N}, bs)$  when one of its nodes can fetch the next request from its program and send it on the bus. A node can only fetch the next request, if it is the first message of the program and there are no conflicting pending requests asking the cache for the same target address. The *grntMsg* rule may fire when the node already has the cache right  $\tau$  to the memory  $k$ , which meets the request  $\overset{n \rightarrow j}{\text{Ask}}_t \tau k$ . Since the node already has the cache right  $\tau$  for the new request, there is no need to make a redundant request. Otherwise, when the node does not have the cache right  $\tau$  for the new request, the *sendMsg* rule may fire.

## 2.4 Reaction Rules

Reaction rules describe how a node reacts to the message input. We write  $N_n, bs \models m \Downarrow N'_n, bs'$ , when node  $N_n$  can receive message  $m$  and update its state from  $N_n$  to  $N'_n$  and change the bus configuration from  $bs$  to  $bs'$ . The node  $N_n$  can only receive messages whose destination is  $n$ , such as  $\overset{i \rightarrow n}{\text{Ask}}_t \tau k$ . When the node receives a message, it changes its state by updating the directory  $D$  and the RAC of the from  $(C, P)$  and replies to the input message by sending new messages on the bus.

We use the context notation on the node to emphasize the changing parts of the node in the reaction rules. The formula  $N_n[C], bs \models m \Downarrow N_n[C'], bs'$  denotes that the reaction only updates the cache of the node. That is, the state of the node changes from  $(D, (C, P), ms)$  to  $(D, (C', P), ms)$ . Similarly, the formula  $N_n[D][P], bs \models m \Downarrow N_n[D'][P'], bs'$ , with two contexts, denotes that the reaction updates both the directory and the pending message list of the node. That is, the state of the node changes from  $(D, (C, P), ms)$  to  $(D', (C, P'), ms)$ .

#### 2.4.1. Ask Rules (handling requests).

*Invalidation Request.* Invalidation requests succeed unless the node receiving the request has exclusive rights to the requested memory.

$$\frac{C(k) \neq \text{Ex}}{N_n[C], bs \models \overset{i \rightarrow n}{\text{Ask}_t \text{In } k} \Downarrow N_n[C[k \mapsto \text{In}]], bs : \overset{n \rightarrow i}{\text{Ack}_t \text{In } k} 0} \text{askInAck}$$

$$\frac{C(k) = \text{Ex}}{N_n[C], bs \models \overset{i \rightarrow n}{\text{Ask}_t \text{In } k} \Downarrow N_n[C], bs : \overset{n \rightarrow i}{\text{Nak}_t \text{In } k}} \text{askInNak}$$

*Shared Requests and Exclusive Requests.* Shared Requests and Exclusive Requests share common traits. Some of the rules are exactly the same for both kinds of requests: The rules *askNotInNakP* and *askNotInNakF* that cause Naks, and the rule *askNotInFwd* which forwards the requests apply to both shared and exclusive requests. Other rules have counterparts sharing similar structure: the rules *askShAckF* and *askExAckF* are paired, and the rules *askShRetry* and *askExRetry* are paired. The only significant difference between handling shared requests and exclusive requests are in the rules *askShAck* and *askExAck*.

$$\frac{\tau \neq \text{In} \quad \text{msgid}(P(k)) \neq t}{N_n[P], bs \models \overset{i \rightarrow n}{\text{Ask}_t \tau k} \Downarrow N_n[P], bs : \overset{n \rightarrow i}{\text{Nak}_t \tau k}} \text{askNotInNakP}$$

The request fails when the receiving node already has another pending message in the pending message slot for the requested memory.

$$\frac{P(k) = \perp \quad \tau \neq \text{In} \quad k \neq n \quad C(k) \neq \text{Ex}}{N_n[(C, P)], bs \models \overset{i \rightarrow n}{\text{Ask}_t \tau k} \Downarrow N_n[(C, P)], bs : \overset{n \rightarrow i}{\text{Nak}_t \tau k}} \text{askNotInNakF}$$

The forwarded request fails when the receiving node already released its exclusive rights to the requested memory. Observe that the rule *askNotInNakF* is for the forwarded request because of  $k \neq n$  one of its premises.

$$\frac{P(n) = \perp \quad \tau \neq \text{In} \quad j \neq n \quad D(j) = \text{Ex}}{N_n[D][P], bs \models \overset{i \rightarrow n}{\text{Ask}_t \tau n} \Downarrow N_n[D][P], bs : \overset{i \rightarrow j}{\text{Ask}_t \tau n}} \text{askNotInFwd}$$

The receiving node forwards a request when some other node  $j$  has exclusive rights to the requested memory, which belongs to the receiving node.

$$\frac{P(k) = \perp \quad k \neq n \quad C(k) = \text{Ex}}{N_n[(C, P)], bs \models \overset{i \rightarrow n}{\text{Ask}_t \text{Sh } k} \Downarrow N_n[(C[k \mapsto \text{Sh}], P)], \quad bs : \overset{n \rightarrow k}{\text{Wrb}_t \text{Sh } i} : \overset{n \rightarrow i}{\text{Ack}_t \text{Sh } k} 0} \text{askShAckF}$$

$$\frac{P(k) = \perp \quad k \neq n \quad C(k) = \text{Ex}}{N_n[(C, P)], bs \models \overset{i \rightarrow n}{\text{Ask}_t \text{Ex } k} \Downarrow N_n[(C[k \mapsto \text{In}], P)], \quad bs : \overset{n \rightarrow k}{\text{Wrb}_t \text{In } i} : \overset{n \rightarrow i}{\text{Ack}_t \text{Ex } k} 0} \text{askExAckF}$$

