Guarded Atomic Actions for Haskell

Austin Clements and Yang Zhang

December 13, 2006

Abstract

The guarded atomic actions model is a programming model introduced in the Bluespec high-level hardware description language. The model expresses parallel behavior with a high degree of modularity and composability. In this project, we present an implementation of guarded atomic actions as a library for the Haskell software programming language, thus introducing the guarded atomic action model of parallelism into the realm of software.

1 Introduction

Bluespec [1] has demonstrated the applicability of the model of guarded atomic actions for expressing parallelism in the realm of hardware. In this model, parallel behavior is implemented by synthesizing hardware (with inherent parallelism) from a set of independent rules for atomic state manipulations, each protected by a condition known as a guard. Multiple atomic rules can act in parallel as long as they do not manipulate the same state, resulting in a model whose behavior appears equivalent to running the rules sequentially.

A lack of such composability has been a source of weakness in parallel programming when using traditional procedural abstraction and abstract data types. We are interested in applying the guarded atomic action model to the realm of software. To this end, our contribution consists of the construction of a combinator library for Haskell that implements the guarded atomic action model.

2 Guarded Atomic Actions

The library consists of two modules: *ATM* (Atomic Transactional Memory) and *GAA* (Guarded Atomic Actions). *ATM* provides transactions and registers, whereas *GAA* provides guarded atomic actions (rules and modules). User code is expected to import *GAA*, which re-exports *ATM* and provides a much higher-level model of parallelism.

2.1 Registers

There are two kinds of registers: stateful registers, which can be thought of as persistent, mutable containers of data; and computed registers, which represent transient values computed from the values other registers. Stateful registers are created explicitly with the *mkReg* function, which takes the name of the register (for debugging purposes) and its initial value:

$$mkReg :: String \rightarrow a \rightarrow IO\ (Register\ a)$$

Computed registers are created by taking advantage of the monadic properties of *Register*. Specifically, monadically binding a register reads its value (whether it is a stateful or computed register). This allows arbitrary values and computations to be lifted into the *Register* monad. For example, *return* 42 creates the computed register whose value is simply 42. Supposing a an b are registers, $liftM2\ (+)\ a\ b$ yields the computed register whose value is the sum of the values of a and b.

2.2 Actions

An action is a value that can be lifted into the *IO* monad to atomically affect the values of register. The basic atomic action is a register write, created with the $\Leftarrow$ operator. The right operand can be either a stateful register or a computed register. The left operand is the stateful register into which the value of the right operand will be stored when the action is executed. For example, $a \Leftarrow b$ creates the action that will write the value stored in register b to register a. More complicated actions can be built up using the following *action combinators*:

- Two actions can be composed *sequentially* with the $<\!\!>$ operator. When the resulting action is executed, the effects of the left operand will be

visible to the right operand. The effect of the overall action is the effect of the left operand, followed by the effect of the right operand. If either action aborts, the overall action will abort. This matches the intuition behind standard sequential operations within a transaction.

- Two actions can be composed *in parallel* with the <|> operator. When the resulting action is executed, the effects of the left operand will *not* be visible to the right operand, and vice-versa. The effect of the overall action is the union of the effects of the two operands. If both operands modify the same register, a double write error will occur. If either action aborts, the overall action will abort. This combinator naturally suggests, though does not require, parallel execution of its sub-actions.

- A *predicated* action can be constructed with the ?− combinator. This behaves like **if** in a regular, imperative language. When the resulting action is executed, the register passed as the left operand will be read. If its value is True, then the effect of the overall action is the effect of the right operand. If its value is False, then the action has no effect.

- Finally, a *guarded* action can be constructed with the *whenA* combinator. This is similar to ?−, except that when the predicate evaluates to False, the action will abort (causing the transaction as a whole to abort).

