CS421 Project: Composable Memory Transactions

Matthew Kennedy,
mk65@illinois.edu

July 23, 2024

Abstract

As programmers, we often take multiple threads and the idea of concurrency for granted. However, this is no easy task to implement. How did researchers in the mid 2000s look to solve various concurrency related issues in functional programming?

1 Introduction

Innovative problem solving is often necessary when dealing with novel tasks. Putting lightning into a flattened rock and giving it the ability to think was the type of innovative thinking we needed in order to use powerful programs like MS Paint to change the future. More importantly, when these same rocks started having multiple cores and threads to think with, we needed a better way to create programming languages in order to properly utilize those resources. This issue is called "concurrency", and is a difficult concept to execute well when writing code. What is even more difficult than creating programs and writing code to handle concurrency, is creating a programming language that can handle concurrency, that people will use to create programs and writing code to handle concurrency. That is to say, how does one address concurrent programming within functional programming interpreters, languages, and patterns?

2 Composable Memory Transactions

2.1 The problem

Imagine, for a moment, the year is 2005; Youtube has recently been invented and flip phones are all the rage. Additionally, according to Harris et al. (2005), concurrent programming was an ongoing notoriously difficult task. "Current lock-based abstractions are difficult to use and make it hard to design computer systems that are reliable and scalable. Furthermore, systems built using locks are difficult to compose without knowing about their internals." (p. 1). There were some promising alternatives to lock-based approaches, such as software transaction memory (STM) which can "perform groups of memory operations atomically", and has promising advantages such as, "automatic roll back on exceptions or timeouts, freedom from deadlock and priority inversion, and freedom from the tension between lock granularity and concurrency." (p. 1). However, these same approaches using STM still had their own set of issues to work through, such as preventing threads from bypassing transaction interfaces.

2.2 The (potential) solution

The authors use this paper to outline how some of these issues can be addressed using a new expression of transaction memory using Concurrent Haskell, which the authors describe as a “declarative language” (p. 1). Additionally, Harris et al. (2005) propose methods to utilize Haskell’s type system for strong type guarantees, as well as transactions that are compositional and can be glued together to form larger transactions. Next, the authors explore a new modular form of blocking which presents to programmers as a simple function called “retry”, which can occur anywhere within the transaction and block until an alternative execution path becomes possible. This removes the need for a programmer to identify the condition under which the transaction can run to completion.

Combining these improvements, the authors hoped to achieve a ”qualitative improvement in language support for modular concurrency, similar to the improvement in moving from assembly code to high-level language. Our main war-cry is compositionality: a programmer can control atomicity and blocking behaviour in a modular way that respects abstraction barriers.” (Harris et al., 2005, p.1). The authors hope to tackle the issue of conflict between abstraction and concurrency as well with this solution.

3 Background

Concurrency is a broad and somewhat abstract term for programming; it can be applied to databases, storage systems, etc. Harris et al. (2005) are specifically dealing with *internal concurrency*, which is ”concurrency between the threads interacting through memory in a single process”. This specific type of concurrency comes with it’s own set of problems that were briefly mentioned in 2.1. To expand more here, lock-based programming is inherently complex and does not scale well, making it challenging to design reliable concurrent systems for threads in a single core. Programmers need to manage conflicting operations, avoid deadlocks, and balance lock granularity for performance, but lock-based programs suffer from poor composability and often require exposing lock management to clients, complicating program design and leading to potential deadlocks and race conditions. Composing blocking operations like waiting for multiple data inputs is also challenging, necessitating higher-level management.