A node acknowledges a forwarded request when it has exclusive rights to the request memory, and sends a writeback to the owner of the memory for releasing its access rights.

$$\frac{P(n) = \perp \quad D(n) = \text{Ex}}{N_n[D][(C, P)], bs \models \overset{i \rightarrow n}{\text{Ask}_t \text{Sh } n} \Downarrow N_n[D][(C[n \mapsto \text{Sh}], P)], \quad bs : \overset{n \rightarrow n}{\text{Wrb}_t \text{Sh } i} : \overset{n \rightarrow i}{\text{Nak}_t \text{Sh } n}} \text{askShRetry}$$

$$\frac{P(n) = \perp \quad D(n) = \text{Ex}}{N_n[D][(C, P)], bs \models \overset{i \rightarrow n}{\text{Ask}_t \text{Ex } n} \Downarrow N_n[D][(C[n \mapsto \text{In}], P)], \quad bs : \overset{n \rightarrow n}{\text{Wrb}_t \text{In } i} : \overset{n \rightarrow i}{\text{Nak}_t \text{Ex } n}} \text{askExRetry}$$

The rules *askShRetry* and *askExRetry* fire when the receiving node has access rights caching its own memory, and if that access right  $\tau$  is of higher precedence than the access rights it can grant the request. In such a case, the receiving node releases its own cache rights by sending a writeback to itself, and lets the requester retry by sending a Nak to the requesting node.

$$\frac{\forall k. D(k) \neq \text{Ex} \quad P(n) = \perp}{N_n[D][P], bs \models \overset{i \rightarrow n}{\text{Ask}_t \text{Sh } n} \Downarrow N_n[D[i \mapsto \text{Sh}]] [P], bs : \overset{n \rightarrow i}{\text{Ack}_t \text{Sh } n} 0} \text{askShAck}$$

A node can acknowledge a shared request to its memory when no node has exclusive access to its memory, and the requested node does have any pending requests for its own memory. In such a case, the node acknowledges the shared request and updates the directory entry to Sh.

$$\frac{\forall k. D(k) \neq \text{Ex} \quad P(n) = \perp \quad \vec{m} = \{ \overset{i \rightarrow j}{\text{Ask}_t \text{In } n} \mid \forall j \neq i. D(j) = \text{Sh} \}}{N_n[D][P], bs \models \overset{i \rightarrow n}{\text{Ask}_t \text{Ex } n} \Downarrow N_n[D[j \mapsto \text{In}]_{\forall j. D(j) = \text{Sh}} [i \mapsto \text{Ex}]] [P], \quad bs : \overset{n \rightarrow i}{\text{Ack}_t \text{Ex } n} \mid \vec{m} : \vec{m}} \text{askExAck}$$

A node can acknowledge an exclusive request to its memory when no node has exclusive cache rights to its memory and the requested node does not have any pending requests for its own memory. In such a case, the node acknowledges the exclusive request and updates the directory entry to **Ex**. In addition, the node sends invalidation requests to all other nodes with shared cache rights to its memory and invalidates corresponding directory entries.

#### 2.4.2. Writeback Rules (handling writebacks).

Writeback messages are generated by either exclusive requests or shared requests. A node sends a writeback when it releases the access rights for its cache to an access rights of lower precedence: from **Ex** to **Sh**, from **Ex** to **ln**, or from **Sh** to **ln**. (See *askShAckF*, *askExAckF*, *askShRetry*, and *askExRetry* in Section 2.4.1.) Ask messages for shared rights may generate sharing writebacks and Ask messages for exclusive rights may generate invalidating writebacks.

$$\frac{}{N_n[D], bs \models \text{Wrb}_t \text{ ln } j \Downarrow N_n[D[n \mapsto \text{ln}]], bs} \text{ wrbIn}$$

$$\frac{}{N_n[D], bs \models \text{Wrb}_t \text{ Sh } j \Downarrow N_n[D[n \mapsto \text{Sh}]], bs} \text{ wrbSh}$$

When a writeback comes from the same node that receives the writeback, the node  $N_n$  only needs to update the directory slot  $n$ , which is the slot for granting the cache rights to the receiving node itself.

$$\frac{i \neq n}{N_n[D], bs \models \text{Wrb}_t \text{ ln } j \Downarrow N_n[D[i \mapsto \text{ln}, j \mapsto \text{Ex}]], bs : \text{Ack}_t \text{ Ex } n \ 0} \text{ wrbInFwd}$$

$$\frac{i \neq n}{N_n[D], bs \models \text{Wrb}_t \text{ Sh } j \Downarrow N_n[D[i \mapsto \text{Sh}, j \mapsto \text{Sh}]], bs} \text{ wrbShFwd}$$

When a writeback comes from another node, which is not the receiving node, the node  $N_n$  should update both the directory slot  $i$ , which is for the sender of the writeback, and the directory slot  $j$ , which is the node that caused the writeback. The *wrbInFwd* rule generates an **Ack**. We explain this **Ack** in Section 2.4.3.