All of these action combinators have the property of closure: given atomic actions for arguments, they will produce an atomic action. Specifically, the types of the action combinators, as well as the write operator, are as follows:

$$
\begin{array}{lll}
(\Leftarrow) & :: \mathit{Register}\ a & \to \mathit{Register}\ a \to \mathit{Action} \\
(<\!\!>), (<|>) & :: \mathit{Action} & \to \mathit{Action} \to \mathit{Action} \\
(?-), \mathit{whenA} & :: \mathit{Register}\ \mathit{Bool} \to \mathit{Action} & \to \mathit{Action}
\end{array}
$$

2.3 Modules and Rules

A *module* consists of set of *rules*, where each rule is an atomic action. Executing a module consists of firing its rules repeatedly until no more rules can fire successfully (that is, all guards fail), at which point no further progress can be made, so the execution stops and returns. These rules appear to fire atomically and sequentially with respect to each other. However, the ambiguity implied by the execution model leaves many choices up to the rule scheduler, such as what order to execute rules in, and whether or not to actually execute the rules in parallel (as long as the illusion of serializability and atomicity is maintained).

We provide two scheduler implementations. The simpler implementation schedules rules sequentially, but has the advantage of guaranteeing fairness. The parallel implementation can take advantage of the natural ambiguity in the execution model to execute multiple rules simultaneously, but currently does not provide any prevention against starvation.

2.4 Example

The following example demonstrates the complete implementation of a module for computing the greatest common denominator of two numbers.

```
mkGCD      :: Int → Int → IO (Module, Register Int)
mkGCD a b =
  do x ← mkReg "x" a
     y ← mkReg "y" b
     let swap     =
           rule (whenA ((x ⩾ y) ⩓ (y ⩄ 0̄))
                       (x ⇐ y <|> y ⇐ x))
         subtract =
           rule (whenA ((x ⩽ y) ⩓ (y ⩄ 0̄))
                       (y ⇐ y − x))
     return (mkModule [swap, subtract], x)
  where (⩾) = liftM2 (>)
        (⩽) = liftM2 (⩽)
        (⩄) = liftM2 (≠)
        (⩓) = liftM2 (∧)
        (−) = liftM2 (−)
        0̄   = return 0
gcd      :: Int → Int → IO Int
gcd a b =
  do (gcd, result) ← mkGCD a b
     runModule gcd
     readCReg result
```

mkGCD creates a module that will compute the GCD of two numbers. It first creates two registers, then a module consisting of a *swap* rule and a *subtract* rule. For convenience, all of the numerical operators needed in these rules are lifted into the *Register* monad (denoted by the use of an overbar on the symbol). It returns both the module, and the register that will hold the ultimate result of the computation. *gcd* uses the module created by *mkGCD* to actually compute GCD's. *runModule* runs the module to completion (that is, until no more rules can fire), at which point the register returned by *mkGCD* will contain the value of the GCD.

3 Implementation

The system is divded into two layers, a low-level general atomic transactional memory system, and a high-level parallel rule system. The interface to the low-level transactional memory system is similar to that provided by Haskell's STM library [2], allowing actions to be executed simply by lifting them into the IO monad. The parallel rule system provides a high-level parallelism abstraction on top of this in which rules consist of guarded actions that are continuously being scheduled until no further rules can fire.

Underneath the covers, both *Action* values and the *Register* monad are fairly thin veils over the *IO* monad. However, there is no exposed mechanism to lift general *IO* operations into these data types, which allows the atomic transaction libraries to maintain two key guarantees:

1. Any IO operation implied by the *Register* monad will never mutate any state. The operation of reading from a register is embedded in the *IO* monad purely for the ability to *read* from mutable state. This guarantee means that IO operations performed by register reads can be ignored when rolling-back the effects of an aborted transaction.

2. Any IO operation implied by an *Action* will always have a way to *reverse* that operation. For example, there is no action to display to the screen because there is no (simple) dual IO operation that would undo the effects of such an operation. Likewise, user input in an *Action* is impossible because there is no way to "undo" user input.

3.1 Atomic Transactional Memory

The transactional memory system is similar in many respects to Haskell's STM library, but provides slightly different operations. Notably, it does not provide support for standard nested transactions, in which a nested transaction can abort without aborting the overall transaction (though support for this could be added). Because our system provides the notion of a "guarded" action, when a guard fails, the entire transaction aborts. However, our transactional memory system does provide *parallel* nested transactions to accommodate the parallel action combinator, as described in Section 2.2.