The first solution Harris et al. (2005) suggest is as follows: ”Rather than fixing locks, a more promising and radical alternative is to base concurrency control on *atomic memory transactions*, also known as *transactional memory*. For example, with memory transactions we can manipulate a hash table thus:

```
    atomic { v:=delete(t1,A); insert(t2,A,v) }
```

and to wait for either p1 or p2 we can say

```
    atomic { p1 ‘orElse‘ p2 }
```

These simple constructions require no knowledge of the implementation of insert, delete, p1, or p2, and they continue to work correctly if these operations may block, as we shall see.” (Harris et al., 2005, p.2)

Transactions have some caveats that need to be addressed - first, they must not do anything irrevocable, since they may be re-run automatically. Secondly, blocking is not composable. Third, no previous transactional memory supports choice (Harris et al., 2005, p.2). The authors address all 3 issues by using the declarative language Concurrent Haskell, which is an extension of Haskell 98.

For the sake of brevity, I won't go over all the benefits listed by the authors, but the key idea is, "a value of type IO a is an *I/O action* that, when performed, may do some I/O before yielding a value of type a" (Harris et al., 2005, p.2). The authors also utilize *monadic bind* combinator by gluing togther I/O actions. These I/O actions can also perform external input/output and operations on mutable cells, while Concurrent Haskell porovides support on threads, each independently performing input/output, using a function forkIO, which takes an IO action as its argument.

4 Composable transactions

With all the previous description on Concurrent Haskell and composable transactions, the authors argue that Concurrent Haskell's purely declarative nature and expressive type system make it an ideal setting for studying transactional memory. They introduce the concept of transactional variables, and describe how transactions are defined and executed using the STM monad, where the atomic function ensures that transactions run atomically with respect to each other.

A key innovation is the retry function, which allows a transaction to block until it can proceed without the programmer needing to specify the blocking condition, thus maintaining abstraction. This function ensures that the transaction will restart automatically when the variables it depends on are modified. Harris et al. (2005) also reference the orElse function, which allows the composition of alternative transactions, enabling a form of non-blocking wait similar to Unix's select system call but in a more compositional manner. Additionally, they discusses the handling of exceptions within transactions, ensuring that exceptions abort transactions without causing side effects. The formal operational semantics of these constructs are provided, demonstrating their behavior and ensuring that the proposed system maintains the properties of atomicity and compositionality, thus significantly improving the modularity and reliability of concurrent programming.

4.1 Examples and application

Harris et al. (2005) highlight how this model simplifies the development of complex concurrent operations and high level concurrency abstractions. They showcase how MVars, which are mutable variables that can be either empty or full, can be implemented using transactional variables. They demonstrate that takeMVar and putMVar operations can be naturally expressed using the STM monad, providing a simpler and more reliable mechanism for thread communication.

Next, multicast channels are discussed, which allow multiple threads to read from a shared channel without interfering with each other. The implementation uses TVars to represent the channel's buffer and demonstrates how transactions can be composed to read from and write to the channel atomically. This approach ensures that each item written to the channel is received by every read port, and multiple writers and readers can operate concurrently without conflicts.

The merge function is another example discussed, which allows a thread to wait for the first available result from multiple transactions. This is accomplished using the orElse function to combine multiple STM actions, enabling a thread to proceed as soon as any of the composed actions completes.

The examples in this section illustrate the power and flexibility of composable memory transactions, showing that they not only simplify the implementation of concurrent data structures and communication channels but also provide a robust framework for composing complex concurrent operations in a modular and reliable way.

4.2 Formal semantics of STM Haskell

The descriptions provided thus far have been informal. To ensure that such descriptions cover all the combinations of functions that might arise, the authors provide a formal, operational semantics for STM Haskell (Harris et al., 2005, p.6). The semantics is designed to clarify the behavior of the system in all scenarios, including less intuitive cases such as exceptions and retries. The authors begin by introducing the syntax for values and terms in STM Haskell, followed by the definition of the program state, which includes a thread soup (a collection of threads) and a heap (a mapping from references to terms).

The operational semantics are divided into two transition relations: I/O transitions and STM transitions. I/O transitions handle the execution of I/O actions and thread management, ensuring that operations such as reading and writing characters, forking threads, and executing atomic blocks are well-defined. STM transitions handle memory transactions, including reading and writing TVars, creating new TVars, and managing nested transactions.