#### 2.4.3. Ack Rules (handling acknowledgements).

*Invalidation Acks.* A node waits for invalidation **Acks** only after receiving an exclusive **Ack** with a positive invalidation pending counter, which occurs when there are nodes with shared access rights to the requested memory. (See *askExAck* in Section 2.4.1.) The node can claim its exclusive rights only after it receives all invalidation **Acks** from the nodes that had shared cache to the same memory.

$$\frac{P(k) = \text{Ack}_t \text{ Ex } k \ s \quad s > 1}{N_n[P], bs \models \text{Ack}_t \text{ ln } k \ 0 \Downarrow N_n[P[k \mapsto \text{Ack}_t \text{ Ex } k \ (s-1)], bs} \text{ ackIn}$$

$$\frac{P(k) = \overset{i \rightarrow n}{\text{Ack}}_t \text{ Ex } k \ 1}{N_n[(C, P)], bs \models \overset{j \rightarrow n}{\text{Ack}}_t \text{ In } k \ 0 \Downarrow N_n[(C[k \mapsto \text{Ex}], P[k \mapsto \perp])], bs} \text{ ackIn1}$$

When such node receives an invalidation Ack, it decrements the pending invalidation counter from  $s$  to  $s - 1$ . The rule *ackIn1* is for the last Ack.

*Sharing Acks.* On receiving a shared Ack, the node updates its cache rights to Sh and clears the pending message slot.

$$\frac{P(k) = \overset{n \rightarrow i}{\text{Ask}}_t \text{ Sh } k}{N_n[(C, P)], bs \models \overset{j \rightarrow n}{\text{Ack}}_t \text{ Sh } k \ 0 \Downarrow N_n[(C[k \mapsto \text{Sh}], P[k \mapsto \perp])], bs} \text{ ackSh}$$

*Exclusifying Acks.* When there are no other nodes with shared access to memory  $k$ , an Ack message with the pending invalidation counter 0 will arrive, as in the following rules *ackEx0K*, *ackEx0*, and *ackEx0F*.

$$\frac{P(k) = \overset{i \rightarrow j}{\text{Ask}}_t \text{ Ex } k}{N_n[(C, P)], bs \models \overset{k \rightarrow n}{\text{Ack}}_t \text{ Ex } k \ 0 \Downarrow N_n[(C[k \mapsto \text{Ex}], P[k \mapsto \perp])], bs} \text{ ackEx0K}$$

$$\frac{P(k) = \overset{i \rightarrow j}{\text{Ask}}_t \text{ Ex } k}{N_n[(C, P)], bs \models \overset{i \rightarrow n}{\text{Ack}}_t \text{ Ex } k \ 0 \Downarrow N_n[(C[k \mapsto \text{Ex}], P[k \mapsto \perp])], bs} \text{ ackEx0}$$

If the Ack is either from  $k$ , the owner of the memory, or from  $i$ , the original request target, the node can claim its exclusive rights immediately.

$$\frac{P(k) = \overset{i' \rightarrow j}{\text{Ask}}_t \text{ Ex } k \quad i \neq k \quad i \neq i'}{N_n[(C, P)], bs \models \overset{i \rightarrow n}{\text{Ack}}_t \text{ Ex } k \ 0 \Downarrow N_n[(C[k \mapsto \text{Ex}], P[k \mapsto \overset{i \rightarrow j}{\text{Ask}}_t \text{ Ex } k])], bs} \text{ ackEx0F}$$

Otherwise, if the Ack is neither from the memory owner nor from the request target, it must be the case that the request was forwarded before it was acknowledged. Then, the node has to wait for another Ack from the memory owner, which is generated by the rule *wrbInFwd* in Section 2.4.2.

When other nodes have shared access rights to  $k$ , an Ack message with the pending invalidation counter  $s > 0$  will arrive.

$$\frac{P(k) = \overset{n \rightarrow i}{\text{Ask}}_t \text{ Ex } k \quad s > 0}{N_n[(C, P)], bs \models \overset{i \rightarrow n}{\text{Ack}}_t \text{ Ex } k \ s \Downarrow N_n[(C, P[k \mapsto \overset{i \rightarrow n}{\text{Ack}}_t \text{ Ex } k \ s])], bs} \text{ ackExAck}$$

Then, the node should update its pending message slot with the Ack message generated by the rule *askExAck* in Section 2.4.1, and wait for  $s$  invalidation Acks to acquire exclusive rights.

#### 2.4.4. Drop Rules (discarding outdated Acks or Naks).

We discard Acks and Naks when no corresponding message is pending in the RAC.

$$\frac{P(k) = \perp \quad \text{ackDropB}}{N_n[P], bs \models \overset{i \rightarrow n}{\text{Ack}}_t \tau k 0 \Downarrow N_n[P], bs} \quad \frac{\text{msgid}(P(k)) \neq t \quad \text{ackDropT}}{N_n[P], bs \models \overset{i \rightarrow n}{\text{Ack}}_t \tau k 0 \Downarrow N_n[P], bs}$$

$$\frac{P(k) = \perp \quad \text{nakDropB}}{N_n[P], bs \models \overset{i \rightarrow n}{\text{Nak}}_t \tau k \Downarrow N_n[P], bs} \quad \frac{\text{msgid}(P(k)) \neq t \quad \text{nakDropT}}{N_n[P], bs \models \overset{i \rightarrow n}{\text{Nak}}_t \tau k \Downarrow N_n[P], bs}$$

#### 2.4.5. Retry Rules (retry on Naks).

On receiving Nak, we retry.

$$\frac{P(k) = \overset{n \rightarrow j}{\text{Ask}}_t \text{Ex } k}{N_n[P], bs \models \overset{k \rightarrow n}{\text{Nak}}_t \text{In } k \Downarrow N_n[P], bs : P(k)} \quad \text{nakInRetryAsk}$$

When an invalidation request fails, we retry the exclusive request, which caused the invalidation request. Invalidation requests are not allowed in programs but only invoked by exclusive requests. (See *askExAck* in Section 2.4.1.)

$$\frac{P(k) = \overset{i \rightarrow n}{\text{Ack}}_t \text{Ex } k \ s}{N_n[P], bs \models \overset{k \rightarrow n}{\text{Nak}}_t \text{In } k \Downarrow N_n[P], bs : \overset{n \rightarrow i}{\text{Ask}}_t \text{Ex } k} \quad \text{nakInRetryAck}$$

$$\frac{\text{msgid}(P(k)) = t \quad \tau \neq \text{In}}{N_n[P], bs \models \overset{i \rightarrow n}{\text{Nak}}_t \tau k \Downarrow N_n[P], bs : P(k)} \quad \text{nakNotInRetry}$$

We have two rules, *nakInRetryAsk* and *nakInRetryAck*, for the invalidation Nak, because the pending message slot may contain not only Ask messages but also a Ack messages while handling exclusive requests. (See *ackExAck* in Section 2.4.3.)