Support for parallel nested transactions causes the implementation to differ significantly from typical transactional memory systems, such as Haskell's STM. In particular, this makes a log-based approach unnatural because a log implies a total ordering of operations, which can be difficult to obtain when a parallel action combinator is introduced. Consider, for example, the simple action $r0 \Leftarrow r1 <|> r1 \Leftarrow r0$, which exchanges the contents of two registers. Using a log would require all read operations to be performed before write operations to correctly describe the behavior of this action, which becomes increasingly difficult to implement when both sequential and parallel combinators are used together. Thus, either a fair amount of log finagling is necessary, or a different approach altogether.

Regular nested transactions form a stack-like structure at any given point in the computation, which fits well with the structure of a log. However, parallel nested transactions form a tree-like structure in which two nested transactions can be *siblings*, both seeing the effects of the parent, but neither seeing the effects of each other. While it is possible to implement this with logging, we instead opt for a transaction mechanism whose design reflects the tree-like nesting structure yielded by the parallel action combinator.

3.1.1 Transaction Trees

Each uncommitted transaction in the system at any given time (including root transactions, as well as nested transactions) is assigned a unique *transaction ID* and carries a *transaction context*. We use increasing integers as transaction ID's, but any form of unique identifier would suffice.

The transaction tree is a structure which reflects the current execution state. Each encounter with a parallel combinator forks two child nodes, one for each parallelly-executing transaction. Sequentially combined transactions execute in-place. Thus, each leaf node represents a transaction which is concurrently executing with all other leaf node transactions. The context of a transaction is its path to the root transaction.

Additionally, every register maintains a *transaction map* from transaction ID's to values. Included in this map is the special *committed transaction* (denoted C in the following diagrams), which maps to the committed value of the register. Excluding the committed transaction, all other values in the map represent values written by uncommitted transactions.

A nested transaction is created by simply generating a fresh transaction ID and prepending it to the transaction context. The context keeps track of the

chain of nestings leading up to the current nested transaction. When a register is read, the context allows it to search upward through the path that led to the current transaction to find the last parent that wrote to the desired register. Contexts always derive from the committed transaction, so reads from registers that have not been modified by the transaction will read the committed value of the register. Writes are always performed to the current transaction ID so they are not visible to parent or sibling transactions.

When a nested transaction completes, the values it has written are *merged* up to the uncommitted values of its parent, making them visible to the parent transaction and the transaction's siblings. As further nested transactions complete, these writes continue to merge up the transaction tree until they are ultimately merged up into the committed transaction at the completion of the action.

Consider the following code snippet, which creates two registers and performs some operations on them.

```
do r0 ← mkReg "r0" 'a'
   r1 ← mkReg "r1" 'b'
   transact (r0 ⇐ return 'c' <>
             (r0 ⇐ return 'd' <|>
              r1 ⇐ return 'e') <>
             r1 ⇐ return 'f')
```

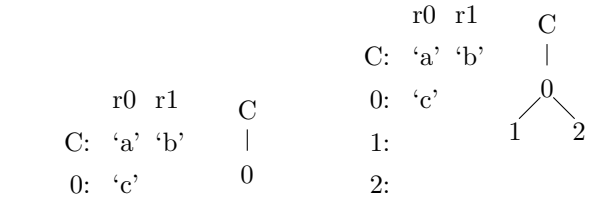
The process carried out during the execution of this action is shown in Figure 1.

These merge operations are performed using *merge sets*. Each register has an associated *merge operator* that merges a value up the register's transaction tree, given the transaction ID of the child, the transaction ID of the parent, and, optionally, the transaction ID of a sibling to check for double-write errors. As a transaction proceeds, it builds up a merge set containing the merge operators of all registers written to. Whenever a nested transaction needs to commit up the transaction tree, it executes the merge operators in the accumulated merge set. Unlike a log, the size of the merge set is proportional to the number of registers written to, not the number of writes, making it both smaller and bounded by the total number of registers.

This single merge operation is sufficient to implement all necessary manipulations of the transaction tree. The parallel merge operation takes advantage of all three arguments to merge each child up the tree, checking for conflicts between the two. The final commit is performed by merging up from the transaction root to the committed transaction without a sibling transaction. Even abort cleans up the transaction

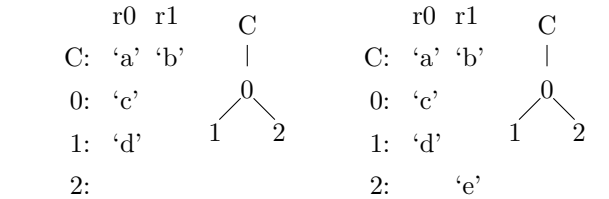
(a) Initially, both registers contain only their committed values.