The formal semantics also define how transactions interact with exceptions and the retry and orElse constructs. When a transaction encounters a retry, it aborts and waits until any of the TVars it accessed are modified. The orElse construct allows the composition of alternative transactions, ensuring that if one transaction retries, the other can proceed.

By providing a formal semantics, the authors ensure that the behavior of STM Haskell is well understood and predictable, enabling developers to reason about their concurrent programs with confidence. This formal foundation is pivotal for verifying the correctness of concurrent operations and for building concurrent systems that are both reliable and modular, and help to maintain the layer of abstraction the authors are striving for.

5 Implementation

After providing theoretical details, as well as informal and formal semantics, Harris et al. (2005) dive into the implementation details of the Software Transactional Memory system in Haskell. This STM is divided into two main layers: the STM operations in the STM Haskell library and the underlying C library integrated into the Haskell runtime system. The STM operations, including STMReadTVar and STMWriteTVar, are responsible for building a thread-local transaction log during a memory transaction, which is used for recording reads and tentative writes. This transaction log is validated and committed atomically when the transaction completes.

The implementation ensures that reads and writes to these TVars remain local to the transaction until it commits, allowing transactions to be aborted simply by discarding the log. Validation checks that the transaction's view of memory is consistent before committing, and this is done atomically to avoid conflicts with other threads.

The retry function causes a transaction to abort and wait until any of the TVars it accessed is updated. This is implemented using wait queues attached to TVars, where transactions can block and be awakened when relevant updates occur. The orElse function supports nested transactions by creating fresh logs for each alternative and merging them as needed.

The authors also touch on the performance of the STM system, noting that initial measurements are encouraging, with STM-based implementations performing comparably to those using traditional synchronization mechanisms like MVars. The STM approach provides clearer, more compositional code with built-in support for handling asynchronous exceptions and maintaining atomicity, while allocating 50% less heap space during testing.

This implementation demonstrates the practicality of STM in a real-world language, showing that it can be integrated into an existing compiler and runtime system with localized changes, offering significant benefits for concurrent programming at the time this paper was written.

5.1 Related work

Before concluding, the authors highlight related studies to the work presented in this paper. The related work shared is focused on transactional models and their integration into programming languages, categorizing this related work into two main areas: transactional memory systems and composable concurrency designs.

Transaction memory systems include areas such as databases and distributed systems, other work on STM as presented in this article, and hardware support. For composable concurrency, topics include concurrent ML, which is (was?) a language directed at composable concurrency, as well as polyphonic C#, which provides a model for synchorinzation that is easy to use but does not support the compositionality and modularity offered by STM (Harris et al., 2005, p.10-11).

6 Conclusion

We started this summary paper by seeing some of the issues rocks face when trying to think about more than a single task at once - it gets messy. Harris et al. (2005) assert that STM offers a significantly improved approach to concurrent programming, particularly through its enhanced composability compared to traditional lock-based methods. They highlight that STM eliminates many common concurrency issues such as deadlocks and provides a more modular and reliable framework for building concurrent applications. By using Haskell as a "laboratory," they demonstrate how STM can be integrated into a practical language, leveraging Haskell's strong type system and declarative nature to ensure safe and efficient transactions.

While the study was conducted within Haskell, the concepts and benefits of STM could be applied to mainstream imperative languages like C# and Java, albeit with some additional work required for mechanisms to prevent side effects within transactions and to support dynamic nesting of atomic blocks. The authors emphasize the potential for STM to raise the abstraction level of concurrent programming, much like high-level programming languages have done for assembly code, and they express confidence in the practicality and scalability of their approach based on initial implementations and performance measurements.

References

Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. 2005. Composable memory transactions. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, page 48–60, New York, NY, USA. Association for Computing Machinery.