### 3 Haskell Implementation

We implement the operational semantics of Section 2 in Haskell. We use this implementation as a reference implementation to check our intuitions as we develop the formal proofs of the protocol properties. We formalized the operational semantics at the same time we implemented the protocol in Haskell. It was an iterative process, correcting the errors in the operational semantics by comparing them with the implementation, and clarifying the implementation by studying the operational semantics and rereading the original paper [LLG<sup>+</sup>90].

We implement each rule as a function in Haskell. We use the list monad to simulate the system's nondeterministic behavior. Our simulator takes possible Worlds as input and computes all possible Worlds that are reachable by legal transitions from any of the World in the input.

#### 3.1 Data Definition

We define data types in Haskell according to the notation of Section 2.1.

```
data St = In | Sh | Ex deriving (Eq,Ord)

data Msg = Ask MsgID From To St About
         | Ack MsgID From To St About Shares
         | Nak MsgID From To St About
         | Wrk MsgID From To St Because
         deriving (Eq,Ord)
type From=Int; type To=Int; type About=Int
type Shares=Int
type Because=Int
type MsgID = (Int,Int)

data World = W [Node] Bus deriving (Eq,Ord,Show,Read)
data Node = N Int Dir RAC [Msg] deriving (Eq,Ord)
type Bus = [Msg]
type Dir = [St]
type RAC = (Dir,PMS)
type PMS = [Maybe Msg]
```

We define `MsgID`, the data type for message id, as a pair of integers. The first integer is the source node, which initiated the request of that message id. The second integer is the sequence number within the program. The information of the source node and the sequence number within the program of the source node ensures the uniqueness of message ids.

#### 3.2 Implementing the Rules of the Operational Semantics

We define three functions to implement the rules of the operational semantics in Sections 2.3 and 2.4.

```

recvMsg :: Node -> Bus -> (Node, Bus)
sendMsg :: Node -> Bus -> (Node, Bus)
runMsg  :: Msg -> Node -> Bus -> (Node, Bus)

```

The functions `recvMsg` and `sendMsg` implements the communication rules in Section 2.3. The `recvMsg` function corresponds to the *recvMsg* rule. The `sendMsg` function corresponds to the *sendMsg* and *grntMsg* rules. Both functions take two arguments, a node and a bus of the original World, and possibly produce a new node state and a bus configuration if any of the corresponding rules are applicable. When there are no corresponding rules that matches the input, the functions just return a pair of the unchanged input arguments. Note that the shape of these two functions are slightly different to the shape of the communication rules. The functions take only one node as its first argument rather than taking the entire list of nodes in the World. These functions capture the action of a particular node in the World, and examines whether it can possibly make a transition.

The function `runMsg` implements the reaction rules in Section 2.4. The `runMsg` function takes three arguments, and produces a new node state and a new bus configuration. The three input arguments are the original state of the node, original configuration of the bus, and the input message. Since *sendMsg* depends on the reaction rules, the function `sendMsg` calls the function `runMsg`.

```

recvMsg node [] = (node, [])
recvMsg node@(N n _ _ _) bus@(b:bs)
  | n /= msgTo b = (node, bus)
  | otherwise    = runMsg b node bs

```

### 3.3 Simulation

We use the list monad to simulate the nondeterministic system behavior. The list monad is a common idiom that allows us to write the code that simulates nondeterministic in the form of a deterministic sequential programming. The program blocks after the `do` looks like a deterministic program but it calculates all possible results and collects them into a list.

The function `step` implements zero or one transition of the World. It nondeterministically chooses a node of the original World, and nondeterministically applies either of the `sendMsg` or `recvMsg` function. The function `step'` implements one transition step of the World, since it excludes zero step transitions from the result of `step`. The function `stepM` lifts `step'` to apply one step transition from a set of multiple possible Worlds to the next set of possible Worlds.

```

step :: World -> [World]
step (W nodelist bus) = nub $ do
  node <- nodelist
  cmd <- [sendMsg, recvMsg]
  let (node'@(N n _ _), bus') = cmd node bus
      nodelist' = update n node' nodelist
  return (W nodelist' bus')

step' w = [w' | w' <- step w, w/=w']

stepM :: [World] -> [World]
stepM worldlist = nub $ do w <- worldlist
  case step' w of
    [] -> return w
    ws -> ws

```

We show some sample test data simulation.

We start with a world called `worldSE` of two processing nodes.

```

Main> worldSE
W [0:II II [Nothing,Nothing] [Ask (0,1) 0 1 S 1],
  1:II II [Nothing,Nothing] [Ask (1,1) 1 1 E 1]]
[]

```

Both processing nodes have all invalid entries in their directories (II), all invalid cache entries (II), and no pending messages in the pending message list ([Nothing, Nothing]). The bus is also empty ([]). Both nodes will be asking for the same memory location. Node 0 will run the program [Ask (0,1) 0 1 S 1], which has one request message asking for shared access rights to node 1's memory. Node 1 will run the program [Ask (1,1) 1 1 E 1], which has one request message asking for exclusive access rights to node 1's memory.