(b) When the transaction begins, an uncommitted root transaction containing no changes is forked from the committed transaction.



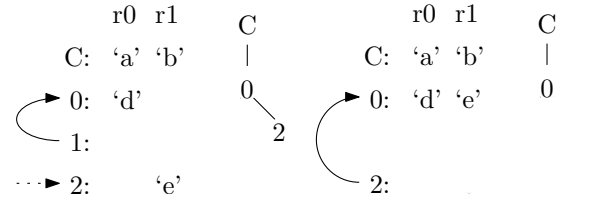
(c) Executing *r0* ⇐ *return* `'c'` in the current transaction.

(d) The parallel combinator forks two sibling nested transactions in which it can execute child actions.



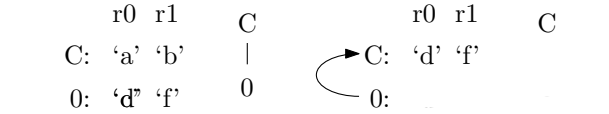
(e) The parallel combinator executes *r0* ⇐ *return* `'d'` in the left transaction context...

(f) ..and *r1* ⇐ *return* `'e'` in the right transaction context.



(g) Once both children have finished, the parallel combinator merges up the modifications from the left child, checking for conflicts with the right child...

(h) ...and merges up the changes from the right child.



(i) The sequential combinator then executes the *r1* ⇐ *return* `'f'` action.

(j) Finally, the action completed successfully, so the modifications accumulated in uncommitted root transaction are merged up into the committed transaction.

Figure 1: Execution of the example action. The left side of each diagram shows register transaction maps. The right side shows the transaction context tree.

tree by performing a merge into a special bit-bucket parent transaction ID.

3.1.2 Atomicity

While the above provides transactional semantics for the actions and action combinators in a sequential system, it does not ensure atomicity and serializability when multiple actions can be executed simultaneously. Before describing the specific implementation in our system, we begin with a set of constraints on how actions can be executed in order to guarantee atomicity.

Let P be the set of pending actions that are waiting to begin execution and $S_1, S_2, \ldots$ be the sequence of actions currently executing, in the order they started execution. $W(\alpha)$ denotes the conservative write set of action α (computed by joining together the registers being written to by all write actions within the action). Let $S_w = \cup_i W(S_i)$ denote the union of the write sets of all currently executing actions. Given these definitions, the following rules are sufficient to guarantee atomicity:

Issue rule. The set of issuable actions I at any given point is $\{\alpha \in P \mid W(\alpha) \cap S_w = \emptyset\}$ (ie, all pending rules that do not have a write conflict with any currently executing rule). Note that only one action from I can be issued at a time, because the act of issuing an action changes I. This rule guarantees that the write sets of all executing actions are pair-wise disjoint, so there can be no write-write conflicts. If I is empty and P is non-empty, then issuing any further action would cause a write-write conflict, so wait until some action in S commits and try issuing again.

Read rule. Let r be a register being read by some action S_i. If $r \in W(S_j)$ for some $j < i$, then the read must block until S_j commits. If $r \in W(S_i)$, then the appropriate uncommitted value must be read. Otherwise, the read must return the current committed value of the register. Intuitively, when reading a register that is being written to by a transaction issued before this transaction, the read must block until the value written by the earlier transaction is available in order to maintain serializability.

Commit rule. An action S_i is only allowed to commit once there are no uncommitted actions S_j where $j < i$. This guarantees that actions commit in the same order they are issued. This, in conjunction with the read rule, guarantees reads are consistent with the issue-order serialization of actions because the effects of one transaction cannot become visible before the effects of an earlier transaction become visible.