We do 1 step simulation for all possible paths of nondeterministic choice.

```

Main> test 1 [worldSE]
W [0:II II [Nothing,Just (Ask (0,1) 0 1 S 1)] [],
  1:II II [Nothing,Nothing] [Ask (1,1) 1 1 E 1]]
[Ask (0,1) 0 1 S 1]

W [0:II II [Nothing,Nothing] [Ask (0,1) 0 1 S 1],
  1:II II [Nothing,Just (Ask (1,1) 1 1 E 1)] []]
[Ask (1,1) 1 1 E 1]

```

The simulation shows two possible next steps from `worldSE`. The former shows the world that has sent node 0's request on the bus first. The latter shows the world that has sent node 1's request on the bus first. Observe the changes of the programs, the pending message lists, and the bus from the initial state `worldSE`.

There are 6 possible worlds in step 3.

```
Main> test 3 [worldSE]
W [0:II II [Nothing, ...
W [0:II IS [Nothing, ...
W [0:II II [Nothing, ...
W [0:II II [Nothing, ...
W [0:II II [Nothing, ...
W [0:II II [Nothing, ...
```

In step 11, we reach to all the final states.

```
Main> test 11 [worldSE]
W [0:II IS [Nothing,Nothing] [],1:SS IS [Nothing,Nothing] []] []
W [0:II II [Nothing,Nothing] [],1:IE IE [Nothing,Nothing] []] []
```

There are two final states: one world with both nodes having shared access rights, and one with node 1 having the exclusive access rights.

Below is the simulation for the world `worldSE` of two processing nodes, both asking for exclusive access rights to node 1's memory.

```
Main> worldEE
W [0:II II [Nothing,Nothing] [Ask (0,1) 0 1 E 1],
  1:II II [Nothing,Nothing] [Ask (1,1) 1 1 E 1]]
[]
```

```
Main> test 1 [worldEE]
W [0:II II [Nothing,Just (Ask (0,1) 0 1 E 1)] [],
  1:II II [Nothing,Nothing] [Ask (1,1) 1 1 E 1]]
[Ask (0,1) 0 1 E 1]
```

```
W [0:II II [Nothing,Nothing] [Ask (0,1) 0 1 E 1],
  1:II II [Nothing,Just (Ask (1,1) 1 1 E 1)] []]
[Ask (1,1) 1 1 E 1]
```

```
Main> test 11 [worldEE]
W [0:II IE [Nothing,Nothing] [],1:EI II [Nothing,Nothing] []] []
W [0:II II [Nothing,Nothing] [],1:IE IE [Nothing,Nothing] []] []
```

## 4 Using $\Omega$ mega to Enforce Properties

After building an implementation in Haskell, we turned the Haskell program into a  $\Omega$ mega program in order to prove properties of the implementation. Hindley-Milner type system To describe the properties in  $\Omega$ mega, we enriched the types that we use in the Haskell programs. Once we describe the property using types, the  $\Omega$ mega type system enforces the properties by type checking.

### 4.1 Static Properties on Directories

Directories have two static properties, which must hold all the time. First, a directory can have at most one exclusive entry. Second, a directory cannot have both exclusive entries and shared entries. Our strategy is to reflect the structure of the directory on the type level to enforce the properties on the three primitive directory manipulating operations: `invalidate`, `exclusify`, and `share`.

Consider the types of two directories that meet this property.

```
#[In', In']v : StLIST #2 #[In, In]t
#[In', Sh']v : StLIST #2 #[In, Sh]t
```

The first list `#[In', In']v` is a value that denotes a directory with two invalid entries. Its type `StLIST #2 #[In, In]t`, indexed by the type level list `#[In, In]t`, reflects the shape of the value `#[In', In']v`. The second list `#[In', Sh']v` is a value that denotes a directory with an invalid entry and a shared entry. Its type `StLIST #2 #[In, Sh]t`, indexed by the type level list `#[In, Sh]t`, reflects the shape of the value `#[In', Sh']v`. Suppose we want to apply the `exclusify` operation on both directories to update the first entry to `Ex'` in each of the lists. Applying `exclusify` on the first directory `#[In', In']v` results in a valid directory `#[Ex', In']v`. However, applying `exclusify` on the second directory `#[In', Sh']v` results in an invalid directory `#[Ex', Sh']v`. We want the `exclusify` operation to be applicable to the first directory, but not the second. In general, the `exclusify` operation should only be applicable to the directories of all `In`'s (e.g. `#[In', In', In']v`). We can enforce such a property by constraining the input argument type of `exclusify` to be the type level lists of all `In`s (e.g. `StLIST #3 #[In, In, In]t`).

We define the type of the `exclusify` function as following to give a restriction on the type of the input directory, and to describe the resulting directory's type precisely based that input directory's type.

```
exclusify :: Nat' k -> StLIST n {repl n In}
          -> StLIST n {set k Ex {repl n In}}
```

Given a natural number `k` and a directory of type `StLIST n {repl n In}`, the `exclusify` function produces a new directory of type `StLIST n {set k Ex {repl n In}}`. The index `k` indicates the entry to update in the given input directory. The `exclusify` function only applies to the directories of all `In`'s, because the type function application `{repl n In}` ranges over the type level lists of all `In`s.

The functions `repl` and `set`, appearing in the curly braces, are type functions. We can define functions over types just as we define functions over values. The type function application `{repl n In}` replicates `In` `n` times to construct a type level list. The type function `{set k Ex l}` updates the `k`th entry of `l` to `Ex`. The  $\Omega$ mega interpreter can reason about type functions. The normal forms of several type function applications are illustrated below.

```
prompt> :n {repl #3 In}          prompt> :n {set #1 Sh {repl #3 In}}
#[In,In,In]t                    #[In,Sh,In]t
```

To illustrate  $\Omega$ mega's reasoning ability, we try applying the `exclusify` operation to each of the two directories.

```
prompt> exclusify #0 #[In',In']v
#[Ex',In']v : StLIST #2 #[Ex,In]t
prompt> exclusify #0 #[In',Sh']v
... some messages ...
... on type checking ... => have no solution
```

$\Omega$ mega accepts the first one but fails to type check the second one as expected.

#### 4.1.1. The Directory Structure

We now give the definitions in  $\Omega$ mega that we used in the example above.

*Data Types* The following types represent the directory structure.

```
kind St = In | Sh | Ex

data St' :: St ~> *0 where
  In'  :: St' In
  Sh'  :: St' Sh
  Ex'  :: St' Ex

kind List a = Nil | Cons a (List a) deriving List(t)

data StLIST :: forall n . n ~> List St ~> *0 where
  StNIL  :: StLIST Z Nil
  StCONS :: St' st -> StLIST n l -> StLIST (S n) (Cons st l)
  deriving List(v)
```

The types `St'` and `StLIST` are singleton types, which have only one value for a particular type. (e.g. `In'` is the only value that has type `St' In`) The `deriving List(t)` clause in on the `List` kind definition allows us to use the list like syntax `#[In,Sh]t`, which is identical to `Cons In (Cons Sh Nil)`. The `StLIST` type definition also has `deriving List(v)` clause as well. The `StLIST` type is indexed by its length represented by a singleton natural number of the kind `Nat` and a list of the kind `List`, which reflects the shape of its value.  $\Omega$ mega provides

singleton natural numbers with the kind `Nat` and the type `Nat'`, which use the same unary notation with a zero (`Z`) and a successor (`S`) for both values and type level objects: `Z` has type `Nat'` `Z` and `S Z` has type `Nat'` (`S Z`). We also have a decimal shorthand notation for the natural numbers: `#0` for `Z` and `#3` for `S(S(S Z))`.

*Functions* We define basic functions over directories, which are singleton lists. The functions over singleton lists, such as `countEx'`, use the type functions over type lists, such as `countEx` in their type declarations.

```
countEx :: List St ~> Nat
{countEx Nil} = Z
{countEx (Cons In xs)} = {countEx xs}
{countEx (Cons Sh xs)} = {countEx xs}
{countEx (Cons Ex xs)} = S {countEx xs}

countEx' :: StLIST n l -> Nat' {countEx l}
countEx' StNIL = Z
countEx' (StCONS In' xs) = countEx' xs
countEx' (StCONS Sh' xs) = countEx' xs
countEx' (StCONS Ex' xs) = S (countEx' xs)
```

We can view the relationship between the functions and the type functions in two ways. First, the type function `countEx` captures the property that `countEx'` counts the number of exclusive entries in a directory. Second, when such type using the type function `countEx` is given, the implementation of `countEx'` is a proof that there exists a way to count the number of exclusive entries in a directory.

#### 4.1.2. Directory Manipulating Primitives.

We define directory manipulating primitives that ensure the validity of the directory by defining functions that have the type which precisely describes the relationship between the inputs and the outputs. Any function definition that type checks is a proof that the relationship hold.

Types in  $\Omega$ mega are a static properties because  $\Omega$ mega is a statically typed language. So, some function can only be used in contexts where certain static properties hold. However, some of the calling code may have to perform dynamic checks to test whether the static properties hold. Because of this, we often have two versions of some functions.

*Static Version.* We know of two ways to constrain the use of the function statically. The first is to use type functions, such as `repl` and `set`, to constructively constrain the types. The second is to use static properties, such as `Equal {countEx l}`, to qualify the types [Jon94]. The types of the primitive operations `invalidate` and `exclusify` use type functions. The type of the primitive operation `exclusify` uses both type functions and qualified types. Since

these static primitives have static constraints, the  $\Omega$ mega type checker will guard against any illegal use of these functions at compile time.

```
invalidate :: Nat' k -> StLIST n l -> StLIST n {set k In l}
invalidate n = setSt' n In'
```

```
exclusify :: Nat' k -> StLIST n {repl n In}
           -> StLIST n {set k Ex {repl n In}}
exclusify n = setSt' n Ex'
```

```
share :: Equal {countEx l} Z => Nat' k -> StLIST n l
      -> StLIST n {set k Sh l}
share n = setSt' n Sh'
```

The type of `invalidate` states that any `StLIST` is a legal input, and the result is the same shape except that `kth` entry has been set to `In`. The type of `exclusify` states that only the `StLISTs` that has shape constructed by `{repl n In}` are legal inputs, and the result is the same shape except that `kth` entry has been set to `Ex`. The type of `share` states that any `StLIST` whose shape has no `In` can be a legal input, and the result of type `StLIST n {set k Ex {repl n In}}` is the same as the input except that the `kth` entry has been set invalid.