The issue rule is implemented by coupling a unique write lock with each register. When an action tries to issue, it attempts to atomically (within a critical section) acquire the write locks for all registers in its conservative write set. If all locks are successfully acquired, then the action is issued. Otherwise, all locks are released, and the issue logic waits for the release of the write lock on the register it failed to acquire before retrying the lock set. The release of this register is not sufficient to prove that the action can be issued, but it is necessary, so waiting on it prevents simply spinning until the action can proceed.

Each root transaction is assigned an increasing *sequence number* that indicates the order in which transactions acquired their write sets. Each register's write lock stores the sequence number of the root transaction that locked it, thus allowing the implementation of the read rule. When a read is performed, the write lock of the desired register is examined, and then the read blocks if necessary.

Finally, the commit rule is implemented by forming a chain of running root transactions, in the order they are issued. The system contains one *commit token* that is passed down this chain as root transactions commit. Each running root transaction knows its own *commit token box* and the commit token box of the next root transaction. When a root transaction is ready to commit, it waits for the commit token to arrive in its box, commits, then passes the commit token into the next box. Simple commit token passing would require aborted transactions to linger simply in order to pass the commit token down the chain. To avoid this problem, a commit token box can also contain a *back pointer* to another commit token box. When a transaction aborts, it places a back pointer in the next root transaction's commit box that points back to its own commit box. When that later root transaction is ready to commit, it sees this back pointer, follows it, and begins waiting for the commit token in the previous transaction's box (this repeats if multiple transactions in a row abort). Write locks are released only once the transaction has successfully committed or aborted.

Note that this algorithm is *not* sufficient to guarantee fairness or lockout-freedom in high-contention situations. These guarantees are left to higher layers.

3.2 Modules and Rules

Guarded atomic actions are implemented atop our atomic transactional memory system. A "rule" is simply a thin wrapper around an action. A module consists of a set of rules, with the intention that the order of the rules does not matter. When a module is run, it begins firing the rules in the module until no more rules can fire, at which point it returns from running the module. There is a great deal of ambiguity with respect to how rules are scheduled. We implemented two schedulers: a simple sequential scheduler, and a more sophisticated parallel scheduler.

3.2.1 Sequential Scheduler

The sequential scheduler does not take advantage of any natural parallelism that may arise from running multiple rules simultaneously. It simply executes the rules sequentially, repeatedly, until every rule aborts. Once every rule aborts, no more progress can be made by further rule firings, so the scheduler terminates.

3.2.2 Parallel Scheduler

We also implemented a parallel scheduler that takes advantage of the ability of the underlying transactional memory system to execute actions in parallel. Our implementation is relatively naïve and does not attempt to guarantee fairness or lockout-freedom, though one possible approach to this is discussed in Section 4. For each rule, a thread is spawned that repeatedly attempts to fire that rule. The threads synchronize with each other to guarantee that, if a transaction aborts, that transaction will not be fired again until some other transaction commits, a slightly conservative rule that ensures an abortive transaction does not fire twice in a row without any changes to the state of the registers. If all transactions have aborted and are waiting for some transaction to commit, then no more progress can be made, so the scheduler stops executing the module and returns.

4 Future Directions

Our current parallel scheduler is simple, but fails to guarantee fairness between the rules. By associating with each rule the number of times its action has been executed, it would be possible to implement a "bounded bypass" scheduler that ensured fairness by not letting any rules fall too far behind. If some rule was continuously blocked by other rules, this would force the scheduler pipeline to eventually drain until no rules conflicted with that rule, allowing it to proceed.

In addition to better fairness properties for the parallel scheduler, the current implementation cannot take advantage of hardware parallelism. Unfortunately, Concurrent Haskell does not currently support SMP parallelism in the IO monad, so increased hardware parallelism will not improve the performance of modules. However, any code written for the module system would, without modification, gain improved performance with the addition of support for SMP parallelism to Concurrent Haskell.

5 Conclusion

In this project, we implemented a guarded atomic action programming model for Haskell. This model of parallelism permits greater composability than is possible with such traditional concurrency primitives as threads and locks. The current trend toward multicore machines encourages more highly parallel applications, and this programming model allows developers to more easily express such parallelism.

References

[1] L. Augustsson et al. Bluespec: Language definition, 2001.

[2] Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. Composable memory transactions. In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 48–60, New York, NY, USA, 2005. ACM Press.