*Dynamic Version.* We build the dynamic version of the primitives by using the static version plus a dynamic test. We can only apply the static version of the primitives to directories that are valid inputs. However, we often need to test whether a given directory is valid for the function input, and then apply the primitive function to the the test succeeds.

monad maybeM -- to use the Maybe monad

```
allin :: StLIST n l -> Maybe (Equal l {repl n In})
allin StNIL          = return Eq
allin (StCONS Sh' xs) = fail ""
allin (StCONS Ex' xs) = fail ""
allin (StCONS In' xs) = do { e@Eq <- allin xs; return Eq }
```

```
tryExclusive :: Nat' k -> StLIST n l -> Maybe (StLIST n {set k Ex l})
tryExclusive n l = do { e@Eq <- allin l; return (exclusify n l) }
tryShare :: Nat' k -> StLIST n l -> Maybe (StLIST n {set k Sh l})
tryShare n l = case countEx' l of Z      -> Just (share n l)
                               (S x) -> Nothing
```

The `tryExclusive` function builds an explicit proof object `e@Eq` at compile time, which certifies that the input list contains all `In`s, using the auxiliary function `allin`. On the other hand `tryShare` relies on the type checker to implicitly build a proof. For both functions, the type checker has a proof that the static primitives are invoked only when the inputs are valid.

Note that the Curry-Howard correspondence on the basic list operation `countEx'`, which looks trivial, is playing a key role while building the `tryShare` function. Because of the type of `countEx'`, the type checker builds a static proof `Equal {countEx 1} Z` for the case branch `Z`, which guarantees that applying the `share` operation is safe when the result of the counting is zero at runtime. Therefore, we can use the static primitive `share` in the case branch `Z`. The case analysis on values also becomes a case analysis on types because of the Curry-Howard correspondence. (Although it is implicit, the `tryExclusive` function also relies on a such case analysis because `monadM`, the Maybe monad, has case analysis internally.) Since it is both a dynamic and static guard, `Omega` detects the violation at compile time. This prevents accidental use of the `share` function in the wrong context, such as in the `(S x)` case branch or outside the case expression.

The functions `tryExclusive` and `tryShare` will only apply the static primitives when the inputs are valid. Otherwise they return the value `Nothing`, which stands for failure or error. The `Maybe` type represents a computation that may fail. Therefore, the protocol implementation always maintains the invariants for directories when we use these primitives.

## 4.2 Approach for Liveness Properties

We have developed an approach for proving liveness properties but have not yet completed the proof. The complete proof will be the future work. We will singleton types for the messages and the communication channel used in the DASH cache coherence protocol, and formulate invariants on the quantity of messages in the system, just as we define the singleton types `St'` and `StLIST` to describe the static properties of directories.

We introduce the definitions and some claims we hope to prove.

**Definition 1.** *Stable World.* A World  $(\mathcal{N}, bs)$  is stable when no node has pending requests. That is,  $\forall i, n \in [N]. P(i) = \perp$  where  $\mathcal{N}(n) = N_n[P]$ .

**Definition 2.** *Valid World.* A World  $(\mathcal{N}, bs)$  is valid when all the directories are valid and the corresponding entries of directories and RACs are coherent. That is,  $\forall i, j \in [N]. D(j) = C(i)$  where  $\mathcal{N}(i) = N_i[D]$ ,  $\mathcal{N}(j) = N_j[C]$ .

**Definition 3.** *Valid Directory.* A directory is valid when it has at most one exclusive entry, and does not have both shared entries and exclusive entries. The validity of directories is a safety property that holds all the time. We can enforce this kind of properties using `Omega`. (See Section 4.1.)

*Claim.* Progress. When  $w$  is a reachable World from a valid stable World, then  $w$  is either a valid stable state or there exists a transition  $w \rightarrow w'$ . Since  $w'$  is also a reachable state, we can apply the progress rule again. The progress property states that the system always makes progress until it reaches a final valid stable state.

*Claim.* Subject Reduction: When  $w$  is a valid stable World and  $w \rightarrow^* w'$ , then  $w'$  is not an invalid stable World. That is, the World  $w'$  is either a valid stable World or an unstable World.

*Claim.* Correct Request Completion: When a request completes, it must complete correctly so that the requesting node acquires the very access rights for the very memory it has requested for. More formally:

Let  $w = (\mathcal{N}, bs)$  be a reachable World from a valid stable World and  $P_w(i) = m$ , where  $P_w = P$  such that  $\mathcal{N}(n) = N_n[P]$  for some  $i, n \in [N]$ , and  $m$  is a message requesting for an access right  $\tau$  for memory  $k$ . If  $w \rightarrow^* w'$  such that  $P_{w'}(i) = \perp$  and  $C_{w'}(k) = \tau$  where  $\mathcal{N}'(n) = N'_n[(C_{w'}, P_{w'})]$  for  $w' = (\mathcal{N}', bs')$ , and  $\forall w''$  between  $w$  and  $w'$ ,  $P_{w''}(i) \neq \perp$ , then  $\forall w''$  between  $w$  and  $w'$ ,  $P_{w''}(i)$  is also a message requesting for an access right  $\tau$  for memory  $k$  where  $\mathcal{N}''(n) = N''_n[P_{w''}]$  for  $w'' = (\mathcal{N}'', bs'')$ .

## 5 Related Works

There are many studies on distributed algorithms and protocol verification. We mention two of them, which gave us insights into our work, and we compare them with our approach.

Mur $\varphi$  is a model checker developed at Stanford, where the DASH system also originated. Norris Ip [Ip96] used Mur $\varphi$  to verify several distributed algorithms and shared memory models including the DASH cache coherence protocol. Ip and his colleagues' work on the DASH cache coherence protocol was up to very specific details including some optimizations such as DMA transactions. They rediscovered errors in the protocol using Mur $\varphi$ , when DMA was used carelessly. They also worked on state reduction to verify larger systems in a reasonable amount of time [ID93]. Their approach can only give definitive answers on fixed size systems, while our approach provides proofs for systems of arbitrary size.

The Theory of Distributed Systems group at MIT use the Input Output Automata (IOA) to formalize distributed algorithms. They used both the model checking approach and the theorem proving approach, and a hybrid approach [UL06]. They are developing tools that can generate programs from the specifications as well as proving the specifications. We use a general purpose language  $\Omega$ mega to implement the programs and prove the program properties at the same time. We hope our approach is more attractive to programmers.

## 6 Conclusions & Future Work

Our contributions are the following. We gave a concrete operational semantics based on transitions, for the DASH cache coherence protocol, which did not have a formal semantics in the original paper. We use the  $\Omega$ mega type systems to describe and prove the properties of the protocol implementation. We prove the safety property of the system, with the help of the rich type system of  $\Omega$ mega. We are trying to apply subject reduction, which is a classical method in programming languages, to prove liveness properties of the system. Our approach is scalable since we prove the properties independently over the size of the system.

We are working on more complete proof on liveness properties based the operational semantics. It becomes harder to compose the program that has more precise types. But once we successfully compose the program, we have a strong guarantee that the program has the properties represented using types. We will refine our claims on liveness properties to make it more suitable to describe in the  $\Omega$ mega type system. Afterwards, we plan to remove the simplifications that made our model less parallel.

We will be improving the quality of the  $\Omega$ mega implementation as well. While we were working on the cache coherence protocol with  $\Omega$ mega, We have identified some bugs in the  $\Omega$ mega implementation, and have been trying to fix them. The type system of  $\Omega$ mega is non-decidable because of recursive type functions which may not terminate. However, we are very sure that the type functions used in this work are all terminating functions, since they are all primitive recursive functions such as computing the length of a list structure.

Our next step is to polish our work based on a clear definition of what *correctness* means for cache coherence protocols. In this paper, we introduced some properties that should intuitively hold without giving definition for correctness. The correctness of cache coherence protocols are the memory models of the multiprocessor systems. We plan to study the relation between cache coherence protocol properties and memory models, starting with more simple protocols in more simple multiprocessor systems such as snooping protocols in unified memory access systems.

Our work shows how  $\Omega$ mega can be used as a tool that supports theorem proving to build proofs for practical real world problems. The theorem proving approach has been used sparingly for system verification because there are few tools that are easy to learn and use. The programming pattern we used to define directory primitives prevents typical errors that can easily occur in program maintenance. One of the typical errors in program maintenance is trying to use a function in the wrong context, where the assumptions for that function do not hold. (e.g. using a binary search on an unsorted sequence.) Traditional type systems cannot catch these kind of errors. The  $\Omega$ mega type system can check whether a certain expression is used in the right context.

## 7 Acknowledgements

Thanks to Tim Sheard and the  $\Omega$ mega group members including Chuan-kai, John, and Tom, for reviewing the ideas and the paper. Thanks to Nancy Lynch and the Theory of Distributed Systems group for explaining the concepts of the IOA and pointers to references on distributed systems. Thanks to Dave Archer for pointers to the literatures on memory coherency. Their help were all great resource for our work and the Research Proficiency Examination.

We also appreciate valuable feedbacks from the Research Proficiency Examination Committee. John Walpole mentioned the importance of studying memory models, which helped us planning our future work. James G. Hook gave general feedbacks on the Research Proficiency Examination paper, which helped writing this technical report.

## References

- [CLS03] Jason F. Cantin, Mikko H. Lipasti, and James E. Smith. The complexity of verifying memory coherence. In *SPAA*, pages 254–255. ACM, 2003.
- [Dij68] Dijkstra. A constructive approach to the problem of program correctness. *BIT: BIT*, 8, 1968.
- [Flo67] Robert W. Floyd. Assigning meanings to programs. In J. T. Schwartz, editor, *Mathematical Aspects of Computer Science*, volume 19 of *Symposia in Applied Mathematics*, pages 19–32. American Mathematical Society, Providence, RI, 1967.
- [Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.
- [How80] W. A. Howard. The formulae-as-types notion of construction. In *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 479–490. Academic Press, Inc., New York, N.Y., 1980.
- [HP06] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach, 4th ed.* Morgan Kaufmann Publishers Inc., Palo Alto, CA 94303, September 2006. ISBN ??
- [ID93] C. Norris Ip and David L. Dill. Better verification through symmetry. In David Agnew, Luc J. M. Claesen, and Raul Camposano, editors, *CHDL*, volume A-32 of *IFIP Transactions*, pages 97–111. North-Holland, 1993.
- [Ip96] C. Norris Ip. State reduction methods for automatic formal verification. Thesis CS-TR-96-1578, Stanford University, Department of Computer Science, December 1996.
- [Jon94] Mark P. Jones. *Qualified Types: Theory and Practice*. Cambridge University Press, 1994.
- [LLG<sup>+</sup>90] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. The directory-based cache coherence protocol for the DASH multiprocessor. In *ISCA '90*, pages 148–159, 1990.
- [LLG<sup>+</sup>92] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Wolf-Dietrich Weber, Anoop Gupta, John Hennessy, Mark Horowitz, and Monica S. Lam. The Stanford DASH multiprocessor. *IEEE Computer*, 25(3):63–69, March 1992.
- [Lyn96] Nancy Lynch. *Distributed Algorithms*. Morgan Kaufmann, San Francisco, CA, 1996.
- [She04] Tim Sheard. Languages of the future. In *OOPSLA Companion*, pages 116–119. ACM, 2004.
- [UL06] Shinya Umeno and Nancy A. Lynch. Proving safety properties of an aircraft landing protocol using I/O automata and the PVS theorem prover: A case study. In *FM*, volume 4085 of *LNCS*, pages 64–80. Springer, 2006.